基于测试充分性准则的非死锁并发缺陷定位方法

陈 诚 郑 征 王皓钦 乔 禹
(北京航空航天大学自动化科学与电气工程学院 北京 100191)

摘 要 并发程序的非确定性使得其调试工作异常困难。基于程序谱的软件缺陷定位方法虽然能够缓解该情况,但其定位结果依赖于调试信息。针对在此过程中难以获得调试信息及如何选择利用调试信息的问题,提出了一种基于测试充分性准则的缺陷定位方法,该方法包括3个部分:预测满足测试充分性准则的条件;制定相应的测试方案;将收集到的调试信息用于缺陷定位分析。依据此方法,用C#语言实现了缺陷定位工具——ConFinder。在含有实际并发缺陷的程序上进行实验,结果表明该方法可以有效找出引起程序失效的原因并且所得结果具有很好的稳定性。

关键词 并发程序,软件测试,软件缺陷定位
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.030

Non-deadlock Concurrency Fault Localization Approach Based on Adequate Test Criteria

CHEN Cheng ZHENG Zheng WANG Hao-qin QIAO Yu
(School of Automation Science and Electrical Engineering, Beihang University, Beijing 100191, China)

Abstract Concurrency programs are difficult to debug due to their congenital nondeterminism. Fault localization approaches may alleviate such situation. However, their performance heavily depends on the debug information. To alleviate this problem, we proposed a non-deadlock concurrency fault localization approach based on adequate test criteria. Our approach consists of three parts: a prediction of the conditions meeting an adequate test criteria, a scheme of a test plan satisfied with the criteria, and a fault localization analysis using the debug information. Besides, a prototype called ConFinder based on this approach was implemented in C#. Moreover, experiments on programs containing real concurrency bugs were carried out. The result shows that our approach can localize non-deadlock concurrency bugs effectively and has stable performance.

Keywords Concurrency program, Software testing, Software fault localization

1 引言

并发程序运行过程的不确定性使得并发程序的调试成为一项极其困难的工作[1-2]。与传统程序失效不同,并发程序失效不仅需要特定的输入参数,同时还需要满足一定的线程交替运行时机。首先,并发缺陷仅在某些特殊的线程交替运行时触发,而由于程序线程交替运行空间极其巨大,发现程序存在并发缺陷十分困难;其次,当发现程序存在并发缺陷时,从巨大的线程交替运行空间找到导致程序失效的线程交替运行序列也是一项十分费时、费力的工作。然而,并发缺陷又是一种常见的缺陷。在2007年对微软公司700名程序开发人员和程序测试人员的一次调查中发现[3],$\frac{2}{3}$的人员处理过并发程序,其中一半以上的人员每个月至少需要进行一次并发缺陷的测试、调试或修复。另一项研究发现[4],修复一个并发缺陷平均需要73天的时间。因此,自动化调试并发程序成为了一个重要的研究方向。

大多数并发程序失效是由线程交替运行的原子性违背和顺序违背造成的。目前,大量的工作通过对程序进行静态或动态分析,来探索如何更准确地检测程序是否存在并发缺陷[5-7,13,18];也有工作尝试用系统或随机的方法暴露程序中隐含的并发缺陷[8]。使用这些方法有利于检测出程序中存在的并发缺陷,但要找到具体的缺陷位置,如具体的线程交替运行时机或具体的内存访问顺序,仍然是一个巨大的挑战。

为了缓解以上问题,提出一些方法专门用来分析程序运行时记录的数据,以便找到导致程序发生失效的具体原因[10-11,19],这类问题被称为缺陷定位问题。在众多的缺陷定位方法中,基于程序谱的缺陷定位方法由于具有简单性和有效性,已经发展出一个庞大的分支[14-15]。其基本定位思想是通过分析比较程序在成功运行和失败运行中谓词覆盖信息的不同,计算出程序包含的各个谓词的可疑程度。而与程序失效关联越紧密的谓词,其可疑度越高,就越有可能是导致程序失效的根本原因。

为使各个谓词的可疑程度得到明显的区分,该方法需要

到稿日期:2016-10-10 返修日期:2016-12-04

陈 诚(1991-),男,硕士,主要研究方向为软件可靠性与测试,E-mail:clarkchenc@gmail.com;郑 征(1980-),男,博士,副教授,主要研究方向为智能决策、软件可靠性与测试,E-mail:zhengz@buaa.edu.cn;王皓钦(1991-),男,硕士,主要研究方向为复杂网络,E-mail:wanghaoqin1991@foxmail.com;乔 禹 男,博士,主要研究方向为软件可靠性,E-mail:qiaoyu@buaa.edu.cn。

一定数量的成功运行的覆盖信息和失败运行的覆盖信息。然而,并发程序的不确定性使得程序并不容易发生失效。有研究显示[8],在 Apache 服务器程序中,复现一次程序失效大概需要 22 小时。另一方面,基于程序谱计算出的各谓词的可疑程度会由于收集和使用的程序覆盖信息不同而有所差异,其最终会引起各个谓词在每次定位所得可疑度列表中的位置产生波动,这表现为定位结果的不稳定性。此不稳定性会影响定位结果的可靠程度,不可靠的定位结果在实际应用中是没有意义的。

