doi:10.3772/j.issn.1002-0470.2018.07.001

基于静态调度的多线程程序分析方法①

周 卿②*** 李 炼*** 冯晓兵***

(*中国科学院计算技术研究所计算机系统结构国家重点实验室 北京 100190)

(**中国科学院大学 北京 100190)

摘 要 静态多线程程序分析是一种在编译时刻分析多线程程序的执行行为和特征的有效方法。本文通过分析多线程程序实际执行的特点,提出了一种基于静态调度的多线程分析方法。该方法通过模拟多线程程序的动态执行方式,从而在不运行程序的情况下也能较准确地获得多线程的行为特征。实验表明,该分析方法可以有效地提高多线程程序中同步关系的识别和匹配精度,为分析和检测多线程性能瓶颈以及程序错误等信息奠定了基础。

关键词 静态多线程程序分析,多线程控制流图,调度,同步关系,可能并行分析

0 引 言

随着计算机的发展,计算机由单核体系结构发展到多核体系结构,硬件并行计算性能的提升也促使了并行编程的发展。但是并行程序具有难触发、难调试、难检测等特点,于是如何保证和维护并行程序的正确性,以及如何分析并行程序的行为和性能便成为近些年来程序分析的研究热点。静态多线程分析[1]是分析多线程程序的一种重要的方法,其可以在编译时刻对多线程程序进行分析,从而可以检测报告出程序中潜在的行为特征以及错误。

当前,使用静态分析检测多线程程序错误的工作主要有数据竞争(data race,DR)[2-7]、死锁(dead lock,DL)[8]和原子性违反(atomic violation,AV)[9]等。这些分析的精度通常都极其依赖于多线程程序模型的建立以及线程之间同步源语(如 signal-wait 等)[10,11]的识别和匹配,而线程同步源语的匹配问题也一直是一个开放的研究性问题。针对该挑战,本文提出了一种基于静态调度的多线程分析方法,其通过模拟实际多线程程序的实际执行方式,可以随机地调度分析任意一个有效的线程代码,并获得多线程程序代码的组织结构和可能的行为特点,进而在编译时刻最大化地匹配线程间的同步源语,提高静态多线程分析精度。

1 相关概念

1.1 Pthreads

Pthreads[12]编程模型是一种用于多线程实现的函数库,其被广泛应用于 C、C++语言的多线程实现中。在 Pthreads 程序中,涉及线程操作的方法都被定义成一组应用程序接口(application program interface,API)的函数调用。为了简明地描述问题,本文定义如下涉及线程交互的函数:*create*(t, *foo*)用于创建一个子线程,该子线程的线程标识符(identifier ,ID)为 t,而子线程的入口函数为 *foo*;*join*(t)用于终止一个线程标识符为 t 的线程;*lock/unlock*(l)用于上锁和解锁操作,锁变量的名字为 l;*signal/wait*(c)用于线程间同步操作,信号量为 c。

① 国家自然科学基金(61432018)资助。

② 男,1987 年生,博士;研究方向:静态程序分析;联系人,E-mail: zhouqing@ict.ac.cn

(收稿日期:2018-01-16)

1.2　多线程程序的执行特点

多线程程序的执行依赖于操作系统的线程调度，不同的线程调度会引起不同的线程交叠情况的产生，从而导致即使在相同输入的情况下，多线程程序的重复执行也可能会产生不同的结果和状态[4,8]。与此同时，多线程程序的执行又具有某些确定的性质，比如每次执行都不变地具有明确执行序的语句关系，如线程之间的创建 create 关系、线程之间终止 join 关系，以及线程之间的同步 signal-wait 关系。

