计算机研究与发展 DOI:10.7544/issn1000-1239.2017.20150701
Journal of Computer Research and Development 54(2):428-445,2017

基于未来锁集的死锁规避

禹 振 苏小红 齐 鹏 马培军
(哈尔滨工业大学计算机科学与技术学院 哈尔滨 150001)
(yuzhen_3301@aliyun.com)

Deadlock Avoiding Based on Future Lockset

Yu Zhen, Su Xiaohong, Qi Peng, and Ma Peijun
(*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001)

Abstract Existing dynamic methods for deadlock avoidance have four main drawbacks: limited capability, passive or blind avoiding algorithm, large performance overhead and no guarantee of correctness of target programs. In order to solve these problems, a combined static and dynamic avoiding method based on future lockset is proposed and named as Flider. The key idea is that, for a lock operation, if none of its future locks are occupied, then it makes sure that executing this lock operation won't lead the current thread to trap into a deadlock state. The future lockset for a lock operation is a set of locks that will be requested by the current thread before the corresponding unlock operation is reached. Firstly, Flider statically computes lock effects for lock operations and function call operations, and inserts them before and after the corresponding operations. Secondly, Flider dynamically intercepts lock operations and computes its future lockset using lock effects inserted by static analysis. Flider permits a lock operation to be executed if and only if all locks of its lockset are not held by any other threads. Otherwise, the lock operation waits until the condition is satisfied. Evaluation and comparison experiments verify that this method can efficiently avoid multi-type deadlocks in an active, intelligent, scalable and correctness-guaranteed way.

Key words concurrency bugs; concurrent testing; deadlock bugs; deadlock detection; deadlock avoidance; future lockset

摘 要 针对现有动态死锁规避方法存在能力有限、被动盲目、开销较大和影响目标程序正确性等问题,提出一种基于未来锁集的动静结合死锁规避方案Flider.基本思想是,对于一个加锁操作,若其未来锁集中的所有锁都是空闲的,则执行该加锁操作不会导致死锁.一个加锁操作的未来锁集包括当前要加锁的锁和从该加锁操作到与之相对应的解锁操作过程中遇到的所有加锁操作所要加的锁.通过静态分析,计算锁效应信息并插桩到相应的加锁操作和函数调用操作前后.通过动态分析,劫持加锁操作,根据其锁效应信息为之计算未来锁集,只有当未来锁集中的所有锁都未被锁定才执行该加锁操作,否则等待.测评实验和对比实验表明Flider能智能主动地规避多种类型死锁,开销较小,扩展性好,不影响程序正确性.

关键词 并发缺陷;并发测试;死锁;死锁检测;死锁规避;未来锁集

中图法分类号 TP311

收稿日期:2015-07-29 **修回日期**:2016-12-22

基金项目:国家自然科学基金项目(61173021)

This work was supported by the National Natural Science Foundation of China (61173021).

通信作者:苏小红(sxh@hit.edu.cn)

多线程程序中,如果某线程集合中的每一个线程都在等待另一个线程占据的互斥性资源,或者等待一个不可能发生的信号,则由此导致的循环等待或者无限等待即为死锁.死锁一般由互斥锁、读写锁、条件变量、信号量甚至线程栅栏造成[1-2].互斥锁和读写锁通常导致有环死锁,而条件变量、信号量和线程栅栏则能够导致无环死锁[3].本文只研究有环死锁.

多方统计调查显示[3-5],死锁缺陷占所有并发缺陷的30%以上,严重威胁并发程序的可用性和可靠性.与其他并发缺陷一样,死锁具有难以检测、难以调试和难以修复的特点[6],且其经常在多个线程按照相反的顺序请求某些资源或者等待某个信号的罕见执行交错下发生.死锁难以检测、调试和修复而又仅在罕见的执行交错下才暴露出来,因此如果合理安排线程请求资源或者等待信号的顺序,避开那些包含死锁的执行交错,则即使多线程程序中存在死锁,也不会被触发进而影响程序执行正确性,此即死锁规避.通常使用执行控制方法在目标程序运行时动态规避死锁.