本文为解决并发缺陷定位过程中的上述两个问题,对并发缺陷的主要类型进行分析,从而得到一种充分测试并发程序的准则,并设计了一种可靠且满足该准则的测试方法,该方法能够有效获得程序失败运行的覆盖信息。此外,相比于随机选取的测试覆盖信息,符合该准则的测试所得到的测试覆盖信息将更加充分,可以有效地降低因信息利用不全面而导致的定位结果的不稳定性。本文有以下贡献:

1)提出了基于缺陷模式类型的并发程序测试充分性准则,并且设计了可满足该准则的测试方案。

2)将测试与缺陷定位结合,一方面缓解了在并发缺陷定位时难以获得失败程序运行信息的情况;另一方面,测试准则确定了定位时所用测试选取的截止条件,实验表明该条件可以有效改善缺陷定位的结果。

3)实现了缺陷定位工具 ConFinder,该工具可以对基于 .NET编写的并发程序进行缺陷定位,而且据了解,这是第一个面向.NET 平台的并发程序缺陷定位工具。

4)通过实验对 ConFinder 进行验证。实验表明,与类似的缺陷定位方法相比,ConFinder 在测试程序中的定位结果中表现出了较好的有效性和稳定性。

本文第 2 节介绍并发缺陷的类型及其对应的测试充分性准则;第 3 节详细介绍基于该准则的缺陷定位方法;第 4 节介绍 ConFinder 的基本实现方法;第 5 节结合实验对该定位方法的性能进行验证;最后总结全文。

2 并发缺陷类型及其测试充分性准则

并发缺陷类型基本可以分为:死锁[22]、数据竞争[23]、原子性违背以及顺序违背。其中,原子性违背和顺序违背是造成非死锁性并发缺陷的两个主要原因,在常见的并发缺陷中,这两类缺陷占总缺陷的 97%[4]。本节首先介绍将使用的符号,之后分别对原子性违背和顺序违背两种缺陷类型进行说明。

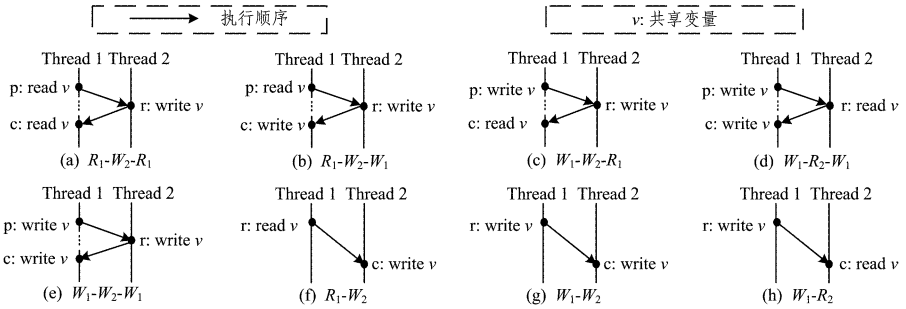
符号说明:用一个四元组$\langle O,t,s,v\rangle$来表示对变量的一次访问,记作 $O_{t,s}(v)$。其中,O 表示访问类型,包括读操作(R)以及写操作(W);t 表示该执行访问的线程编号;s 表示执行访问的语句编号;v 是此次访问的变量。例如,$R_{1,3}(v)$表次来自 1 线程 3 语句对变量 v 的一次读访问。此外,用符号“$\prec$”表示对变量访问的序关系,例如,$R_{1,2}(v)\prec W_{2,3}(v)$表示来自 1 线程的 2 语句和来自 2 线程的 3 语句先后对相同变量 v 进行了读操作和写操作。

(1)原子性违背

在一个原子性的方法或代码块中的操作应当连续执行,若来自另一线程的操作破坏了此连续性,最终导致程序异常,则称该过程为原子性违背。基本的原子性违背由 3 个对相同的变量的访问构成,其中两个来自同一线程,分别记作 *p-access* 和 *c-access*,另一个来自其他线程,记作 *r-access*。*r-access* 在 *p-access* 和 *c-access* 之间执行,破坏 *p-access* 和 *c-access* 的原子性,记为:*p-access*$\prec$*r-access*$\prec$*c-access*。图 1(a)—图 1(e)给出了原子性违背的主要的 5 种模式。

(2)顺序违背

对同一变量的两次访问中至少一次为写操作,若由于缺少合适的同步操作,两次访问没有按照期望的顺序执行,导致程序异常,则称该过程为顺序违背。基本的顺序违背由来自两个不同线程对相同变量的两次访问构成,其中一个用 *r-access* 表示,另一个用 *c-access* 表示,*r-access* 的实际执行在 *c-access*之前,破坏了预期程序的执行结果,记为 *r-access*$\prec$*c-access*,其主要表现类型如图 1(f)—图(h)所示。