图 1 为一个 Pthreads 程序的示例。其中，*main* 函数在第 5 行和第 10 行分别创建了子线程 *thread*1 和 *thread*3，而 *thread*1 又在第 17 行创建了子线程 *thread*2；*thread*1 在第 19 行 *signal*(*c*) 发信号唤醒 *main* 的第 7 行的 *wait*(*c*)，从而使得 *main* 在第 9 行 *join*(*tid*2) 可以终止对应的 *thread*2；然后，*main* 又于第 12 行 *signal*(*c*) 发信号唤醒 *thread*1 第 21 行的 *wait*(*c*)，以使得 *thread*1 在第 23 行 *join*(*tid*2) 可以顺利地终止其所对应的 *thread*3；最后，*main* 在第 13 行 *join*(*tid*1) 其所对应的 *thread*1。

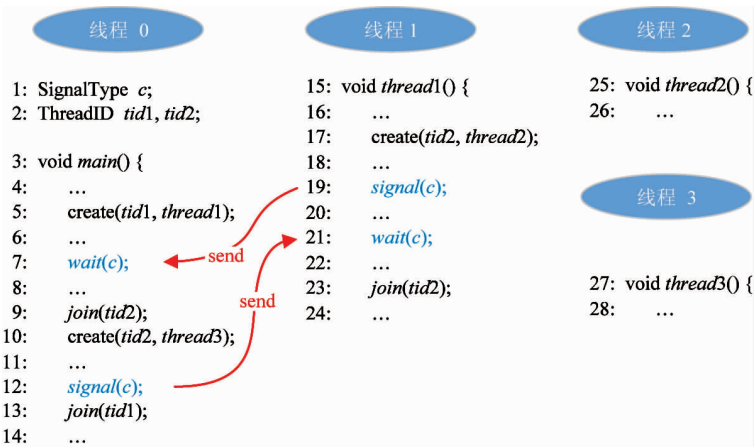

图 1　一个 Pthread 程序的示例

上述程序刻画了一个较为复杂的多线程执行序关系，该执行序关系图如图 2 所示。在实际的多线程程序运行过程中，语句 *join*(*t*) 和 *wait*(*c*) 会停滞当前线程的执行，一直到某些条件满足为止。因此，在图 2 中，*main* 会一直停滞执行，直到 *thread*1 发出信号唤醒 *main* 为止；同时，*thread*1 也会在执行完 *signal*(*c*) 后停滞执行，直到 *main* 发出信号唤醒它为止。这两条 signal-wait 的同步关系同时也促使了 *main* 中的 *join*(*tid*2) 和 *thread*1 中的 *join*(*tid*2) 可以正确的执行。即线程 *thread*1 的生存周期为 *t*0 到 *t*5，线程 *thread*2 的生存周期为 *t*1 到 *t*2，而线程 *thread*3 的生存周期为 *t*3 到 *t*4。在 *t*2 到 *t*3 的时间间隔中，线程标识符 *tid*2 是没有意义的。多线程程序的这种特点，使得静态分析程序试图去匹配 *join*(*tid*2) 和其对应线程的工作充满了挑战。

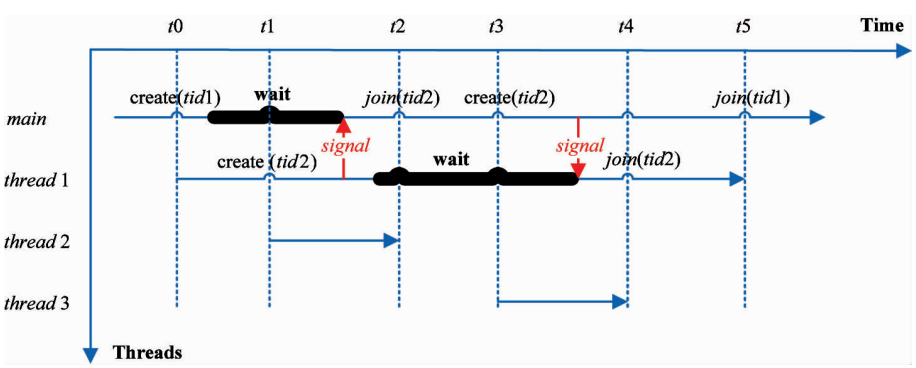

图 2　多线程程序的执行序关系图

面对这种由于多线程程序复杂的执行序关系所引起的同步关系的匹配问题,当前已有的研究工作大多基于启发式方法[10,11],或者忽略该类复杂的同步关系[3-5,13-15]。这些方法通常采用部分分析或者不分析复杂的同步语句关系,从而达到分析结果的保守性。但是,这样的处理方式总是会造成大量真实的先序(happen-before,HB)关系被遗漏,进而造成静态分析结果的不准确。

