软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(5):1107−1117 [doi: 10.13328/j.cnki.jos.005210]

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

面向收敛的并发程序执行轨迹静态简化方法*

常 曦[1], 薛建新[1,2], 张 卓[2], 毛晓光[2]

[1](上海第二工业大学 软件工程系,上海 201209)

[2](国防科学技术大学 计算机学院,湖南 长沙 410073)

通讯作者: 薛建新, E-mail: jxxue@sspu.edu.cn

摘 要: 轨迹静态简化技术是在确保与原轨迹等价的前提下,通过随机减少程序执行时线程切换的数量,达到提高程序员调试并发程序效率的目的.然而,轨迹中可减少的线程切换分布往往是不均匀的,因此,随机简化策略难以有效地发现可简化的线程切换.为此,提出了面向收敛的合并算法致力于这个问题.该算法的基本思想是:不断地随机选择一线程执行区间作为中心,在同一线程内,采用面向收敛的合并算法迭代地寻找可与其合并的前置执行区间和后置执行区间.实验结果表明,该方法可以高品质地减少执行轨迹中的线程切换数量,进而有助于程序员快速发现引发错误的线程交错.

关键词: 并发程序;执行轨迹;轨迹等价;轨迹简化;调试

中图法分类号: TP311

中文引用格式: 常曦,薛建新,张卓,毛晓光.面向收敛的并发程序执行轨迹静态简化方法.软件学报,2017,28(5):1107−1117. http://www.jos.org.cn/1000-9825/5210.htm

英文引用格式: Chang X, Xue JX, Zhang Z, Mao XG. Convergence-Oriented static approach for simplifying execution traces of concurrent programs. Ruan Jian Xue Bao/Journal of Software, 2017,28(5):1107−1117 (in Chinese). http://www.jos.org.cn/1000-9825/5210.htm

Convergence-Oriented Static Approach for Simplifying Execution Traces of Concurrent Programs

CHANG Xi[1], XUE Jian-Xin[1,2], ZHANG Zhuo[2], MAO Xiao-Guang[2]

[1](Department of Software Engineering, Shanghai Polytechnic University, Shanghai 201209, China)

[2](College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: Static trace simplification technique is a method to improve the efficiency of debugging through reducing the number of thread contexts in a buggy execution trace without changing the dependency information. However, the application of static trace simplification technique to simply a buggy trace still faces a strong challenge in that the distribution of reduced thread contexts is usually non-uniform, and therefore a random trace simplification technique is difficult to achieve. In this paper, a convergence-oriented static trace simplification approach is proposed to address this challenge. The essential idea of this approach is to build a static trace simplification model, and then repeatedly adopt a convergence-oriented merging algorithm to search the mergeable previous and succeeding intervals in the same thread. Experimental results show that this new approach can simply the traces with high quality. It can help programers find the thread interleaves that cause the faults.

Key words: concurrency program; execution trace; trace equivalence; trace simplification; debugging

* 基金项目: 国家自然科学基金(61502296, 61379054, 61672529); 上海市自然科学基金(15ZR1417000)

Foundation item: National Natural Science Foundation of China (61502296, 61379054, 61672529); Natural Science Foundation of Shanghai (15ZR1417000)

收稿时间: 2016-07-15; 修改时间: 2016-09-25; 采用时间: 2016-12-07; jos 在线出版时间: 2017-01-20

CNKI 网络优先出版: 2017-01-20 16:06:32, http://www.cnki.net/kcms/detail/11.2560.TP.20170120.1606.004.html

由于并发程序执行时不确定的线程切换,即使错误执行轨迹被记录,程序员往往也难以高效地推导引发错误的线程交错.致力于提高程序员推导错误原因的效率,现有研究主要涉及两个方面的工作.第一,加快错误重现的速度.现有方法[1−8]通过多种方式从轨迹中抽取错误相关事件,以实现快速地错误重现.第二,减少引发错误的轨迹中包含的线程切换数目.轨迹中包含大量的细粒度线程切换,这些线程切换会干扰程序员推导错误时常规的线性化思维,明显地增加程序员发现错误原因所需的时间.

针对快速重现问题的研究取得了瞩目的进展,但较少的研究致力于第 2 方面的问题.为减少引发错误的轨迹中包含的线程切换数目,Jalbert 等人[9]证明了获得线程切换数量最少的轨迹是 NP 难问题,并最先提出启发式动态轨迹简化方法,启发式地改变轨迹中事件的位置来减少线程切换.然而对于每次线程切换,至少需要重新执行程序 1 次以确认改变后的轨迹是否满足与原轨迹相同的属性.这种动态确认将会 5x~100x 地慢于原始执行速度.因此,动态轨迹简化技术难以高效简化轨迹.为提高轨迹简化的效率,Huang 等人[10]提出了执行轨迹静态简化方法.该方法基于一个严格的依赖图,采用随机合并同一线程中相邻事件来简化轨迹.这种静态方法可在保证简化轨迹与原轨迹等价的前提下,对执行轨迹进行简化.但轨迹中可减少的线程切换分布往往是不均匀的,每一次简化计算都依赖于随机选择的细粒度同线程相邻事件,不利于有效地发现可减少的线程切换.

致力于提高简化轨迹的有效性,本文提出一种面向收敛的静态简化轨迹方法(简称 COSIM).该方法采用两个方面的优化策略.第一,用粗粒度的线程执行区间替换细粒度的事件作为合并单元.线程执行区间是指线程中的一次最大连续执行事件序列.既然原轨迹是可行的,那么其中的连续执行事件肯定是符合合并条件,无需对线程执行区间内的事件再次进行检查.第二,观察发现,当一对相邻本地执行区间符合合并条件时,其邻近的周边对象往往也可与该对合并区间满足合并条件.因此,一旦一对相邻线程区间被确定可合并,沿着线程分支继续查找邻近区间就有利于提高成功合并的概率.