(a)—(e)为原子性违背类型,(f)—(h)为顺序违背类型[4,11]

图 1 并发缺陷模式

由于基本的原子性违背由 3 次访问组成,而顺序违背仅需 2 次访问,因此也将原子性违背称为三元(访问)模式,将顺序违背称为二元模式。文献[9]指出,越复杂的缺陷模式会生成越庞大的测试空间,同时调查[4]发现,实际应用中发生在两线程间的缺陷占总缺陷数的 96%。因此,综合考虑缺陷描述能力以及测试复杂程度两个因素,从上述两种并发缺陷类型出发,提出一种测试充分性准则。

定义 1(基于缺陷模式的并发测试充分性准则) 给定一个程序,当且仅当测试覆盖该程序所有可以满足定义缺陷类型的访问序列时,测试对此并发程序在定义缺陷类型下是充分的。

图 2 给出了一个含有缺陷的并行程序,借此对符合二元缺陷模式、三元缺陷模式的测试充分性准则进行解释说明。该程序包含两个线程,每个线程分别含有对共享变量 $a.b$ 的

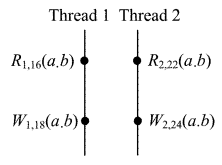
读访问 $R_{1,16}(a.b)$ 和 $R_{2,22}(a.b)$,以及写访问 $W_{1,18}(a.b)$ 和 $W_{2,24}(a.b)$,因此可将程序化简为图 2(b)中的模型。为书写方便,将其分别简写为 R_1,R_2,W_1 及 W_2。由于程序没有对这 4 个访问的同步限制,因此,该程序能够达到符合上述缺陷类型的所有访问序列分别为:

$\{R_1 \prec W_2, W_1 \prec R_2, W_1 \prec W_2, R_2 \prec W_1, W_2 \prec R_1, W_2 \prec W_1, R_1 \prec W_2 \prec W_1, R_2 \prec W_1 \prec W_2\}$

当且仅当测试覆盖到此 8 个访问序列时,满足基于以上定义缺陷模式的测试充分性准则。

```
1. class BankAccount{
2.     int b;
3.     object _lock=new object();
4.     public BankAccount(int money)
5.         { lock(_lock) { b=money;}}
6.     public int GetBalance()
7.         { lock(_lock) { return b;}}
8.     public int WithDraw(int money)
9.         { lock(_lock) { b=b-money;}}
10.    public int Deposit(int money)
11.        { lock(_lock) { b= b+money;}}
12. }
13. BankAccount a=new BankAccount(600);
14. Fun1(){ //Thread 1 (User 1)
15.    int amount=400;
16.    int balance=a.GetBalance(); //R
17.    if (balance >= 400)
18.        a.WithDraw(amount); //W
19. }
20. Fun2(){ //Thread 2(User 2)
21.    int amount=300;
22.    int balance=a.GetBalance(); //R
23.    if (balance >= 300)
24.        a.WithDraw(amount); //W
25. }
```

(a)



(b)

图 2　含有并发缺陷的程序及其执行模型

3　基于缺陷模式测试充分性准则的缺陷定位方法

本节首先概述基于缺陷模式测试充分性准则的定位并发缺陷的基本过程,然后对各步骤进行详细介绍。

该定位过程包括以下 4 个步骤:1)插桩并运行目标程序;2)分析执行结果,预测符合缺陷模式的序列并生成测试方案;3)按测试方案测试目标程序;4)分析可疑度并给出定位结果。

该方法通过预先分析目标程序可能的测试空间大小生成测试方案;然后按照测试方案,通过插入的控制代码逐个调度线程实现目标访问序列,以满足测试准则;最后根据历次程序的执行结果及其覆盖的访问序列信息,分析各个特征序列的可疑程度,返回分析结果。

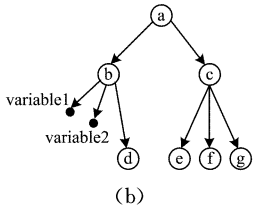
3.1　程序插桩

为获得程序运行过程中内存的访问情况,对.NET的中间语言(MSIL)进行插桩。在不影响程序原行为功能的情况下,首先向桩点前后插入 *Monitor. Enter* 和 *Monitor. Leave* 锁函数,然后在此间插入探针,使记录的内存访问情况与实际执行情况一致,达到记录程序执行路径的目的。另一方面,程序执行时,系统给出的线程和变量标识为绝对标识,此标识在每次运行时均不相同,为使调试信息为缺陷定位分析过程所用,需要使程序每次执行时的线程和变量的编号保持一致,因此需要重新对线程和变量进行统一标识,这里利用二者申请的先后关系具有稳定性的特点建立有序树,并据此进行重新标识。