2　基于静态调度的多线程分析算法

2.1　多线程控制流图

线程是程序执行流的最小单位,是被系统独立调度和分派 CPU 的基本单位。静态线程是动态运行时线程实例的抽象表示。在静态程序分析中,静态线程通常可以用源码中 *create*(*t*, *foo*)语句的位置来表示[10,11,14]。

静态多线程控制流图是一个基于传统控制流图的有向图,可以方便地表达静态线程以及静态线程之间的交互关系。首先为每一个函数构建一个过程内控制流图,图中的每一个节点代表一条程序语句,节点之间通过控制流边相互连接,这些控制流边表达了语句间的控制流关系。本文特别定义如下与线程相关的节点:create 节点代表 *create*(*t*, *foo*)语句,join 节点代表 *join*(*t*)语句,signal 节点代表 *signal*(*c*)语句,wait 节点代表 *wait*(*c*)语句。对于每个函数,为其构建唯一的函数入口节点和唯一的函数出口节点。然后,每个函数的过程内控制流图可以通过函数之间的调用语句连接成为过程间控制流图。最后,通过 *create*(*t*, *foo*)语句将 create 节点与函数 *foo* 的入口节点相连,将 join 节点与其相应的静态线程的出口节点相连接,以及将 signal 节点与其相匹配的 wait 节点相连接,从而构成静态多线程控制流图。

图 3 表达了图 1 中 Pthreads 程序所对应的多线程控制流图。在这个图中,n_2、n_6 和 n_{11} 表示 create 节点,n_5、n_9 和 n_{14} 表示 join 节点,n_7 和 n_{12} 表示 signal 节点,n_4 和 n_{13} 表示 wait 节点;我们称 $n_{12}\xrightarrow{\text{signal}}n_4$ 和 $n_7\xrightarrow{\text{signal}}n_{13}$ 为 signal 边,$n_{17}\xrightarrow{\text{join}}n_5$、$n_{19}\xrightarrow{\text{join}}n_{14}$ 和 $n_{15}\xrightarrow{\text{join}}n_9$ 为 join 边。于是,如何正确地匹配这些 signal 边和 join 边就是本文要解决的重要问题。