基于以上优化策略,COSIM 将轨迹简化过程分为 3 个阶段.

1) 通过扫描已记录的引发错误的执行轨迹,构建一切换标识的依赖树.该依赖树不仅刻画了细粒度的事件之间依赖关系,还收集粗粒度的线程执行区间,并双向刻画线程切换,以便于实现同线程内前置区间和后置区间向中心区间合并.
2) 基于已构建的依赖树,不断地随机选择一个线程执行区间作为中心,采用面向收敛的合并算法迭代地检查和合并其当前本地前置区间和本地后置区间.
3) 根据合并后的树重组轨迹,以实现轨迹的简化.实验结果表明,该方法可以有效地减少轨迹中的线程切换数量,进而帮助程序员快速推导引发错误的线程交错.

本文贡献体现在以下 3 个方面.

1) 基于并发执行轨迹静态简化技术,提出一种面向收敛的合并算法,基于随机选择的线程执行区间,迭代地将符合合并条件的本地前置区间和本地后置线程合并至该中心区间,以实现有效的轨迹简化.
2) 提出一种切换标识的依赖树来抽象轨迹.该树利用原轨迹中连续执行事件信息,避免细粒度的合并条件检查;双向编码线程切换,有利于本地前置区间和本地后置区间向目标中心区间的合并;
3) 实现方法对应的工具 COSIM.实验表明,COSIM 方法可以有效和高效地对轨迹进行简化.

1 理论基础

1.1 轨 迹

本文采用文献[11]中对执行轨迹的定义,即并发程序执行轨迹被抽象成一个事件序列 $E=\langle e_1,\ldots,e_i,\ldots,e_n\rangle$,其中,$e_i$ 为以下 5 类事件之一.

- $MEM(\sigma,sv,a,t)$:表示线程 t 访问全局变量 sv,其中,σ表示访问事件的产生语句位置,$a\in\{Read,Write\}$以区分访问的类型.
- $ACQ(l,t)$:表示线程 t 获取一个锁 l.
- $REL(l,t)$:表示线程 t 释放一个锁 l.

- *SND*(*g*,*t*):表示线程 *t* 发送一个信号 *g*.
- *RCV*(*g*,*t*):表示线程 *t* 接收一个信号 *g*.

1.2 依赖关系

与文献[10]一致,本文采用的依赖关系不仅包括本地线程中的依赖关系,而且包括线程之间的依赖关系.即事件 e_i 和事件 e_j 之间存在一个依赖关系 $e_i \rightarrow e_j$,当满足下列情况之一时.

- 本地依赖关系:e_i 和 e_j 是相同线程中相邻访问事件($i<j$).
- 远程依赖关系:
  - e_i 是发送信号 g 事件,e_j 是接收信号 g 事件;
  - e_j 是获取锁 l 事件且锁 l 刚被事件 e_i 释放;
  - e_i 和 e_j 是不同线程中关于同一变量的连续访问事件($i<j$),且 e_i 和 e_j 至少有 1 个为写事件.

1.3 静态简化轨迹方法

静态简化轨迹方法是一种静态减少执行轨迹中线程切换的方法[10],该方法采用基于重新调度的轨迹等价理论,即给定一条轨迹 *tr*,在不改变依赖关系(见第 1.2 节)的前提下,重新调度该轨迹 *tr* 中的事件与原轨迹等价.该方法实现轨迹简化分为 3 步:首先,对原轨迹建立一张依赖图以刻画事件之间的依赖关系;其次,基于建立的依赖图,根据每条本地依赖边(n_i,n_j)随机生成一个事件序列,并检测除了经过本地依赖边以外,节点 n_i 沿着该评估序列可否达到 n_j;当不可达时,则合并这条本地依赖边相连的节点.虽然现有的静态轨迹简化技术在依赖关系不变的情况下,通过随机求得序列来判断是否可减少线程切换,但简化的效果依赖于随机选择的评估序列,缺少一种快速而安全的策略以有效地实现轨迹简化.我们在下文中将致力于对这个问题进行说明.

2 示 例

本节使用图 1 的一个示例来说明 COSIM 方法的意图.图 1 中执行轨迹包含 4 个线程、12 次线程切换,分别发生在:线程 t_0 执行事件 4 之后和线程 t_1 执行事件 5 之前,线程 t_1 执行事件 6 之后和线程 t_2 执行事件 7 之前,线程 t_2 执行事件 8 之后和线程 t_3 执行事件 12 之前,线程 t_3 执行事件 13 之后和线程 t_2 执行事件 14 之前,线程 t_2 执行事件 15 之后和线程 t_3 执行事件 16 之前,线程 t_3 执行事件 17 之后和线程 t_1 执行事件 18 之前,线程 t_1 执行事件 19 之后和线程 t_0 执行事件 20 之前.