如图 3(a)所示,线程 a 在 Fun1 中开启线程 b 和线程 c;线程 b 开启线程 d,同时依次申请内存 variable1 和 variable2;线程 c 依次开启线程 e,f,g。根据其申请发生的相对关系,建立图 3(b)所示的有序树。其中,父线程早于子线程创建,且具有从属关系;兄线程点间具有序关系,左边线程点早于右边线程创建;变量的创建总从属于某一线程,且在同一线程下,左边变量早于右边变量创建。根据此规则,可以对程序中各线程和变量进行唯一标识。

```
Fun1() //Thread a
{
   new Thread(Fun2); //Thread b
   new Thread(Fun3); //Thread c
}
Fun2()
{
   variable1 = new variable();
   variable2 = new variable();
   new Thread(Fun4); //Thread d
}
Fun3()
{
   new Thread(Fun4); //Thread e
   new Thread(Fun5); //Thread f
   new Thread(Fun6); //Thread g
}
Fun4() { //do something}
Fun5() { //do something}
Fun6() { //do something}
```

(a)



(b)

图 3　线程和内存的有序树

3.2　预测符合缺陷模式的序列

假设一个程序各个线程内部的执行情况基本不变,在文献[12-13]中也有相同的假设。对目标程序运行数次后,可以获得该程序中各个线程执行序列的基本情况。结合定义的缺陷类型,预测该程序中符合缺陷模式的所有可能的访问序列,以做出符合测试充分性准则的测试方案。

目标程序 p 执行完毕后，可以得到一个全部的内存访问序列 $\prec_p$，之后将 $\prec_p$ 按线程编号及变量编号分拣出独立的子序列 $\prec_{ti}(v1)$，$\prec_{ti}(v2)$，…，$\prec_{ti}(vn)$。将任意两个满足关系 $vi=vj$ 且 $ti\neq tj$ 的子序列 $\prec_{ti}(vi)$ 和 $\prec_{tj}(vj)$ 组成分析对，分析其所有可能符合缺陷模式的访问序列。其过程如算法 1 所示。

算法 1 预测符合缺陷模式的访问序列

输入：待分析的一对访问序列 $trace_{ti}(v)$，$trace_{tj}(v)$；二元缺陷模式集合 $Pattern_B$；三元缺陷模式集合 $Pattern_T$；

输出：满足缺陷模式的序列集合 C_f

```
1. FOR i←0 to trace_ti(v). length DO:
2.     IF trace_ti(v)[i] fit P_B∈Pattern_B THEN:
3. P_B'←P_B
4.     FOR j← 0 to trace_ti(v). length DO:
5. IF trace_ti(v)[i]≺trace_ti(v)[j] fit P_B' THEN:
6. C_B. add(trace_ti(v)[i]≺trace_ti(v)[j])
7. END
8.    END
9.     IF i+1≤trace_ti(v)[i]. length THEN:
10. FOREACH pair IN C_B DO:
11.          IF pair≺trace_ti(v)[i+1] fit P_T∈Pattern_T THEN:
12. C_f. add(pair≺trace_ti(v)[i+1]), C_f. add(pair)
13. C_f. remove(pair)
14.          END
15.       END
16.    END
17. C_f. add(C_B), C_B. clear()
18.     END
19. END
20. REPEAT(trace_tj(v),trace_ti(v))
21. RETURN C_f
```

首先，选取某个分析对中的第 1 个序列 $trace_{ti}(v)$ 作为起始序列，并设置一个锚点，该点将作为所预测序列的起始访问点，记为 *f-access*。之后，在另一个序列 $trace_{tj}(v)$ 中依次选取访问点作为预测序列的第二个访问点，记为 *s-access*，若 *f-access*≺*s-access* 符合某一二元缺陷模式，则将此预测序列加入到预测结果中，如算法 1 中第 5－7 行所示。为预测三元模式序列，尝试将与锚点处在同一线程且紧邻锚点之后的访问点 *t-access* 加入到此时所获取的所有二元访问之后，若组成的三元访问序列 *f-access*≺*s-access*≺*t-access* 满足某一三元缺陷模式，则将此序列加入到预测结果中，如算法中第 11－14 行。按此方法依次在 $trace_{ti}(v)$ 中选取锚点，并进行预测。最后，以另一个序列 $trace_{tj}(v)$ 作为起始序列进行下一轮预测。当按此方法分析完所有的分析对后，得到的预测结果即为最终预测的程序满足缺陷模式的所有访问序列。

此过程是对各线程中共享内存的访问序列进行遍历，因此所得结果符合定义缺陷模式下的测试充分性准则。值得注意的是，此过程并未考虑源程序可能存在的同步代码，得到的预测集合中存在一些不合理的访问序列，但这些序列的存在并不影响此集合满足定义的测试充分性准则。