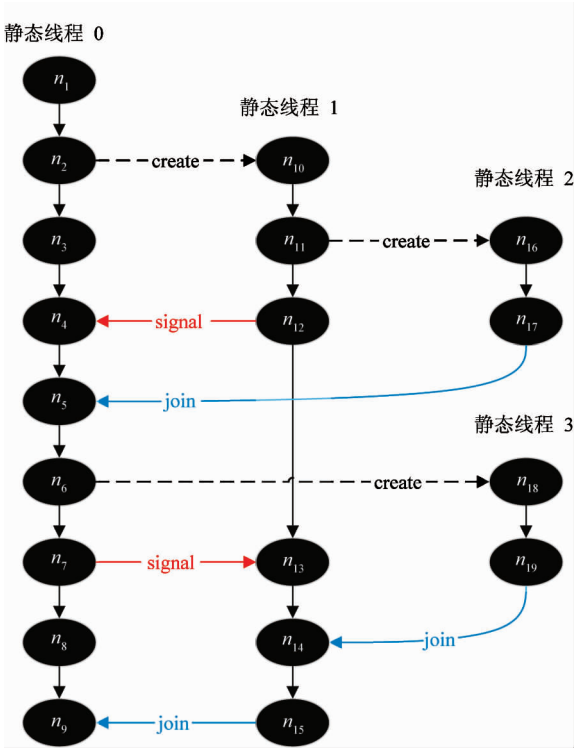

图 3　多线程控制流图

2.2　基于静态调度的多线程分析算法

1.2 节已经介绍了多线程程序实际执行的特点。为了能够准确地匹配同步语句,在分析多线程程序时,本文定义语句 *join*(*t*)和 *wait*(*c*)具有阻塞的含义。即在程序分析的过程中,*join*(*t*)和 *wait*(*c*)有能力阻止分析程序继续分析当前线程的能力,一直到某个条件满足为止。同时,通过构建静态调度器来随机地选择静态线程的语句进行分析,从而达到模拟多线程程序动态执行的特点。为了表述的简明性,定义函数 *push*()为向容器压入元素;函数 *pop*()为从容器的尾端弹出一个元素;函数 *entry*()返回函数的第一条语句;函数 *exit*()返回函数的最后一条语句;函数 *succ*()返回后一条语句的位置;函数 *hungry*()返回当前是否具有新的语句可供分析程序进行分析。

算法 1 基于调度器的多线程程序分析算法

输入:源程序的语句

输出:wait 和 join 语句的匹配关系

1. 令调度器队列 *sch _ list* 的每个元素为一个容器栈 *stack*;*stack* 的每个元素由 < *id*, *p* > 组成,其中 *id* 表示线程标识符,*p* 表示程序语句的位置;
2. *s. push*(< *main. id*, *main. entry*( ) >), 其中 *main. id* 为 *main* 函数的线程标识符;
3. *sch _ list. push*(*s*)
4. **while***sch _ list* 中存在未分析完毕的线程 **begin**
5. 随机选择 *sch _ list* 中的一个元素 *s*
6. 获得 *s* 的栈顶元素 < s_T. *id*, s_T. *p* >
7. **if** s_T. *p* 为函数调用语句 *A*( ) **begin**
8. *s. pop*( ); *s. push*(< s_T. *id*, *succ*(s_T. *p*) >)
9. *s. push*(< s_T. *id*, *A. entry*( ) >)
10. **end**
11. **else if** s_T. *p* 为 *exit*( ) 语句 **begin**
12. *s. pop*( )
13. **end**
14. **else if** s_T. *p* 为线程创建语句 *create*(*tid*, *foo*) **begin**
15. *s. pop*( ); *s. push*(< s_T. *id*, *succ*(s_T. *p*) >)
16. *s'. push*(< *tid*, *foo. entry*( ) >); *sch _ list. push*(*s'*)
17. **end**
18. **else if** s_T. *p* 为线程终止语句 *join*(*tid*) **begin**
19. **if***hungry*( ) **begin**
20. 匹配 *join*(*tid*) 和 *sch _ list* 中线程标识符为 *tid* 的线程
21. 将线程标识符为 *tid* 的线程从 *sch _ list* 中删除
22. *s. pop*( ); *s. push*(< s_T. *id*, succ(s_T. *p*) >)
23. **end**
24. **end**
25. **else if**s_T. *p* 为线程同步语句 *wait*(*c*) **begin**
26. **if***hungry*( ) **begin**
27. 匹配 *wait*(*c*) 和 *sch _ list* 中 *signal*(*c*)
28. *s. pop*( ); *s. push*(< s_T. *id*, succ(s_T. *p*) >)
29. **end**
30. **end**
31. **elsebegin**
32. s. pop( ); *s. push*(< s_T. *id*, succ(s_T. *p*) >)
33. **end**
34. **end**