目前研究者们已提出多种方法规避死锁,较为著名和代表较新水平的有 Dimmunix[7],Glock[8],Sammati[9],Grace[10],Flock[11] 和 Scrider[12] 等.Dimmunix 先检测死锁,并记录导致死锁的线程的栈帧作为死锁的签名,然后在并发程序下次运行时,有意识地控制线程调度,阻止相关线程再次进入相应的栈帧,避免已遇到的死锁再次发生.Dimmunix 离线规避死锁,而 Glock 在线规避死锁.Glock 检测潜在的死锁环并在该死锁环对应的代码段前添加门锁,使这些代码段互斥执行以规避死锁.Sammati 和 Grace 都使用基于软件事务内存的回滚/重新执行技术规避死锁,但:1)事务粒度不同,Sammati 以关键区为粒度,而 Grace 以2个线程交互操作(如线程开始/线程创建、线程开始/线程结束和线程开始/信号发送等)之间的代码段为粒度;2)规避方式不同,Sammati 保持互斥锁语义,在对互斥锁加锁时检测是否有死锁环存在,如果存在,则在死锁环中寻找导致当前加锁操作死锁的锁,将当前线程的执行回退到对该锁的最早加锁操作前,并丢弃所有与该最早加锁操作相关的内存更新,重新执行;Grace 则置所有加锁解锁操作为空操作,纯粹以事务内存机制来解决内存冲突,当一个事务执行完毕将要提交时,检查它是否与已经提交的事务存在冲突(即检查它是否读取过时内存),当存在冲突时丢弃该事务从开始到目前为止的所有内存更新,回滚到开始点并重新执行.Flock 动静结合地计算一个即将执行的加锁操作的未来锁集,仅当其中的所有锁都未被占据的情况下才允许该加锁操作执行.Scrider 动态地将多线程程序的指令序列划分为关键区和非关键区2类区域,并保证在任何时刻只有一个线程执行关键区中的指令.

现有死锁规避方法存在4个缺点:1)能力有限,大部分方法(除 Grace 和 Scrider 外)只能规避由互斥锁造成的死锁;2)被动盲目,Dimmunix,Sammati,Glock 需要检测到死锁环后才能规避死锁,而 Scrider 使所有关键区,包括没有可能造成死锁的关键区,串行执行;3)开销较大,在线规避方法如 Glock,Grace,Scrider 等的时间开销都超过10%,Scrider 更超过30%,离线规避死锁方法 Dimmunix,需要重新执行目标程序;4)影响目标程序执行正确性,Grace 和 Sammati 不能回滚 IO 操作,因此可能在规避死锁时导致程序输入输出错误,Scrider 不能处理条件变量,可能导致没有死锁的程序发生死锁或者活锁.

本文的研究目标是"智能、主动、高效地规避多种类型死锁,不影响程序正确性",而 Flock 基于未来锁集技术,通过动静结合分析能在"不影响程序正确性"的前提下,"智能、主动、高效地规避互斥锁死锁".受 Flock 基本思想启发,本文提出一种动静分析结合的基于未来锁集的死锁规避方案 Flider (future lockset based deadlock avoider),如图1所示,使之从死锁规避能力方面改进 Flock,不仅能处理互斥锁还能处理读写锁,以规避由互斥锁和读写锁造成的多种类型死锁,而不仅仅是互斥锁死锁.在图1中,Flider 首先使用 Clang 对源码进行词法语法分析,生成语法树并建立过程内控制流图.然后在过程内控制流图上进行路径敏感分析,计算和插桩锁效应信息,使得加锁操作和函数调用能够"提前知道"将来要进行哪些加锁解锁操作,为动态分析阶段的"智能、主动、高效"规避提供基础.最后通过动态分析,劫持加锁操作,根据静态插桩的锁效应信息计算未来锁集,并根据未来锁集执行规避逻辑动态规避死锁.

在 Flider 中,动态阶段的死锁规避由于有静态阶段的信息辅助,只会使可能存在死锁的2个关键区串行执行,而不影响非关键区和不可能产生死锁的关键区的并行执行.Flider 主动规避死锁,死锁不可能发生,故无需遇到死锁时回滚目标程序并重新执行,从而不会导致程序输入输出错误.