3.3 测试目标程序

在一般情况下，线程的调度由操作系统根据当前系统的运行情况进行分配，可认为是随机分配运行的过程，但在预测集合 C_f 中存在一些在随机分配运行中难以出现的访问序列。一方面，为符合测试充分性准则，本文的测试需要覆盖这些访问序列；另一方面，有工作表明[4]，在程序实际运行过程中，某些访问序列的出现需要数小时甚至上百小时。考虑到测试的时间开销，通过向源程序插入控制代码并根据目标访问序列的需求直接调度相关访问序列出现，以满足测试充分性准则。

在之前研究中，有些工作通过调度一个线程的工作状态来使某访问序列出现，如 CHESS[1]，RaceFuzzer[17]；有些工作通过在某些合适的位置设置适当的延时来提高某个访问序列出现的概率，如 Ctrigger[8]。但这些方法只能一定概率使得目标访问序列出现，并不能保证其可靠地出现。为实现某访问序列的可靠调度，设计带有线程调度的同步函数 *AccessBegin* 和 *AccessEnd*。它们将分别替代 *Monitor. Enter* 及 *Monitor. End*，并插入到相应的位置上。同步函数 *AccessBegin* 和 *AccessEnd* 及调度算法如算法 2 所示。

算法 2 关键节点同步及目标访问序列调度

输入：待判断的变量访问 access；是否有访问正在进行 flag；目标访问序列 feature；下一步访问在 feature 中的位置 index；当前活动的线程数 aliveNum

```
AccessBegin(access)
1. WHILE TRUE:
2.   LOCK(_lock):
3.     IF flag IS TRUE THEN:
4.       CONTINUE
5.     END
6.     flag←TRUE
7.   END
8.   IF CanPass(access) THEN:
9.     BREAK
10.  END
11.  flag←FALSE
12. END
AccessEnd()
13. flag←FALSE
CanPass(access)
14. Dim canPass←TRUE
15. FOR i←index + 1 TO feature. length DO:
16.   IF access IS feature[i] THEN:
17.     IF aliveNum≥2 THEN:
18.       canPass←FALSE
19.     END
20.   END
21. END
22. IF access IS feature[index] THEN:
23.   index←index + 1
24. END
25. RETURN canPass
```

同步函数 *AccessBegin* 和 *AccessEnd* 的作用与 *Monitor. Enter* 和 *Monitor. Leave* 类似，可保证在其间的指令执行是原子性的；但与 *Monitor. Enter* 不同的是，在算法 2 的第 8－10

行中,同步函数 *AccessBegin* 使用 *CanPass* 函数对待执行的内存访问进行检查,若待执行的访问符合要求,则可以退出 *AccessBegin* 函数,进而可以对该内存进行访问;否则,阻止本次请求,随后将不断对该访问进行检查,直至等到合适的时机执行该访问。

给定目标访问序列 *feature*,*CanPass* 函数可以根据下一步期望执行的访问 *feature*[*index*]判断此时是否是待判断访问 *access* 的执行时机。若待判断访问 *access* 不在 *feature* 中,则可以认为该访问的执行不会对实现 *feature* 产生影响,因此允许其执行;若 *access* 存在于目标序列 *feature* 中且是 *feature*[*index*]之后的访问,此时允许其执行,那么 *access* 将先于 *feature*[*index*]执行,因此将无法实现目标序列,所以可以认为目前并不是 *access* 执行的时机,应阻止本次访问。但为了避免程序因调度算法产生阻塞,即调度算法阻止了当前唯一的可活动线程,仅当程序中存在其他可活动的线程时,调度算法才会阻止 *access* 执行,让其等待适当的执行时机,如算法 2 第 17—19 行所示。此外,在算法 2 第 22—24 行,如果 *access* 恰好是期望的访问 *feature*[*index*],则将 *index* 增加,产生新的目标访问。

现在分析使用该算法对目标访序列调度失败的原因。由上述过程可知,当且仅当活动线程数为 1 时,导致调度失败的访问才被算法允许执行。而在并发访问过程中,同步代码会将活动中的线程置为阻塞状态。源程序和算法中均包含同步代码。在算法中,我们仅在 *AccessBegin* 中加入了同步代码,如算法 2 第 2—7 行,但判断代码 *CanPass* 并不在同步代码中,因此在判断时算法可以保证至少两个线程能同时活动,所以活动线程数为 1 并不是该调度算法造成的,而是源程序本身的行为。可以认为,算法无法调度出的访问序列是预测过程中因未考虑同步机制而得到的不合理的访问序列,不在测试充分性准则的考虑范围内。

由调度过程及对它的分析可知,与使用延时函数不同,我们使用的调度算法可以根据调度方案来精准地控制目标程序的执行过程,使合理的目标访问序列能够可靠出现。因此,按符合测试充分性准则的测试方案进行测试,最终覆盖到的内存访问序列将满足该测试充分性准则。同时,当算法无法调度出目标访问序列时,程序不会进入阻塞状态,可以继续完成之后的指令。