算法 1 模拟了多线程程序的实际运行机制,以静态调度器为核心,逐条分析所有可达的程序代码,并在此过程中构建出静态线程之间的关系,从而能够最大程度地发掘同步语句的匹配关系。在算法 1 的第 19 行和 26 行,我们使用了函数 *hungry*( ) 来表达了语句 *join*(*t*) 和 *wait*(*c*) 具有阻塞的含义,即对于语句 *join*(*t*),静态调度器总是等待线程标识符为 *t* 的静态程序被分析完毕,才会继续分析 *join*(*t*) 之后的语句;而对于语句 *wait*(*c*),静态调度器总是等待所有可能的 *signal*(*c*) 都被分析后,才会继续分析 *wait*(*c*) 之后的语句。

本文将算法 1 作用于图 3 上。当分析器分析节点 n_5 所对应的 *join*(*tid*2) 语句时,*sch _ list* 中只有静态线程 2 的线程标识符为 *tid*2。然后,该处的 *join*(*tid*2) 语句顺利匹配静态线程 2,并从 *sch _ list* 中删除静态线程 2,表示静态线程 2 已不再存在。最后当分析器分析节点 n_{14} 所对应的 *join*(*tid*2) 语句时,*sch _ list* 中只有静态线程 3 的线程标识符为 *tid*2。因此,算法 1 可以完全正确地匹配所有同步语句的关系。

图 4 给出了上述算法的结构图。图中描述了基于调度器的多线程分析器在分析程序时的某个中间状态:调度器随机调度分析多个静态线程,其中静态线程 0 已经被分析到达 p_1 位置;而静态线程 1 已经被分析到达 p_3 位置(静态线程 1 包含两个函数,图中的 p_2 为一条函数调用语句,表达了静态线程 1 的两个函数之间的调用关系)。在调度器维护的分析列表中,分别记录了上述静态线程的分析位置:{T_0/p_1} 表达了静态线程 0 的分析位置,{T_1/p_2;T_1/p_3} 表达了静态线程 1 的分析位置。其中,{T_1/p_2;T_1/p_3} 表示一个栈结构(栈底为 T_1/p_2,栈顶为 T_1/p_3),表达了位置 p_2 所在函数和位置 p_3 所在函数之间的调用和被调用的关系,即当分析器完成 p_3 所在函数的分析后,可以自动删除{ T_1/p_3},并从新的栈顶{ T_1/p_2} 位置继续分析。

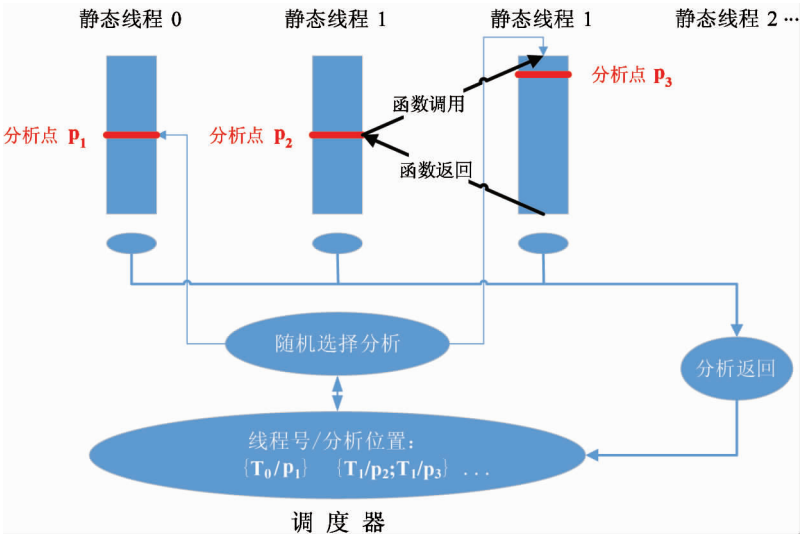

图 4　基于静态调度器的多线程分析算法结构图

但是,对于多生产和单一消费者的多线程程序,算法 1 给出的同步源语的匹配关系会过于激进。多生产和单一消费者的同步源语的匹配问题作为静态程序分析的挑战之一,已有的工作提出了多种启发式方法[11]。在这篇文字中,我们总是采用启发式方法消除多 signal 单 wait 的匹配情况,从而保证静态分析结果的保守性。