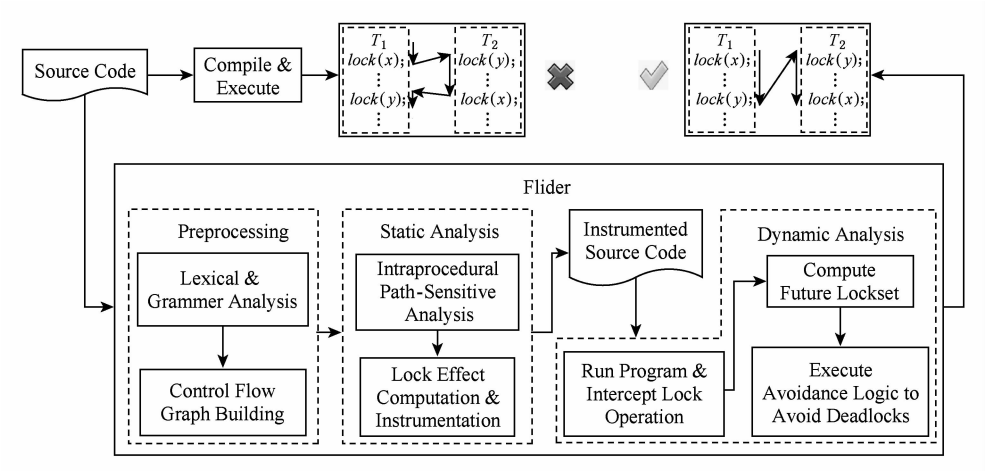


Fig. 1 Scheme of deadlock avoiding based on future lockset

图 1 基于未来锁集的死锁规避方案

1 规避对象

Flider 与 Flock 的死锁规避能力不同,Flock 只能规避由互斥锁导致的死锁,而 Flider 则试图规避由 pthread 库中 2 种同步设施,即互斥锁和读写锁,造成的 5 类死锁:互斥锁造成的死锁、读写锁造成的死锁、互斥锁与读写锁造成的混合死锁、互斥锁自锁和读写锁自锁,如图 2 所示:

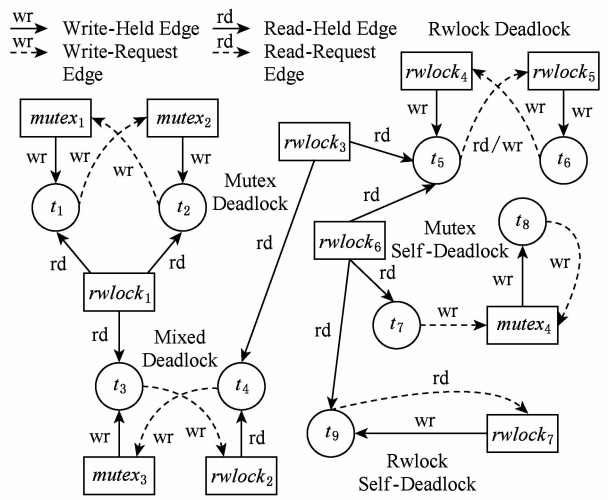


Fig. 2 Five deadlock scenarios caused by mutex and rwlock

图 2 互斥锁和读写锁造成的 5 种死锁场景

实际多线程程序中,互斥锁和读写锁的使用频率都很高.程序员通常使用互斥锁来保护只允许读写操作互斥执行的共享变量,而对于允许多个读操作同时执行而写操作互斥执行的同步场景,程序员为提高性能往往使用读写锁而不是互斥锁来保护共享变量.Flock 不能规避由读写锁造成的死锁,以及读写锁和互斥锁造成的混合死锁,因此 Flider 在死锁规避能力上对 Flock 的扩展具有重要意义和实际应用价值.

互斥锁与读写锁的区别是:互斥锁在任何时刻只能被至多一个线程占据,因此任何线程对互斥锁的占据和请求都是互斥占据和互斥请求;读写锁在任何时刻可被多个线程同时读占据,但只能被至多一个线程写占据,因此对读写锁的占据和请求各自分为 2 类,即共享占据与互斥占据和共享请求与互斥请求.本文将对锁(不论互斥锁还是读写锁)的互斥占据和互斥请求称为写占据和写请求,将对锁的共享占据和共享请求称为读占据与读请求.图 2 中用不同的线型和标签表示这 4 种操作.

本节定义混合锁分配图(multiple-type lock allocation graph, MLAG)以表征多线程程序相对于互斥锁和读写锁的同步状态,并在此基础上给出 5 类死锁的严格定义.

定义 1. 混合锁分配图.混合锁分配图 G 是一个动态的简单图$(V(t),E(t))$,其中:

1) $V(t)$和 $E(t)$分别表示在时刻 t 的顶点和边的集合.

2) $V(t)$中的顶点分成 3 类:代表线程的顶点集合 $T(t)$、代表互斥锁的顶点的集合 $M(t)$和代表读写锁的顶点集合 $RW(t)$,显然 3 个顶点集合中任 2 个交集为空.

3) $E(t)$中的每一条边 e 是一个 3 元组(v,thd,tp_1)或者(thd,v,tp_2),其中 $v\in M(t)\cup RW(t)$,$thd\in T(t)$,$tp_1\in\{\text{wr\_held},\text{rd\_held}\}$,$tp_2\in\{\text{wr\_request},\text{rd\_request}\}$.$(v,thd,tp_1)$表示同步设施 v 正被线程 thd 以 tp_1 方式占据,(thd,v,tp_2)表示线程 thd 正以 tp_2 方式请求同步设施 v.tp_1 和 tp_2 的取值依下列

规则而定:①若 $v\in M(t)$,则 $tp_1=$wr_held,$tp_2=$wr_request.②若 $v\in RW(t)$,则 $tp_1=$wr_held 当且仅当不存在$(v,thr,$rd_held$)$和$(v,thr,$wr_held$)$,$tp_1=$rd_held 当且仅当不存在$(v,thr,$wr_held$)$,其中 $thr\in T(t)$;tp_2 取值 wr_request 或者 rd_request.