3.4　缺陷定位分析

传统的基于程序谱的缺陷定位思想是通过分析语句的覆盖信息与程序运行结果的关联程度,推测出导致程序发生失效的缺陷语句。基于这样的缺陷定位思想,研究者从不同角度分析了影响定位结果的因素,从而提出了不同的定位公式,如 Tarantula[14],Jaccard[15] 等。与文献[11]相同,这里使用 Jaccard 缺陷定位公式,其中 s 表示待评价的访问序列;总测试次数表示对目标程序的总测试数目;测试通过次数(s)表示经过待评价访问序列且该测试可以得到正确结果的次数;测试不通过次数(s)表示经过待评价访问序列且该测试未能正常运行或得到错误的结果的次数。公式如下:

$$可疑度_{Jaccard}(s)=\frac{测试未通过次数(s)}{总测试次数+测试通过次数(s)}$$

当所有测试完成后,按此公式计算所收集到的所有访问序列的可疑程度,并按最终得到的可疑度结果进行降序排列,得到可疑度排位表。因此,在该排位表中,排名越靠前的访问序列越有可能导致程序失效。

4　方法实现

采用 C# 实现该方法,并编写了完整的缺陷定位工具 ConFinder。除去使用的其他工具,ConFinder 共有 4783 行代码,其中包括 4 个主要部分:源程序静态插桩、访问执行序列解析和预测、访问序列调度以及可疑度分析。

使用开源项目 Mono.Ceil 提供的 MSIL 代码注入的接口对目标程序进行静态插桩,插入的探针可以在程序动态运行过程中获取内存的访问信息,并将其转化为标识后的访问信息(第 3.1 节)。之后将此信息实时发回 ConFinder,用于访问序列的分析和预测(第 3.2 节)。ConFinder 得到预测结果后,生成测试方案,并在之后的测试过程中依次将测试方案发送给插桩后的目标程序,使其按照测试方案执行(第 3.3 节)。若程序可以正常执行完毕并且计算结果符合测试预期结果,则该次测试会被标记为通过;否则,它们均会被 ConFinder 设计的异常捕获机制捕获,并且被标记为未通过。最后,当测试方案完成后,ConFinder 结合历次测试的结果和其覆盖信息计算出各个访问序列的可疑程度,生成排位表并反馈给程序员(第 3.4 节)。

5　实验研究

如表 1 所列,使用 8 个常用的含有并发缺陷的实验程序进行实验,其中 5 个来自于实验用例[16],2 个是 Java 1.4.2 版本的库函数[20],还有 1 个为开源项目程序[17]。它们均采用 Java 语言编写。

表 1　实验程序

测试程序		代码行数	平均内存访问次数	失效率/%	缺陷类型
实验程序	Racer	68	7	47.9	顺序违背
	Bank Account	172	266	0.8	原子性违背
	Airline Tickets	95	52	57.4	原子性违背
	Buffer Writer	255	269	1.2	原子性违背
	Lottery	359	154	28.2	原子性违背
Java 库	Link List	460	922	0.2	原子性违背
	String Buffer	1320	1096	25.8	原子性违背
开源程序	Cache4j	3897	1213	2.0	顺序违背

目前还没有公开的基于.NET 平台的实验程序,我们选择以上 8 个程序作为所设计的工具的实验对象,并使用 Java to C# Converter[21] 代码转换工具将其转换成 C# 版本的程序。该工具通过分析 Java 程序的源码,将其按照相同的实现方式用 C# 语言重新编写。因此,转换后的程序仍然保留了程序本身的逻辑特性,可以对本文工作进行验证。本文实验均在转换后的程序上进行。通过对实验程序的分析,得到了各程序的代码规模、动态特性以及导致程序发生失效的缺陷类型,见表 1 第 2—5 列。本文实验均在同一台计算机上进行,其配置如下:CPU 为 Interl Core i3 2.10GHz 2.10GHz 处理器;内存大小为 4GB;操作系统为 64 位 Windows 7。

为评价 ConFinder 的性能，将其与目前类似的工具 Falcon[11]进行比较。该工具同样使用统计的方法进行缺陷定位，但不同的是，其在收集测试信息时，采用的方案是对目标程序随机运行 100 次，并且使用固定大小的窗格来截取并分析符合缺陷模式的访问片段，这里设置的窗格大小以及其他参数均与文献[11]中的设置相同。

(1)实验 1：有效性和效率分析

以上 8 个程序的缺陷定位结果如表 2 所列。分别比较两种方法在所覆盖的访问序列、定位过程所消耗的时间、实际缺陷定位结果排名以及并列第一的访问序列数的区别。由第 2 列和第 6 列可知，Falcon 使用的随机分配测试方法很难满足测试充分性准则，即使在设置的每个程序运行 100 次的条件下，测试中仍有无法覆盖到的访问序列。表 2 第 3 列和第 7 列分别显示两种方法完成整个测试和定位过程所用的时间(其中 Confinder 包含预测分析所需时间)，相比之下，ConFinder 制定出了有效的测试计划并且可以高效地按照测试计划覆盖程序中所有符合缺陷模式的访问序列。在 ConFinder 定位方法中，所需要的测试数目与预测的访问序列数以及各次程序覆盖到的访问序列有关，若要使得预测到的访问序列数目更多，则需要更多的测试数目。然而一些预测到的访问序列并不合理，因此在实际测试中并没有出现，但更多的测试可以提供更多的覆盖信息，可以被缺陷定位过程所利用；Falcon 试图通过运行大量的测试来达到尽可能多地覆盖可疑序列的目的，然而，由于缺少对测试计划的安排，在测试中累计覆盖的各个访问序列次数并不均匀[8]，一些访问序列总会被测试所覆盖，而另一些被执行的次数则很少，这导致在之后进行可疑度计算时会出现一定的不平衡倾向，从而影响最终的定位结果。