3　相关实验

本文的测试程序集分别来自 posix-textsuite、splesh2、icecast 和 apache,实验平台为 8 × Six-Core AMD Opteron(tm) Processor 8425 HE,缓存大小为 512KB,内存大小为 50GB。

表 1 分别列举了各测试程序的代码大小、静态线程个数以及多线程控制流图的节点个数。其中,#LOC为各测试用例的代码行数;#T 统计了不同上下文调用 *create*(*t*, *foo*)所创建的静态线程的个数;#N为采用克隆建图方式所统计的多线程控制流图的节点个数。由表中数据可知,大多数测试用例的静态线程个数都不多,只有 icecast 和 httpd 例外。这是因为 icecast 的线程创建 *create*(*t*, *foo*)位置处于大量条件分支中;而 httpd 的程序规模较大,其调用线程创建 *create*(*t*, *foo*)的位置数目也较多。

表 1　测试用例

Benchmarks	#LOC	#T	#N
posixtext suite. 1-2. c	286	3	140
posixtext suite. pitest-2. c	363	7	335
posixtext suite. stress. c	358	6	406
posixtext suite. s-c1. c	648	3	1097
posixtext suite. s-c2. c	409	3	476
splesh2. radix	1041	2	936
splesh2. volrend	4537	2	3982
splesh2. ocean-non	3410	2	6895
splesh2. fmm	7168	2	10180
splesh2. cholesky	5411	2	59831
icecast	20914	943	364741
httpd	160175	25	125324

上述多线程控制流图的建立可以简单地使用深度优先搜索(depth-first search,DFS)的方式来完成,这也是串行程序分析中常用的手段:将待分析的程序视为语句的集合,并逐条分析每一条语句,当遇见函数调用语句时,优先分析被调用的函数语句,直到待分析程序的所有语句被分析完毕为止。虽然该方法的时间复杂度较低(与程序语句成正比),但是,由于其不能很好地获得待分析程序的实际执行特点,因而无法获得同步源语(join 和 signal-wait)精确的匹配关系(如图 1 所示的程序示例)。因此,本文采用算法 1 的方法,以调度器为核心,在模拟多线程实际运行的机制下构建多线程控制流图,从而精确

地匹配 join 和 signal-wait 同步源语。图 5 给出了采用普通深度优先搜索和基于调度器算法的建图时间的对比直方图(将采用普通深度优先搜索建图的时间归一化)。由图 5 可知,随着待分析程序规模的增大,基于调度器的建图方法所消耗的时间会显著上升(时间比最大的为 httpd 的 147 倍,普通建图时间为 12s,而基于调度器的建图时间为 30min)。基于调度器的建图方法具有较高时间开销的主要原因有:其一,算法 1 中的调度器采用随机调度分析策略,同一个被阻塞的代码片段可能被多次重复调用进行分析;其二,算法 1 通过函数栈的方式来精确地维护了函数调用的层次关系,从而使得同一个函数的代码片段可能被分析多次。面对算法 1 引入的较高分析时间开销的问题,我们可以通过优化调度器策略、设计高效的数据结构来表示函数调用关系,以及减少冗余计算等方法来解决。这也是本文之后的工作方向之一。