混合锁分配图根据多线程程序执行的加锁解锁操作而实时更新,当图上出现环时,可以根据环中顶点的类型和边的类型判断其是否代表死锁以及是哪种死锁.假设 G 中存在一个环 $p=v_1e_1v_2e_2v_3\cdots v_ke_kv_1$,其中 $1\leqslant k\leqslant\min(|V(t)|,|E(t)|)$,$v_1\in(M(t)\cup RW(t))$,记 $tp(e)$为边 e 的到其类型值的映射.根据 G 的定义,k 必定是偶数.

定义 2. 互斥锁死锁.环 p 代表一个互斥锁死锁当且仅当同时满足下列条件:1)$k\geqslant4$;2)如果 i 是奇数,则 $v_i\in M(t)$,$tp(e_i)=$wr_held;3)如果 i 是偶数,则 $v_i\in T(t)$,$tp(e_i)=$wr_request.其中 $1\leqslant i\leqslant k$.

定义 3. 读写锁死锁.环 p 代表一个读写锁死锁当且仅当同时满足下列条件:1)$k\geqslant4$;2)如果 i 是奇数,令 $e_0=e_k$,则 $v_i\in RW(t)$,且$(tp(e_{i-1})\neq$rd_request$\parallel tp(e_i)\neq$rd_held$)$成立;3)如果 i 是偶数,则 $v_i\in T(t)$.其中 $1\leqslant i\leqslant k$.条件 2 要求读写锁顶点的入边和出边不能同时是读类型的边.

定义 4. 混合死锁.环 p 代表一个混合死锁当且仅当同时满足下列条件:1)$k\geqslant4$;2)存在奇数 i 和 j,使得 $v_i\in M(t)$,$v_j\in RW(t)$;3)如果 i 是奇数,令 $e_0=e_k$,则$(tp(e_{i-1})\neq$rd_request$\parallel tp(e_i)\neq$rd_held$)$成立;4)如果 i 是偶数,则 $v_i\in T(t)$.其中 $1\leqslant i\leqslant k$,$1\leqslant j\leqslant k$.

定义 5. 互斥锁自锁.环 p 代表一个互斥锁自锁当且仅当同时满足下列条件:1)$k=2$;2)$v_1\in M(t)$,$v_2\in T(t)$;3)$e_1=$wr_held 且 $e_2=$wr_request.

定义 6. 读写锁自锁.环 p 代表一个读写锁自锁当且仅当同时满足下列条件:1)$k=2$;2)$v_1\in RW(t)$,$v_2\in T(t)$;3)$(tp(e_2)\neq$rd_request$\parallel tp(e_1)\neq$rd_held$)$成立.

2　锁效应与未来锁集

Flider 与 Flock 的基本思想相同,即若一个加锁操作的未来锁集中的所有锁都未被锁定,则执行该加锁操作是安全的,不会导致死锁.一个加锁操作的未来锁集可能分布在多个函数内,因此 Flider 似乎需要进行过程间分析以计算未来锁集.但利用锁效应概念和函数调用机制,Flider 只需在静态阶段进行过程内分析以为加锁操作和函数调用操作计算锁效应信息,而为加锁操作计算跨过程分布的未来锁集则推迟到动态阶段.

Flock 最早针对互斥锁提出锁效应和未来锁集的概念,但未给出形式化定义.本节针对互斥锁和读写锁给出这些概念的严格定义与实例,第 3 节给出死锁规避逻辑并展示基于未来锁集的死锁规避过程.

假设一条非空路径 $path$ 由 n 个指令按顺序构成,即 $path=ins_1,ins_2,\cdots,ins_n$,其中 $n\geqslant2$,ins_n 表示结束指令.令 x 表示一个互斥锁或读写锁,$lock(x)$表示对互斥锁或读写锁 x 的(任意)加锁操作,$unlock(x)$表示对 x 的解锁操作.记 $type(x)$表示锁 x 的类型,取值 MTX 或 RWLOCK.

定义 7. 关键区.指令序列$\langle ins_u,ins_{u+1},ins_{u+2},\cdots,ins_{v-1},ins_v\rangle(1\leqslant u<v\leqslant n)$称为一个关键区,当且仅当:1)$ins_u=lock(x)$且 $ins_v=unlock(x)$;2)若存在 $u<q<v$ 使得 $ins_q=lock(x)$,则存在 $q<w<v$ 使得 $ins_w=unlock(x)$,且指令序列$\langle ins_q,ins_{q+1},ins_{q+2},\cdots,ins_{w-1},ins_w\rangle$也是一个关键区.