1 *RCV*(g_1,t_0)	5 *RCV*(g_2,t_1)	6 *RCV*(g_3,t_2)	12 *RCV*(g_4,t_3)
2 *SND*(g_2,t_0)			13 *MEM*(σ_5,*x*,*READ*,t_3)
3 *SND*(g_3,t_0)	7 *MEM*(σ_1,*x*,*READ*,t_1)	8 *MEM*(σ_3,*x*,*READ*,t_2)	
4 *SND*(g_4,t_0)			
	9 *ACQ*(*l*,t_1)	14 *MEM*(σ_4,*y*,*READ*,t_2)	16 *MEM*(σ_6,*y*,*READ*,t_3)
	10 *MEM*(σ_2,*y*,*WRITE*,t_1)	15 *SND*(g_6,t_2)	
20 *RCV*(g_5,t_0)	11 *REL*(*l*,t_1)		18 *SND*(g_7,t_3)
21 *RCV*(g_6,t_0)			
22 *RCV*(g_7,t_0)	17 *MEM*(σ_7,*x*,*READ*,t_1)		
23 *SND*(g_8,t_0)			
	19 *SND*(g_5,t_1)		

Fig.1 A sample trace

图 1 示例轨迹

我们利用文献[12]中的术语线程执行区间(缩写为 TEI)来定义线程连续执行的最大事件序列.假设随机选择依线程执行区间{9,10,11},COSIM 采用一组对称策略进行检测和合并:首先检测线程执行区间的本地前驱区间{7}能否合并至被选择区间之前,检测符合合并条件,将区间{7}移至区间{9,10,11}之前;然后检测线程执行区间的本地后继区间{17},检测符合合并条件,将区间{17}移至区间{9,10,11}之后;于是,COSIM 按这种方式进行合并,从而切换数量减少至 8 次(考虑到事件 6 和事件 8、事件 16 和事件 18 随之合并).涉及合并后的轨迹如图 2 所示(括弧内为简化后轨迹中对应事件的发生顺序).

1 $RCV(g_1,t_0)$	5 $RCV(g_2,t_1)$	6 $RCV(g_3,t_2)$	12(13) $RCV(g_4,t_3)$
2 $SND(g_2,t_0)$		8(7) $MEM(\sigma_3,x,READ,t_2)$	13(14) $MEM(\sigma_5,x,READ,t_3)$
3 $SND(g_3,t_0)$	7(8) $MEM(\sigma_1,x,READ,t_1)$		
4 $SND(g_4,t_0)$	9 $ACQ(l,t_1)$		
	10 $MEM(\sigma_2,y,WRITE,t_1)$	14(15) $MEM(\sigma_4,y,READ,t_2)$	16(17) $MEM(\sigma_6,y,READ,t_3)$
20 $RCV(g_5,t_0)$	11 $REL(l,t_1)$	15(16) $SND(g_6,t_2)$	18 $SND(g_7,t_3)$
21 $RCV(g_6,t_0)$	17(12) $MEM(\sigma_7,x,READ,t_1)$		
22 $RCV(g_7,t_0)$			
23 $SND(g_8,t_0)$	19 $SND(g_5,t_1)$		

Fig.2 Simplified trace after implementing the mergence one time

图 2 第 1 次合并之后的轨迹片段

此外,COSIM 第 2 次检测线程执行区间{5},{19}是否可合并至刚合并后的线程执行区间{7,9,10,11,17}之前和之后,从而切换次数可减少为 6 次.涉及合并后的轨迹如图 3 所示.

1 $RCV(g_1,t_0)$		6(5) $RCV(g_3,t_2)$	12(14) $RCV(g_4,t_3)$
2 $SND(g_2,t_0)$	5(7) $RCV(g_2,t_1)$	8(6) $MEM(\sigma_3,x,READ,t_2)$	13(15) $MEM(\sigma_5,x,READ,t_3)$
3 $SND(g_3,t_0)$	7(8) $MEM(\sigma_1,x,READ,t_1)$		
4 $SND(g_4,t_0)$	9 $ACQ(l,t_1)$		
	10 $MEM(\sigma_2,y,WRITE,t_1)$	14(16) $MEM(\sigma_4,y,READ,t_2)$	16(18) $MEM(\sigma_6,y,READ,t_3)$
20 $RCV(g_5,t_0)$	11 $REL(l,t_1)$	15(17) $SND(g_6,t_2)$	18(19) $SND(g_7,t_3)$
21 $RCV(g_6,t_0)$	17(12) $MEM(\sigma_7,x,READ,t_1)$		
22 $RCV(g_7,t_0)$	19(13) $SND(g_5,t_1)$		
23 $SND(g_8,t_0)$			

Fig.3 Simplified trace after implementing the mergence two times

图 3 第 2 次合并之后的轨迹片段

COSIM 采用一组对称的合并策略,当选择任一线程执行区间之后,能够将其本地前驱和后继执行区间进行检测,这可以增加每次合并的范围.此外,观察发现,一旦发现本地相邻的事件区间可实现合并,往往其一定范围内的邻近区间也能与之合并.因此,采用面向收敛的方法往往能够更有效地找到可合并的连续执行区间.

3 方 法

本文提出的简化技术包含 3 个阶段(如图 4 所示):构建切换标识的依赖树、检测和合并线程执行区间和重组轨迹.

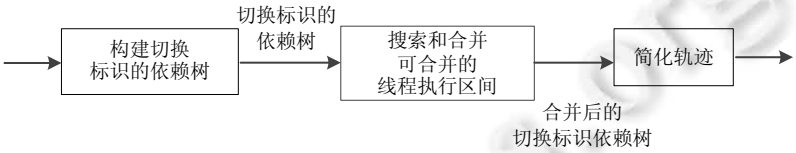


Fig.4 Overview of COSIM

图 4 COSIM 流程图

3.1 构建切换标识的依赖树

为了在保留依赖关系的前提下实现有效简化并发程序执行轨迹中的线程切换,我们设计了一个切换标识的依赖树.该树不仅刻画细粒度的事件之间依赖关系,而且具有两个方面的特点:(1) 轨迹按照线程被分割成若干分支,每个线程被分割成若干线程执行区间;(2) 双向标识线程切换点.由于依赖树收集原轨迹中的最大连续执行事件序列,可避免重复检测细粒度的本地相邻事件之间是否满足合并条件.此外,采用双向编码线程切换便于下文中实现面向收敛的合并算法(见第 3.2 节).