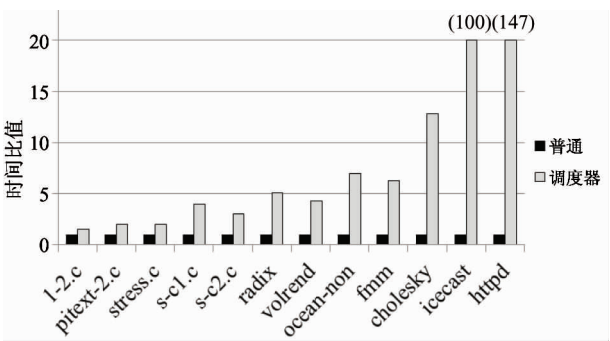

图 5 普通建图方法和基于静态调度器的建图方法的时间比较

进一步,为了验证采用算法 1 匹配同步关系的效果,本实验分别建立 3 种不同的多线程控制流图,并在这些图上运行多线程程序的基础性分析——Naumovich[6] 可能并行分析[10,11,13-18] (may-happen-in-parallel,MHP,静态的分析多线程程序可能并行执行的语句对数)分析。这 3 种不同的多线程控制流图都是保守正确的,它们分别是:不包含 join 边和 signal-wait 边的图 G_1、包含 join 边但不包含 signal-wait 边的图 G_2 以及即包含 join 边和 signal-wait 边的图 G_3(G2 和 G3 的 join 边和 signal-wait 边都是采用算法 1 的建图方式获得的)。本文用 MHP_G 来表示 MHP 分析结果的语句对数。理论上,如果 join 边和 signal-wait 边被正确地加入到多线程控制流图中,那么 MHP 分析的结果就越精确。

图 6 给出了 3 种不同的多线程控制流图对 MHP 计算精度的影响,并将 G1 的 MHP 计算结果归一化。对于大多数的测试程序,都能得到,即说明本文提出的算法可以有效地匹配 join 和 signal-wait 语句,从而直接减少了 MHP 分析结果的误报。但是,对于 splensh2. volrend 和 icecast,MHP_{G2} 和 MHP_{G3} 并没有少于 MHP_{G1},这是因为这两个程序中,没有匹配的同步源语。此外,由实验结果可以发现:对于很多测试用例,MHP_{G3} 的效果和 MHP_{G2} 的效果基本一样。这是由于 signal-wait 语句的匹配通常要难于 join 语句的匹配,并且 signal-wait 更多地出现在较复杂的多线程程序结构中,尤其如果 signal-wait 出现在多生产者单消费者的模型中,那么总是需要利用启发式方法将这类 signal-wait 的匹配删除,以达到 MHP 分析结果的保守性。

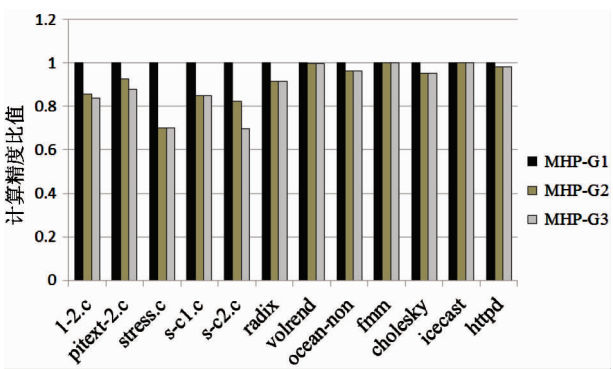

图 6 3 种不同多线程控制流图对 MHP 计算精度的影响

4 结 论

本文从静态多线程程序分析的难点出发,通过实际总结多线程程序实际执行过程中的特点,创新地将操作系统中的调度器思想引入静态多线程程序分析中,从而可以模拟多线程程序实际执行。该基于调度器的静态多线程分析算法,结合了随机调度方法和传统的静态串行程序的分析方法,进而可以在程序分析过程中,逐步发现多线程程序的行为结构特点,提高了同步语句匹配的精度。实验表明,在

本文的测试用例上,该算法所构建的多线程控制流图能够有效地提高 MHP 分析算法的精度。此外,该算法还可以结合其他静态多线程分析问题,比如,数据竞争、死锁分析和原子性违反等应用,进一步提高这些应用的分析精度,从而使得多线程程序的检错更加有效。

参考文献

[1] Duesterwald E, Soffa M L. Concurrency analysis in the presence of procedures using a data-flow framework[C]. In: Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4). New York, USA, 1991. 36-48