关键区既可局限在一个函数内,也可跨越函数而存在.对关键区的定义考虑到了可递归互斥锁的语义.图 3 给出了一个线程可能执行的 5 条路径,其中 x 和 z 是互斥锁,y 是读写锁.在 $path_1$ 中,$\langle ins_2,ins_3\rangle$,$\langle ins_4,ins_5\rangle$,$\langle ins_1,ins_2,ins_3,ins_4,ins_5,ins_6\rangle$都是关键区,而$\langle ins_1,ins_2,ins_3\rangle$,$\langle ins_2,ins_3,ins_4,ins_5\rangle$,$\langle ins_1,ins_2,ins_3,ins_4,ins_5\rangle$都不是关键区,因为$\langle ins_2\rangle$,$\langle ins_3,ins_4\rangle$,$\langle ins_2,ins_3,ins_4\rangle$都不是关键区.在 $path_2$ 中,$\langle ins_1,ins_2,ins_3,ins_4,ins_5\rangle$和$\langle ins_2,ins_3,ins_4,ins_5,ins_6\rangle$都是关键区.

定义 8. 锁效应.假设 $path$ 是非空函数 $f()$的一条路径,ins_n 表示函数结束指令.记 $effect(ins)$为指令 ins 的锁效应信息.如果 ins 不是加锁操作或函数调用操作,则 $effect(ins)$为空.

1) 若存在指令序列$\langle ins_u,ins_{u+1},ins_{u+2},\cdots,ins_{v-1},ins_v\rangle(1\leqslant u<v\leqslant n)$构成一个关键区,则对于 $ins_u=lock(x)$,其锁效应 $effect(ins_u)$按如下步骤计算:①检查所有 ins_i,$u+1\leqslant i\leqslant v-1$,若 $ins_i=lock(y)$且 $y\neq x$,则将$(y,+,type(y))$从尾部插入 $effect(ins_u)$,若 $ins_i=unlock(y)$且 $y\neq x$,则将$(y,-,type(y))$从尾部插入 $effect(ins_u)$;②从尾部向 $effect(ins_u)$插入$(x,-,type(x))$;③逆向遍历 $effect(ins_u)$,对于元素 ef,删除后续遍历过程中遇到的所有与之相等的元素.

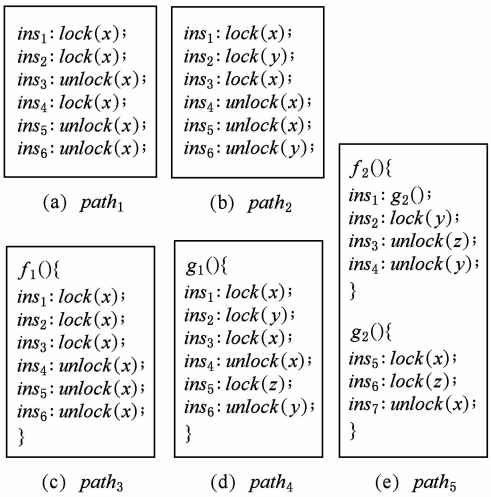


Fig. 3 Five paths possibly taken by a single thread

图 3 单个线程可能执行的 5 条路径

2) 若存在指令 $ins_u=lock(x)$, $1\leqslant u\leqslant n$, 但不存在 $ins_v=unlock(x)$, $u<v\leqslant n$, 使得 $\langle ins_u, ins_{u+1}, ins_{u+2}, \cdots, ins_{v-1}, ins_v\rangle$ 构成一个关键区, 则锁效应 $effect(ins_u)$ 按如下步骤计算: ①检查所有 ins_i, $u+1\leqslant i\leqslant n-1$, 若 $ins_i=lock(y)$ 且 $y\neq x$, 则将 $(y,+,type(y))$ 从尾部插入 $effect(ins_u)$, 若 $ins_i=unlock(y)$ 且 $y\neq x$, 则将 $(y,-,type(y))$ 从尾部插入 $effect(ins_u)$; ②逆向遍历 $effect(ins_u)$, 对于元素 ef, 删除后续遍历过程中遇到的所有与之相等的元素;

3) 若存在指令 ins_u 为自定义函数调用操作, $1\leqslant u\leqslant n$, 则 $effect(ins_u)$ 的计算过程与 2) 相同(除判断任意锁 y 是否等于 x 外, 因为这里没有特定锁 x).

锁效应的概念和定义只局限在单个函数范围内. 只为加锁操作和自定义函数调用操作计算锁效应信息. 对于某函数内一条路径上的加锁操作, 其锁效应是从该加锁操作到与之相对应的解锁操作或者函数结束点的过程中遇到的所有非冗余加锁解锁信息. 函数调用操作的锁效应信息与之类似. 在图 3 的 $path_3$ 中, $effect(ins_1)$, $effect(ins_2)$, $effect(ins_3)$ 都为 {(x, −, MTX)}; $path_4$ 中, $effect(ins_2)$ 为 {(x,+,MTX),(x,−,MTX),(z,+,MTX),(y,−,RWLOCK)}, $effect(ins_1)$ 为 {(y,+,RWLOCK),(z, +, MTX), (y, −, RWLOCK)}; $path_5$ 中, $effect(ins_5)$ 为 {(z, +, MTX), (x, −, MTX)}, $effect(ins_1)$ 为 {(y,+,RWLOCK),(z,−,MTX),(y,−,RWLOCK)}.