定义(切换标识的依赖树). 给定一条轨迹 tr,切换标识的依赖树 $SDT(Root,SDT(t_1),SDT(t_2),...,SDT(t_n))$包含一

个根节点和一系列分支,每个分支 $SDT(t)$对应一个线程 t.每一个分支由若干线程执行区间组成$(tei_1,...,tei_m)$,每个线程执行区间表示一个最大的属于同一线程的连续节点集序列,且被定义为一个元组(*TID*,*Nodes*,*PRE*,*NEXT*),其中,

- *TID* 为线程执行区间标识集合.我们用 $Tei(I,T)$来标识线程执行区间,例如,$Tei(i,t)$表示属于线程 t 的第 i 个线程执行区间.
- *Nodes* 表示一组节点集合,每个节点(*K*@*T*,*NT*,*RP*,*LP*)代表一个事件,其中,
  - *K* 为事件全局标识集合;
  - *T* 为线程标识;
  - *NT* 为一个节点类型集合,一个节点类型可为{*WRITE*,*READ*,*ACQ*,*REL*,*SND*,*RCV*}之一;
  - *RP* 为远程前驱集合;
  - *LP* 为本地前驱集合.
- *PRE* 为前置线程执行区间集合,任一 $pre\in PRE$,指向该区间的前一个线程执行区间.
- *NEXT* 为后置线程执行区间集合,任一 $next\in NEXT$,指向该区间的后一个线程执行区间.

图 5 所示为图 1 原轨迹对应的切换标识的依赖树.