表 2 缺陷定位结果

	测试程序	ConFinder				Falcon			
		覆盖的访问序列数	平均耗费时间(s)/执行次数	实际缺陷访问序列排名	并列排名第 1 访问序列数	覆盖的访问序列数	平均耗费时间(s)/执行次数	实际缺陷访问序列排名	并列排名第 1 访问序列数
#1	Racer	15	1.5/12	1	1	12	10.3/100	1	3
#2	Bank Account	73	9.3/59	1	1	71	12.7/100	1	2
#3	Airline Tickets	14	11.2/80	2	1	10	11.5/100	2	1
#4	Buffer Writer	96	42.7/227	1	1	91	14.9/100	3	1
#5	Lottery	21	2.4/20	1	1	21	11.6/100	1	1
#6	Link List	26	146.1/967	1	1	22	11.2/100	4	2
#7	String Buffer	30	3.4/23	1	2	22	11.6/100	1	2
#8	Cache4j	17	19.2/15	1	2	16	317.5/100	1	2

表 2 第 4 列和第 8 列分别列出了两种方法对实际缺陷的排名情况，ConFinder 方法中只有 Airline Tickets 程序将实际缺陷排在第二的位置，而 Falcon 中则有 3 个程序未能将实际缺陷排在第一的位置。另一方面，表 2 第 5 列和第 9 列分别记录了定位结果中排在第一位的缺陷数目，该数据表明定位方法对各个谓词的区分程度。ConFinder 和 Falcon 使用了相同的定位公式，而 ConFinder 可以更好地区分各个谓词，可以认为，ConFinder 使用了更加充分的覆盖信息。根据上述实验，ConFinder 可以高效地完成并发程序的缺陷定位任务，同时，其结果也能很好地帮助程序员找到程序错误。

(2)实验 2：稳定性分析

为评价两个方法定位结果的稳定性，分别用两种方法对测试程序各重复进行 30 次实验，并对所得结果进行稳定性分析。

为分析结果的稳定性，计算各个程序在 30 次实验中的排位表的波动情况。设排位表中的某谓词为 s_i，其在第 j 次实验后所处排位表的位置为 $r_j(s_i)$，根据以下公式计算排位表的波动情况：

$$\sigma=\frac{1}{n}\sum_{i=1}^{n}\sqrt{\frac{1}{N}\sum_{j=1}^{N}(r_j(s_i)-\overline{r(s_i)})^2}$$

其中，σ 表示排位表的波动值，n 表示程序中谓词的个数，N 表示实验次数。σ 越小，表示定位结果越稳定。另外，规定排位表的总长度不超过 20，若超过，则按照 20 位计算。这是因为在实际定位过程中程序员只能按排位表检查一定数目的谓词，若超过此数目的检查量，该排位表将无法起到辅助调试的作用。

实验结果如图 4 所示。可以看到，除 Racer 程序和Lottery 程序外，ConFinder 的定位结果的波动值均小于 Falcon。这是因为，新信息的引入会影响方法对谓词与程序失效的关联程度的判断，Falcon 由于随机决定程序的执行情况，在不同实验中会使用不同的信息，因此定位结果不稳定。而 ConFinder 会在测试满足充分性准则时停止实验，在一定程度上保证了使用信息的稳定性。ConFinder 在 Racer 程序和 Lottery 程序中的稳定性弱于 Falcon。对于 Racer 程序，Falcon 给出的定位结果的区分度不足，因此在排名上容易趋于稳定；对于 Lottery 程序，ConFinder 和 Falcon 均满足了测试充分性准则，但 Falcon 对该程序运行的次数更多，使用了更多的覆盖信息建立谓词与程序失效的联系，因此其结果更加稳定。

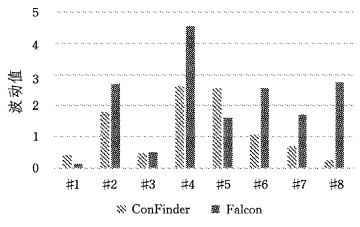


图 4 缺陷定位结果的稳定性比较

综上所述，基于测试充分性准则的缺陷定位方法可以有效地找到导致非死锁并发程序失效的原因，其结果也具有较好的稳定性。

结束语 本文针对非死锁并发缺陷，通过研究分析并发缺陷的类型，提出了一种基于缺陷模式的测试充分性准则，并

设计了一套基于该准则的缺陷定位方法。该方法通过采集程序信息来预测可以满足该准则的条件并制定了相应的调度方案，再利用收集到的程序覆盖信息对导致程序失效的谓词进行定位。实验数据证明了该方法的有效性和定位结果的可靠性。