[2] Flanagan C, Freund S N. Type-based race detection for Java[C]. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation (PLDI'00). New York, USA, 2000. 219-232

[3] Voung J W, Jhala R, Lerner S. RELAY: static race detection on millions of lines of code[C]. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE'07). New York, USA, 2007. 205-214

[4] Naik M, Aiken A, Whaley J. Effective static race detection for Java[C]. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06), New York, USA, 2006. 308-319

[5] Pratikakis P, Foster J S, Hicks M. LOCKSMITH: Practical static race detection for C[J]. *ACM Transaction. Program Language System*, 2011, 33(1): 1-55

[6] Callahan R, Choi J. Hybrid dynamic data race detection [C]. In: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03),New York, USA, 2003. 167-178

[7] Lee T A, Schardl T B. Efficiently detecting races in Cilk programs that use reducer hyperobjects[C]. In: Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15), New York, USA, 2015. 111-122

[8] Eslamimehr M, Palsberg J. Sherlock: scalable deadlock detection for concurrent programs[C]. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), New York, USA, 2014. 353-365

[9] Flanagan C, Freund S N, Lifshin M, et al. Types for atomicity: Static checking and inference for Java[J]. *ACM Transaction Program Language System*, 2008, 30(4): 1-53

[10] Naumovich G, Avrunin G S, Clarke L A. An efficient algorithm for computing MHP information for concurrent Java programs[C]. In: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7), London, UK, 1999. 338-354

[11] Zhou Q, Li L, Wang L, et al. May-Happen-in-Parallel analysis with static vector clocks[C]. In: Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18), New York, USA, 2018. 228-240

[12] Barney B. POSIX Threads Programming[EB/OL]. https://computing.llnl.gov/tutorials/pthreads/. 2017

[13] Di P, Sui Y L, Ye D, et al. Region-based May-happen-in-parallel analysis for C programs[C]. In: Proceedings of the 44th International Conference on Parallel Processing (ICPP 2015), New York, USA, 2015. 889-898

[14] Barik R. Efficient computation of May-Happen-in-Parallel information for concurrent Java programs[C]. In: Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC 2005), Berlin, Germany, 2005. 152-169

[15] Joshi S, Shyamasundar R, Aggarwal S K. A new method of MHP analysis for languages with dynamic barriers[C]. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PHD Forum (IPDPSW), New York, USA, 2012. 519-528

[16] Sankar A, Chakraborty S, Nandivada V K. Improved MHP analysis[C]. In: Proceedings of the 25th International Conference on Compiler Construction (CC 2016), New York, USA, 2016. 207-217

[17] Albert E, Genaim S, Gordillo P. May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronization[C]. In: Proceedings of the 22th Interna-

tional Static Analysis Symposium, Lecture Notes in Computer Science. Berlin, Heidelberg, 2015. 72-89

[18] Agarwal S, Barik R, Sarkar V, et al. May-happen-in-parallel analysis of X10 programs[C]. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07), New York, USA, 2007. 183-193

Multi-threaded program analysis with static scheduler

Zhou Qing[*,**], Li Lian[*,**], Feng Xiaobing[*,**]

([*] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**] University of Chinese Academy of Sciences, Beijing 100190)

Abstract

Static multi-threaded program analysis is an effective method to analyze the behavior and characteristics of multi-threaded programs at compile time. This paper proposes an algorithm based on static scheduler to simulate the execution of multi-threaded programs. Thus the behavior of multi-threaded programs can be obtained more accurately without running the code. The experimental results show that the method can effectively improve the recognition and matching accuracy of synchronization in multi-threaded programs. And it can be regarded as a fundamental analysis for detecting the performance bottleneck and bugs in multi-threaded programs.

Key words: static multi-threaded program analysis, multi-threaded control flow graph (CFG), schedule, synchronization relation, may-happen-in-parallel analysis