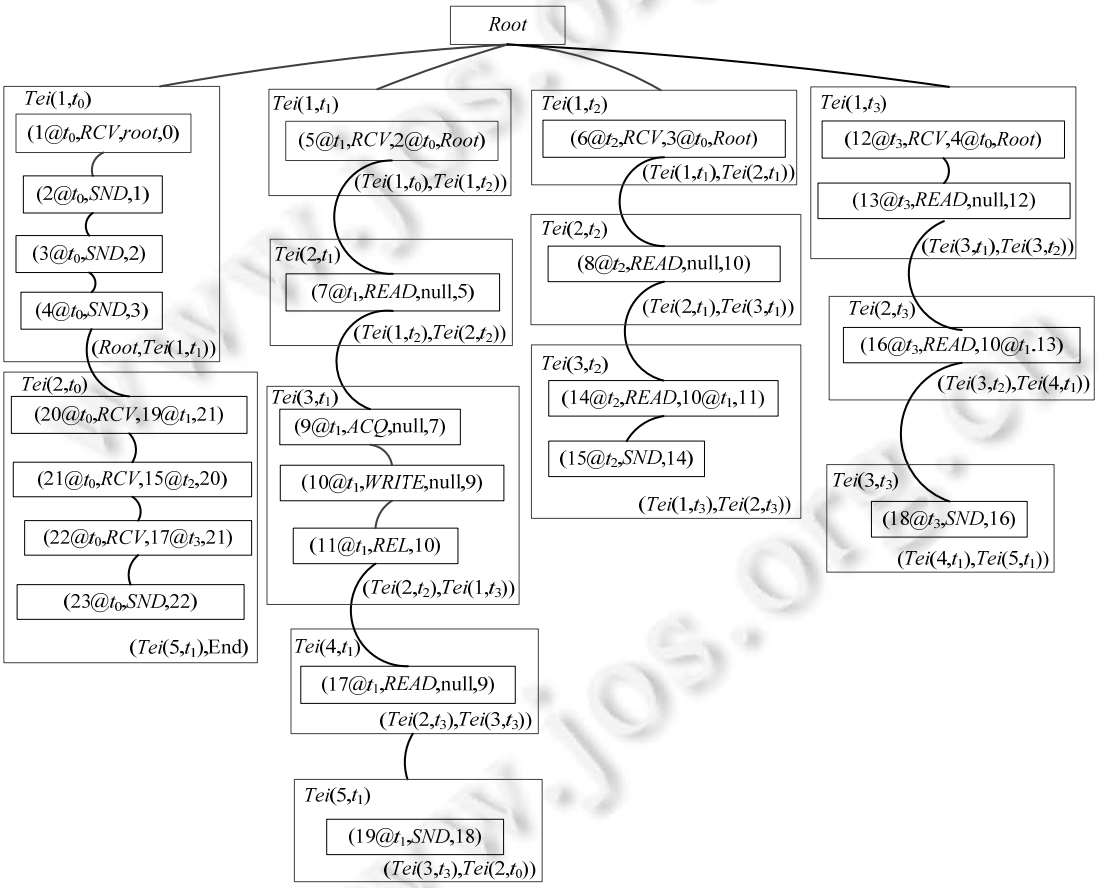


Fig.5 Switch-Identified dependency tree

图 5 切换标识的依赖树

轨迹中包含的 4 个线程被刻画成 4 条分支.分支 1 包含线程执行区间 $Tei(1,t_0)$和 $Tei(2,t_0)$.分支 2 包含线程

执行区间 $Tei(1,t_1)$,$Tei(2,t_1)$,$Tei(3,t_1)$和 $Tei(4,t_1)$.分支 3 包含线程执行区间 $Tei(1,t_2)$,$Tei(2,t_2)$和 $Tei(3,t_2)$.分支 4 包含线程执行区间 $Tei(1,t_3)$,$Tei(2,t_3)$和 $Tei(3,t_3)$.其中,区间 $Tei(1,t_1)$中的节点 $5@t_1$、区间 $Tei(1,t_2)$中的节点 $6@t_2$、区间 $Tei(1,t_3)$中的节点 $12@t_3$ 的远程前驱分别为 $Tei(1,t_0)$中的节点 $2@t_0$,$3@t_0$ 和 $4@t_0$;区间 $Tei(3,t_2)$中的节点 $14@t_2$ 和区间 $Tei(2,t_3)$中的节点 $16@t_2$ 的远程前驱节点都为区间 $Tei(3,t_1)$中的节点 $10@t_1$;区间 $Tei(2,t_0)$中的节点 $20@t_0$,$21@t_0$ 和 $22@t_0$ 的远程前驱节点分别为区间 $Tei(5,t_1)$中的节点 $19@t_1$、区间 $Tei(3,t_2)$中的节点 $15@t_2$ 和区间 $Tei(3,t_3)$中的节点 $18@t_3$.这刻画节点与其远程前驱节点之间存在远程依赖关系.

构建切换标识的依赖树算法如算法 1 所示.具体来说,算法首先输入一条执行轨迹 *tr*,初始化待构建的依赖树 *SDT* 和当前线程执行区间 tei_{cur} 都为根节点.在构建过程中,算法通过线性扫描轨迹 *tr* 收集 3 个方面的信息:线程执行区间、线程切换点和相互依赖的事件.算法第 5 行~第 10 行用于标识线程区间和双向标识线程切换点.如果当前分析事件与 tei_{cur} 不属于同一线程,则说明该执行点发生了一个线程切换.因此新建一执行区间 tei_{new},将 tei_{cur} 的 *next* 域指向 tei_{new},tei_{new} 的 *pre* 域指向 tei_{cur},并更新 tei_{cur} 为 tei_{new}.算法第 11 行~第 31 行用于添加当前事件至当前执行区间中,并依据轨迹中的各种事件类型找到其依赖的事件.

算法 1. Algorithm for model construction.

Input: A recorded trace *tr*.

Output: *SDT*.

ModelConstruction(*tr*)

1. **begin**
2. $GDT \leftarrow root$;
3. $tei_{cur} \leftarrow root$;
4. **For** k=1 to $|tr|$ **do**
5. **If** $T(e_i)$!=$tei_{cur}.t$ **then**
6. new an interval tei_{new} in $SDT(T(e_i))$
7. $tei_{cur}.next \leftarrow tei_{new}.id$
8. $tei_{new}.pre \leftarrow tei_{cur}.id$
9. $tei_{cur} \leftarrow tei_{new}$
10. **end if**
11. **Switch** e_i **do**
12. **case**: $ACQ(l_i,t_i)$
13. add an *ACQ* node $i@t_i$ to $SDT(t_i).tei_{cur}$
14. $n_{i@t_i}.rp \leftarrow$ get the *REL* node releasing the lock l_i
15. **case**: $REL(l_i,t_i)$
16. add a *REL* node $i@t_i$ to $SDT(t_i).tei_{cur}$
17. refer the node $i@t_i$ as the node releasing the lock l_i
18. **case**: $SND(g_i,t_i)$
19. add a *SND* node $i@t_i$ to $SDT(t_i).tei_{cur}$
20. refer the node $i@t_i$ as the node sending the message g_i
21. **case**: $RCV(g_i,t_i)$
22. add a *RCV* node $i@t_i$ to $SDT(t_i).tei_{cur}$
23. $n_{i@t_i}.rp \leftarrow$ get the *SND* node sending the message g_i
24. **case**: $MEM(\sigma_i,WRITE,sv_i,t_i)$
25. add a *WRITE* node $i@t_i$ to $SDT(t_i).tei_{cur}$
26. refer the node $i@t_i$ as the last node writing the variable sv_i

27. $n_{i@t_i}.rp \leftarrow$ get the last *MEM* node accessing the variable sv_i
28. **case**: $MEM(\sigma_i,READ,sv_i,t_i)$
29. add a *READ* node $i@t_i$ to $SDT(t_i).tei_{cur}$
30. $n_{i@t_i}.rp \leftarrow$ get the last *WRITE* node writing the variable sv_i
31. **end for**
32. **end**

切换标识的依赖树有利于抽象对应的执行轨迹.一条执行轨迹可被定义成一条从根节点出发到最后一个线程执行区间的双向序列.

定义(切换轨迹). 一条切换轨迹被定义为一个有限的线程执行区间双向序列.

$$Root \underset{Root}{\overset{Tei(i_1,t_{i_1})}{\longrightarrow}} Tei(i_1,t_{i_1}) \underset{Tei(i_1,t_{i_1})}{\overset{Tei(i_2,t_{i_2})}{\longrightarrow}} \cdots \underset{Tei(i_{n-1},t_{i_{n-1}})}{\overset{Tei(i_n,t_{i_n})}{\longrightarrow}} Tei(i_n,t_{i_n}),$$

其中,$Tei(i_j,t_{i_j}) \underset{Tei(i_j,t_{i_j})}{\overset{Tei(i_{j+1},t_{i_{j+1}})}{\longrightarrow}} Tei(i_{j+1},t_{i_{j+1}})$ 表示一个从线程执行区间 $Tei(i_j,t_{i_j})$ 到 $Tei(i_{j+1},t_{i_{j+1}})$ 的线程切换.