定义 9. 未来锁集. 假设 $path$ 为非空线程 thd 的一条路径, ins_n 表示当前 thd 将要执行的指令. 记 $flset(ins)$ 为指令 ins 的未来锁集. 如果 ins 不是加锁操作, 则 $flset(ins)$ 为空. 记 $stack$ 为线程 thd 专用的一个栈, 在 thd 被创建时为空.

1) 若指令 ins_n 是自定义函数调用操作 call $f()$, 将锁效应 $effect(ins_n)$ 中的元素逆序入栈 $stack$.

2) 若指令 ins_n 是自定义函数返回操作 ret $f()$, 假设与之相对应的自定义函数调用操作是 $ins_u=$ call $f()$, $1\leqslant u\leqslant n$, 则从栈 $stack$ 中正序弹出锁效应 $effect(ins_u)$.

3) 若指令 ins_n 是加锁操作 $lock(x)$, 则首先将 x 从尾部插入 $flset(ins_n)$, 并将 $effect(ins_n)$ 逆序入栈 $stack$, 然后从栈顶向栈底查找 $(x,-,type(x))$, 在查找过程中一旦遇到 $(y,+,type(y))$ 且 $y\neq x$, 就将 $(y,type(y))$ 从尾部插入 $flset(ins_n)$. 不论成功找到或者直到栈底都没有找到 $(x,-,type(x))$, 都从栈 $stack$ 中逆序弹出 $effect(ins_n)$.

未来锁集的概念与定义只局限在单个线程范围内. 只为加锁操作计算未来锁集. 对于某线程内一个路径上的加锁操作, 其未来锁集包括该加锁操作所要加的锁和从该加锁操作到与之相对应的解锁操作或线程结束点的过程中遇到的所有非冗余加锁操作所要加的锁. 在图 3 的 $path_3$ 中, $flset(ins_1)$, $flset(ins_2)$, $flset(ins_3)$ 都为 {(x,MTX)}; $path_4$ 中, $flset(ins_2)$ 为 {(y,RWLOCK),(x,MTX),(z,MTX)}; $path_5$ 中, $flset(ins_6)$ 为 {(z,MTX),(y,RWLOCK)}.

3 规避逻辑

假设一个多线程程序有 $thd_1, thd_2, \cdots, thd_m$ 共 m 个线程, $m\geqslant 1$, 线程 thd_i 到目前为止的执行路径是 $path_i$, $1\leqslant i\leqslant m$. $path_i$ 由指令 $ins_{i,1}, ins_{i,2}, \cdots, ins_{i,len(i)}$ 构成, 其中 $len(i)$ 表示当前路径 $path_i$ 的长度.

定理 1. 对于线程 thd_i, 若当前路径 $path_i$ 的当前指令 $ins_{i,len(i)}=lock(x)$, 且其未来锁集 $flset(ins_{i,len(i)})=\{l_1,l_2,\cdots,l_s\}$, 其中 x 是一个互斥锁或者读写锁, $l_1=x$, $s\geqslant 1$, 那么下列命题成立: 若任意 l_j 都没有被占据, $1\leqslant j\leqslant s$, 则 thd_i 执行 $ins_{i,len(i)}$ 会成功获取锁 x 且不会导致死锁.

证明. 1) 因为任意锁 l_j (包括 x) 都没有被锁定, $1\leqslant j\leqslant s$, 所以执行 $ins_{i,len(i)}$ 必定能成功获取锁 x; 2) 若成功获取锁 x 导致线程 thd_i 在将来执行与 $ins_{i,len(i)}$ 对应的释放锁 x 操作之前的某个时刻参与到某个死锁环, 则说明当 thd_i 执行 $ins_{i,len(i)}$ 时, 某个

线程 thd_h 已占据$\{l_1,l_2,\cdots,l_s\}$中的某个锁,$1\leqslant h\leqslant m$,否则根据混合锁分配图(定义 1)的定义,thd_i 不可能参与构成死锁环,但根据命题前提可知,这不可能发生,因此 thd_i 不会陷入死锁. 证毕.