一方面，由于程序的测试空间巨大，如何优化测试方案以更快地满足测试充分准则是今后工作的研究方向之一。另一方面，本文仅考虑了顺序违背和原子违背中两种最基本的表现形式，即二元模式和三元模式，这两种模式可以有效地分析双线程单变量缺陷，但缺乏对多线程多变量缺陷分析的能力。通过考虑更多复杂的缺陷模式来分析该方法的有效性和性能是下一步的研究计划。

参 考 文 献

[1] MUSUVATHI M, QADEER S. Iterative context bounding for systematic testing of multithreaded programs[C]// ACM Sigplan Notices. New York, USA, 2007: 446-455.

[2] YIN Z, YUAN D, ZHOU Y, et al. How do fixes become bugs? [C]// Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. Sunnyvale, USA, 2011: 26-36.

[3] GODEFROID P, NAGAPPAN N. Concurrency at Microsoft: An exploratory survey[J/OL]. http://patricegodefroid. github. io/public_psfiles/ec2. pdf.

[4] LU S, PARK S, SEO E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics[C]// ACM Sigplan Notices. New York, USA, 2008: 329-339.

[5] FLANAGAN C, FREUND S N. Type-based race detection for Java[C]// ACM SIGPLAN Notices. New York, USA, 2000: 219-232.

[6] PRATIKAKIS P, FOSTER J S, HICKS M. LOCKSMITH: context-sensitive correlation analysis for race detection[C]// ACM SIGPLAN Notices. New York, USA, 2006: 320-331.

[7] SAVAGE S, BURROWS M, NELSON G, et al. Eraser: A dynamic data race detector for multithreaded programs[J]. ACM Transactions on Computer Systems, 1997, 15(4): 391-411.

[8] PARK S, LU S, ZHOU Y. CTrigger: exposing atomicity violation bugs from their hiding places[C]// ACM SIGARCH Computer Architecture News. New York, USA, 2009: 25-36.

[9] LU S, JIANG W, ZHOU Y. A study of interleaving coverage criteria[C]// The 6th Joint Meeting on European Software Engineering Conference and The ACM SIGSOFT Symposium on The Foundations of Software Engineering: Companion Papers. New York, USA, 2007: 533-536.

[10] ARULRAJ J, CHANG P C, JIN G, et al. Production-run software failure diagnosis via hardware performance counters[C]// ACM SIGARCH Computer Architecture News. New York, USA, 2013: 101-112.

[11] PARK S, VUDUC R W, HARROLD M J. Falcon: fault localization in concurrent programs[C]// Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York, USA, 2010: 245-254.

[12] ZHENG M, EDENHOFNER J G, LUO Z, et al. CIVL: applying a general concurrency verification framework to C/Pthreads programs (competition contribution)[C]// International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, GER, 2016: 908-911.

[13] ZHANG T, LEE D, JUNG C. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory[C]// Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. New York, USA, 2016: 159-173.

[14] JONES J A, HARROLD M J. Empirical evaluation of the tarantula automatic fault-localization technique[C]// Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. New York, USA, 2005: 273-282.

[15] ABREU R, ZOETEWEIJ P, VAN GEMUND A J C. On the accuracy of spectrum-based fault localization[C]// Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION. Windsor, UK, 2007: 89-98.

[16] EYTANI Y, HAVELUND K, STOLLER S D, et al. Towards a framework and a benchmark for testing tools for multi-threaded programs[J]. Concurrency and Computation: Practice and Experience, 2007, 19(3): 267-279.

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

[18] WU Z, LU K, WANG X, et al. Collaborative technique for concurrency bug detection[J]. International Journal of Parallel Programming, 2015, 43(2): 260-285.

[19] PARK S, VUDUC R, HARROLD M J. UNICORN: a unified approach for localizing non-deadlock concurrency bugs[J]. Software Testing, Verification and Reliability, 2015, 25(3): 167-190.

[20] PARK C S, SEN K. Randomized active atomicity violation detection in concurrent programs[C]// Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York, USA, 2008: 135-145.

[21] http://www. tangiblesoftwaresolutions. com/Product_Details/Java_to_CSharp_Converter. html

[22] CAI Y, CAO L. Fixing deadlocks via lock pre-acquisitions[C]// Proceedings of the 38th International Conference on Software Engineering. New York, USA, 2016: 1109-1120.

[23] YU Y, RODEHEFFER T, CHEN W. Racetrack: efficient detection of data race conditions via adaptive tracking[C]// ACM SIGOPS Operating Systems Review. New York, USA, 2005: 221-234.