构建切换标识的依赖树的算法空间复杂度是 $O(n\times m^2)$,n 为轨迹中的事件数目,m 为远程依赖事件数目.通常,远程依赖事件数目都远小于轨迹数目,因此该依赖树所需空间不会远大于轨迹空间.

3.2 面向收敛的搜索算法

要减少轨迹中的上下文切换,这需要尽量扩展每个线程中的连续执行区间.COSIM 基于已构建的依赖树,重复地随机选择一个线程执行区间作为中心区间,启发式地对称检查该中心区间的本地前驱区间是否可以合并至其之前以及该中心区间的本地前驱区间是否可以合并至其之后.为便于说明算法,先约定如下符号标识.

- SDT_m 表示当前已合并的 *SDT*;
- *TEISet* 表示待计算的线程执行区间集合;
- M_l/M_r 表示当前待合并的本地前驱/本地后继 *TEI* 集合.

面向收敛算法包含以下 6 个基本操作.

- *lp-can-merge*(*tei'*,*tei*):用于检查区间 *tei'* 是否能够合并到区间 *tei* 之前,其中,*tei'* 为 *tei* 的本地前驱区间.假设 *RTei* 为轨迹⟨*tei'*,...,*tei*⟩中不属于线程 *tei.t* 的区间集合,该操作检查 *RTei* 中每个节点的远程前驱节点是否在 *tei* 中.只有所有节点的远程前驱节点都不在中时返回“真”,否则返回“假”.
- *ls-can-merge*(*tei*,*tei'*):用于检查区间 *tei'* 是否能够合并到区间 *tei* 之后,其中,*tei'* 在 *tei* 之后执行且属于同一线程.假设 *RTei* 为轨迹⟨*tei*,...,*tei'*⟩中不属于线程 *tei.t* 的区间集合,该操作检查 *tei* 中每个节点的远程前驱节点是否在 *RTei* 中.只有所有节点的远程前驱节点都不在中时返回“真”,否则返回“假”.
- *lp-merge*(tei_p,tei_m,*SDT*):用于在 *SDT* 中合并 tei_p 中所有节点到 tei_m 中第 1 个节点之前.
- *ls-merge*(tei_m,tei_s,*SDT*):用于在 *SDT* 中合并 tei_s 中所有节点到 tei_m 中最后一个节点之后.
- *rlink*(*tei*,*tei''*,*SDT*):用于重新连接两个区间 *tei* 和 *tei''*,如图 6 所示.假设 $Tei(3,t_1)$被选择成为合并中心区间,$Tei(2,t_1)$待合并到 $Tei(3,t_1)$之前.$Tei(1,t_2)$为该待合并区间 $Tei(2,t_1)$的前一个区间,$Tei(2,t_2)$为其后一个区间,则该操作将 *SDT* 中 $Tei(1,t_2).next$ 指向 $Tei(2,t_2)$,$Tei(2,t_2).pre$ 指向 $Tei(1,t_2)$,并将 $Tei(2,t_1)$中的节点置入 $Tei(3,t_1)$后返回“*SDT*”.
- *remove*(*tei*,*SDT*):用于从 *SDT* 中移除一个区间 *tei*.此时,需保证被移除的区间中没有任何节点.

为了扩展每个线程中的连续执行区间,算法首先初始化集合 *TEISet* 为轨迹中的所有线程执行区间,并从这个集合中随机挑选一个 *tei* 作为合并中心区间.对于每个被挑中的区间 *tei*,利用上述两个对称的检查操作来判断 *tei* 的本地前驱区间和本地后继区间是否可合并至其之前和之后.1) 在第 6 行~第 11 行中,算法应用操作 *lp-can-merge* 检查该区间 *tei* 的本地前驱区间是否能合并到 *tei* 之前,若返回“是”,则利用 *lp-merge* 合并两区间内的节点,然后更新相邻区间后将原区间删除,并且将 *tei* 的本地前驱区间移除 *TEISet* 中.重复以上步骤,直到返回“否”或者本地前驱为根节点为止.2) 在第 12 行~第 17 行中,算法应用操作 *ls-can-merge* 检查该区间 *tei* 的本地后继区间

是否能合并到 *tei* 之后,若返回“是”,则利用 *ls-merge* 合并两区间内的节点,然后更新相邻区间后将原区间删除,并且 *tei* 的本地后继区间移除 *TEISet*.重复以上步骤,直至返回“否”或者无本地后继为止.最后,将 *tei* 从 *TEISet* 中移除.以上步骤重复进行,直到全部区间都检测完为止.

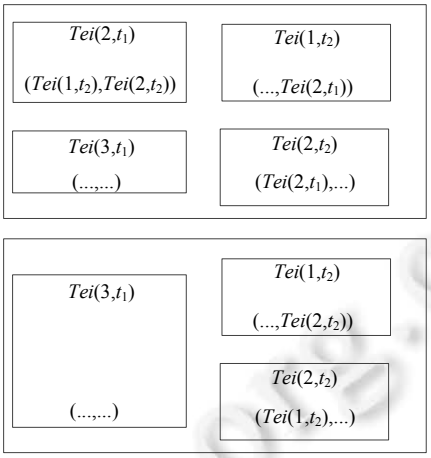


Fig.6 Mergence operations

图 6 合并操作

COSIM 方法仅会在 3 个方面改变依赖树.

1) 对称的合并操作 *lp-merge*/*ls-merge*.由于合并的前提是合并区间之间的其他区间与待移动的区间没有任何远程依赖关系,因此合并操作不会改变轨迹中原有依赖关系.
2) *rlink* 操作.由于该操作没有改变涉及区间的原有顺序,因此不会引入新的依赖关系.
3) *remove* 操作.由于已经保证该被移除区间中不包含任何节点,因此不会改变原轨迹的依赖关系.