定理 1 指出,只有在加锁操作的未来锁集中所有锁都未被占据的情况下,线程执行加锁操作才不会导致死锁,但没有说明若未来锁集中存在某些锁被占据的情况下,线程应执行何种动作.实际上,在这种情况下,Flider 让线程延迟执行加锁操作直到该加锁操作的未来锁集中所有锁都空闲.定理 1 也没有考虑到加锁操作的未来锁集中的锁已被当前线程所占据的情况.在实现时,当线程 thd 执行加锁操作 $lock(x)$,若发现 x 已被自身占据,则将该加锁操作和相应的解锁操作视为空操作而执行,若发现该加锁操作的未来锁集中某个非 x 的其他锁已被自身占据,则仍然执行该加锁操作而不等待(当然需要检查 x 是否已被自身占据).这样 Flider 就能够规避互斥锁自锁与读写锁自锁(4.5 节).

定理 2. 对于线程 thd_i,若当前路径 $path_i$ 的当前指令 $ins_{i,len(i)}=lock(x)$,且其未来锁集 $flset(ins_{i,len(i)})=\{l_1,l_2,\cdots,l_s\}$,其中 x 是一个互斥锁或者读写锁,$l_1=x$,$s\geqslant 1$,那么下列命题不成立:若 $ins_{i,len(i)}$ 是写请求加锁,则若任意 l_j,$1\leqslant j\leqslant s$,都没有被(写或读)占据,则 thd_i 执行 $ins_{i,len(i)}$ 不会导致死锁;若 $ins_{i,len(i)}$ 是读请求加锁,则若任意 l_j 都没有被写占据,则 thd_i 执行 $ins_{i,len(i)}$ 不会导致死锁.

证明. 若要证明定理 2 成立,只需构造一个例子使得其中的命题不成立.假设图 4 中多线程代码的执行顺序是 $ins_{1,1}$,$ins_{2,1}$,$ins_{1,2}$,$ins_{2,2}$,x 和 y 是读写锁.

thd_1	thd_2
$ins_{1,1}$: $rdlock(x)$;	$ins_{2,1}$: $rdlock(y)$;
$ins_{1,2}$: $wrlock(y)$;	$ins_{2,2}$: $wrlock(x)$;
$ins_{1,3}$: $unlock(x)$;	$ins_{2,3}$: $unlock(y)$;
$ins_{1,4}$: $unlock(y)$;	$ins_{2,4}$: $unlock(x)$;

Fig. 4　A deadlock example caused by rwlocks

图 4　一个读写锁死锁例子

图 4 中的指令根据定理 2 中命题的规则执行.当线程 thd_1 执行 $ins_{1,1}$ 时,由于 $ins_{1,1}$ 是读请求加锁且 $flset(ins_{1,1})=\{(x,\text{RWLOCK}),(y,\text{RWLOCK})\}$ 中的所有锁都未被写占据,则 thd_1 执行 $ins_{1,1}$ 成功地读占据 x.然后当 thd_2 执行 $ins_{2,1}$ 时,由于 $ins_{2,1}$ 也是读请求加锁且 $flset(ins_{1,1})=\{(y,\text{RWLOCK}),(x,\text{RWLOCK})\}$ 中的所有锁都未被写占据,则 thd_2 执行 $ins_{2,1}$ 成功地读占据 y.当 thd_1 执行 $ins_{1,2}$ 时,由于 $ins_{1,2}$ 是写请求加锁且 $flset(ins_{1,2})=\{(y,\text{RWLOCK})\}$ 中的 y 已被线程 thd_2 读占据,所以 thd_1 不执行 $ins_{1,2}$ 而等待 thd_2 释放 y.当 thd_2 执行 $ins_{2,2}$ 时,由于 $ins_{2,2}$ 是写请求加锁且 $flset(ins_{2,2})=\{(x,\text{RWLOCK})\}$ 中的 x 已被线程 thd_1 读占据,所以 thd_2 不执行 $ins_{2,2}$ 而等待 thd_1 释放 x.这样就造成了 thd_1 和 thd_2 之间的相互循环等待,也即死锁,因此定理 2 成立. 证毕.

定理 2 指出比定理 1 更细粒度的规避逻辑是不可行的.即对于一个读请求加锁操作,即使其未来锁集中的锁都被读占据,Flider 也不会允许其执行,因为根据定理 2,允许此读请求加锁操作执行可能导致程序在将来遭遇死锁.在定理 1 和定理 2 中,检查未来锁集中的锁是否已被占据是通过搜索混合锁分配图来完成的.在目标程序运行时,Flider 通过劫持加锁解锁操作以建立和更新混合锁分配图.

Flider 进行过程内分析,为每条执行路径上的加锁操作和自定义函数调用操作计算锁效应信息(定义 8)并插桩到相应操作执行前后.而在动态分析时利用插桩的锁效应信息为加锁操作计算未来锁集(定义 9).最后根据未来锁集中的锁是否空闲(定义 1)执行规避逻辑(定理 1)以控制线程调度规避死锁.基于定理 1,Flider 只使可能发生死锁的关键区串行,而不会影响不可能发生死锁的关键区和非关键区的并行执行,从而达到"智能、高效"的规避目标.我们用图 5 展示整个基于未来锁集的死锁规避过程.