因此,算法 2 不会改变原轨迹中的依赖关系,简化后的轨迹与原轨迹等价.

算法 2. Algorithm for congruence-oriended mergence algorithm.

Input: *SDT*.

Output: SDT_m.

Congruence-oriended-mergence(*SDT*)

```
1.  begin
2.    SDT_m←SDT; tei←null
3.    TEISet←SDT.TEI;
4.    while TEISet!=∅ do
5.      tei ← random select a TEI from TEISet
6.      while lp-can-merge(tei.lp,tei) and tei.lp!=root do
7.        SDT_m←lp-merge(tei.lp,tei,SDT_m)
8.        SDT_m←rlink(tei.lp.pre,tei.lp.next,SDT_m)
9.        SDT_m←remove(tei.lp,SDT_m)
10.       TEISet←TEISet/{tei.lp}
11.     end while
12.     while ls-can-merge(tei.ls,tei) and tei.ls!=null do
13.       SDT_m←ls-merge(tei.ls,tei,SDT_m)
14.       SDT_m ←rlink(tei.ls.pre,tei.ls.next,SDT_m)
```

15. SDT_m←*remove*(*tei.ls*,SDT_m)
16. *TEISet*←*TEISet*/{*tei.ls*}
17. **end while**
18. *TEISet*←*TEISet*/{*tei*}
19. **end while**
20. **end**

3.3 轨迹重组

COSIM根据已合并的SDT从根节点开始,沿着每个区间的*next*域,依次取出当前线程执行区间中的节点对应事件以简化轨迹.对于示例轨迹合并之后的结果,轨迹简化为图7.

1 $RCV(g_1,t_0)$		5 $RCV(g_3,t_2)$	14 $RCV(g_4,t_3)$
2 $SND(g_2,t_0)$	7 $RCV(g_2,t_1)$	6 $MEM(\sigma_3,x,READ,t_2)$	15 $MEM(\sigma_5,x,READ,t_3)$
3 $SND(g_3,t_0)$	8 $MEM(\sigma_1,x,READ,t_1)$		
4 $SND(g_4,t_0)$	9 $ACQ(l,t_1)$		
	10 $MEM(\sigma_2,y,WRITE,t_1)$	16 $MEM(\sigma_4,y,READ,t_2)$	18 $MEM(\sigma_6,y,READ,t_3)$
20 $RCV(g_5,t_0)$	11 $REL(l,t_1)$	17 $SND(g_6,t_2)$	19 $SND(g_7,t_3)$
21 $RCV(g_6,t_0)$	12 $MEM(\sigma_7,x,READ,t_1)$		
22 $RCV(g_7,t_0)$	13 $SND(g_5,t_1)$		
23 $SND(g_8,t_0)$			

Fig.7 Recombined trace

图7 重组后的轨迹

由于获取最少线程切换的等价轨迹问题已被证明是NP难问题[9],与SimTrace类似,COSIM不能保证对轨迹进行全局简化只能求得局部简化解.但观察发现,可合并的区间往往会聚集在一定范围内内,COSIM通过迭代搜索本地前驱区间和本地后继区间,能够有效发现聚集的可合并区间.

4 评 价

我们采用基准多线程JAVA程序(IBM ConTest benchmark suite)作为实验对象.该基准程序包括Critical, BuggyPrg,Shop,Mergesort和Bubblesort.本实验的实验平台是一台2.4GHz主频i3-2370M CPU,4GB主存的Linux机器.实验使用Oracle's Java HotSpot 64-bit Server VM(version 1.6.0)进行动态信息的记录.同时,与轨迹简化工具SimTrace采用的随机简化方法进行比较.

实验主要关注以下问题:

1) 简化能力:COSIM的简化能力如何?
2) 简化效率:COSIM的简化效率如何?

4.1 简化能力

实验从降低线程切换数量的角度来比较面向收敛的轨迹简化技术COSIM与现有的静态轨迹简化技术SimTrace,实验结果见表1.我们采用线程数相关工作量*Thr*、共享变量个数*SV*和关键事件数量*E*表示分析轨迹Trace的规模.列Consumed time分别给出COSIM和SimTrace分析所消耗的时间,其中,SimTrace的耗损时间指执行程序50次的时间,COSIM的耗损时间是指执行完成时间.列CS中的子列Tr_o,TR_{cos}和TR_{sim}列出了原轨迹、COSIM简化后轨迹和SimTrace简化后轨迹分别包含的线程切换数.

Table 1 Comparison of simplified ability

表 1 简化能力比较

Program	LOC	Trace			Consumed time	CS		
		Thr	*SV*	*E*		Tr_o	Tr_{co}	Tr_{sim}
Philosopher	81	6	1	131	6ms	54	17	17
Bubble	417	26	25	1 493	23ms	450	159	160
Elevator	514	4	13	2 104	8ms	89	14	16
TSP	709	5	234	636 499	143s	9 301	1 100	1 342
Cache4j	3 897	4	5	1 225 167	570s	409	30	32
Weblench	35 175	3	26	11 630	52ms	170	18	26
OpenJMS	154 563	32	365	376 187	36s	100 163	9 012	11 805
Jigsaw	381 348	10	126	19 074	120ms	2 409	62	68

由实验结果可知,由于 COSIM 采用了面向收敛的合并方法,能够比 SimTrace 采用随机选择的合并算法减少更多的线程切换.

4.2 简化效率

我们先利用 SimTrace 算法执行 50 次的简化操作,得到其对应的简化率;然后,将对应的简化率作为 COSIM 算法的需求,获得对应的简化率所需时间.表 2 列出了相应的简化率数据.由实验数据可知:就绝大多数程序而言, COSIM 能够花费较少的时间取得相同的简化率;仅只对程序 Cache4j,COSIM 的效率稍低于 SimTrace,这是因为 Cache4j 的执行轨迹线程切换分布均匀,因此,SimTrace 采用的随机算法可与 COSIM 采用的面向收敛方式取得相同的效率.

Table 2 Comparison of simplification efficiency

表 2 简化效率比较

程序	轨迹	简化率(%)	花费时间	
			COSIM	SimTrace
Philosopher	131	68	5.4ms	6ms
Bubble	1 493	64	20ms	23ms
Elevator	2 104	82	7.3ms	8ms
TSP	636 499	86	120s	143s
Cache4j	1 225 167	92	620s	570s
Weblench	11 630	85	50ms	52ms
OpenJMS	376 187	88	27s	36s
Jigsaw	19 074	97	90ms	120ms

5 结 论

本文提出了一种面向收敛的执行轨迹静态简化方法 COSIM.COSIM 通过扫描轨迹构建轨迹简化模型.该模型中既包含轨迹中需保留的依赖信息,又标识线程切换位置.基于该轨迹简化模型,COSIM 采用面向收敛的合并算法不断扩大可连续的线程执行区间,以实现降低线程切换数量的目的.实验结果表明,在保留轨迹中依赖信息的同时,COSIM 能够快速而有效地降低线程切换数量,这有助于有效提高程序员理解错误轨迹的效率.

References:

[1] Zamfir C, Altekar G, Candea G, Stocia I. Debug determinism: The sweet spot for replay-based debugging. In: Matt W, ed. Proc. of the 13th Workshop on Hot Toics in Operating Systems. Berkeley: USENIX Asscication, 2011. 18−23.

[2] Veeraraghavan K, Lee D, Wester B, Ouyang J, Chen PM, Flinn J, Narayanasamy S. DoublePlay: Parallelizing sequential logging and replay. ACM Trans. on Computer Systems, 2012,30(1):3:1−3:24. [doi: 10.1145/2110356.2110359]

[3] Hower DR, Hill MD. Rerun: Exploiting episodes for lightweight memory race recording. In: Proc. of the 35th Annual Int'l Symp. on Computer Architecture. Washington: IEEE Computer Society, 2008. 265−276. [doi: 10.1109/ISCA.2008.26]

[4] Georges A, Christiaens M, Ronsse M, Bosschere KD. JaRec: A portable record/replay environment for multi-thread Java applications. Software Practice & Experience, 2004,34(6):523−547. [doi: 10.1002/spe.579]

[5] Montesinos P, Ceze L, Torrellas J. DoLorean: Recoding and deterministically replaying shared-memory multiprocessor execution effciently. In: Proc. of the 35th Annual Int'l Symp. on Computer Architecture. Washington: IEEE Computer Society, 2008. 289−300. [doi: 10.1109/ISCA.2008.36]

[6] Olszewski M, Ansel J, Amarasinghe P. Kendo: Efficient deterministic multithreading in software. In: Soffa ML, Irwin MJ, eds. Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating System. New York: ACM Press, 2009. 97−108. [doi: 10.1145/1508244.1508256]

[7] Huang J, Liu P, Zhang C. LEAP: Light weight deterministic multi-processor replay of concurrent Java programs. In: Roman GC, Sullivan KJ, eds. Proc. of the 18th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2010. 207−216. [doi: 10.1145/1882291.1882323]

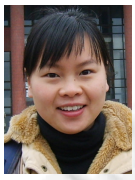
[8] Netzer RHB, Xu YK. Replaying distributed programs without message logging. In: Proc. of the 6th Int'l Symp. on High Performance Distributed Computing. Washington: IEEE Computer Society. 1997. 137−147. [doi: 10.1109/HPDC.1997.622370]

[9] Jalbert N, Sen K. A trace simplification technique for effective debugging of concurrent programs. In: Roman GC, Sullivan KJ, eds. Proc. of the 18th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2010. 57−66. [doi: 10.1145/1882291.1882302]

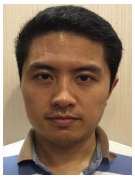
[10] Huang J, Zhang C. An efficient static trace simplification technique for debugging concurrent programs. In: Yahav E, ed. Proc. of the 18th Int'l Static Analysis Symp. Berlin, Heidelberg: Springer-Verlag, 2011. 163−179. [doi: 10.1007/978-3-642-23702-7_15]

[11] Sen K. Race directed random testing of concurrent programs. In: Gupta R, Saman PA, eds. Proc. of the ACMSIGPLAN 2008 Conf. on Programming Language Design and Implementation. New York: ACM Press, 2008. 11−21. [doi: 10.1145/1375581.1375584]

[12] Tallam S, Tian C, Gupta R, Zhang X. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In: Rosenblum DS, Elbaum SG, eds. Proc. of the ACM/SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2007. 207−218. [doi: 10.1145/1273463.1273491]



常曦(1979－),女,湖南长沙人,博士,讲师,CCF 专业会员,主要研究领域为程序测试,程序分析.



张卓(1984－),男,博士生,主要研究领域为程序切片,程序测试.

薛建新(1980－),男,博士,副教授,CCF 专业会员,主要研究领域为并发理论,程序分析,可信软件.

毛晓光(1970－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为可信软件,软件维护与演化.