在图 5 中,首先静态分析源码,针对过程内的单个路径为加锁操作和自定义函数调用操作计算锁效应信息,这里我们省略锁效应信息中的锁类型信息,例如 $g_T_1()$内 $lock(x)$的锁效应信息为$\{y+,y-\}$,$f_T_2()$内$g_T_2()$的锁效应信息是$\{y-\}$;然后按照图中标示的顺序执行代码.在 T_1 调用 $g_T_1()$时,将锁效应信息 $effect(f_T_1:g_T_1())$入栈 $stack\#T_1$,在执行 $g_T_1()$内 $lock(x)$时,先将锁 x 加入未来锁集 $flset(g\_T_1:lock(x))$,并将锁效应 $effect(g_T_1:lock(x))$入栈,然后在栈中查找 $x-$,将查找过程中遇到的 $y+$中的 y 从尾部插入 $flset(g\_T_1:lock(x))$,于是得到 g_T_1 内 $lock(x)$的未来锁集是$\{x,y\}$.由于这 2 个锁都没有被锁定,因此 Flider 允许 T_1 执行 $lock(x)$以成功占据锁 x.线程 T_1 占据锁 x 后,操作系统调度 T_2 执行.在 T_2 调用 $g_T_2()$时,将锁效应信息 $effect(f_T_2:g_T_2())$入栈 $stack\#T_2$,在执行 $g_T_2()$内 $lock(y)$时,先将锁 y 加入未来锁集

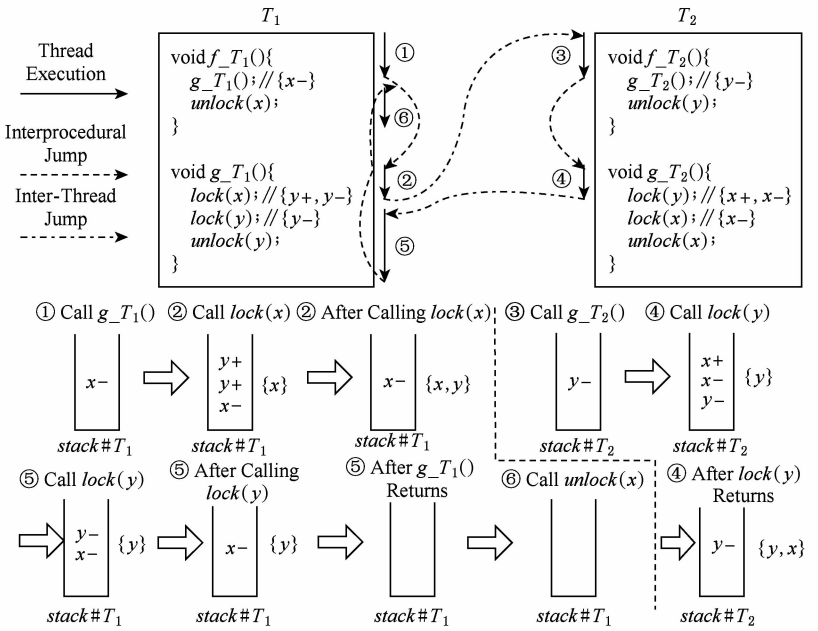


Fig. 5 Demonstration of deadlock avoiding mechanism based on future lockset

图 5 基于未来锁集的死锁规避示例

$flset(g\_T_2:lock(y))$，并将锁效应 $effect(g_T_2:lock(y))$入栈，然后在栈中查找 $y-$，将查找过程中遇到的 $x+$中的 x 从尾部插入 $flset(g\_T_2:lock(y))$，于是得到 g_T_2 内 $lock(y)$的未来锁集是$\{y,x\}$. 由于$\{y,x\}$中的一个锁 x 已被 T_1 占据，因此 Flider 不允许 T_2 执行 $lock(y)$而是等待直到 y 和 x 都变得空闲. 此时操作系统挂起 T_2 而去重新执行 T_1. 在 T_1 执行 $g_T_1()$内 $lock(y)$，先计算出 $flset(g_T_1:lock(y))$为$\{y\}$. 由于 y 未被占据，则 Flider 允许 T_2 执行 $lock(y)$以成功获取锁 y. 在 $g_T_1()$返回时，将 $effect(f_T_1:g_T_1())$从栈中弹出. T_1 执行 $unlock(x)$和 $unlock(y)$时，Flider 会更新混合锁分配图，从而 T_2 能够通过搜索混合锁分配图知道 x 和 y 已被释放，进而执行 $g_T_2()$内 $lock(y)$. 至此，Flider 成功规避了图 5 中多线程代码在执行时可能发生的死锁.

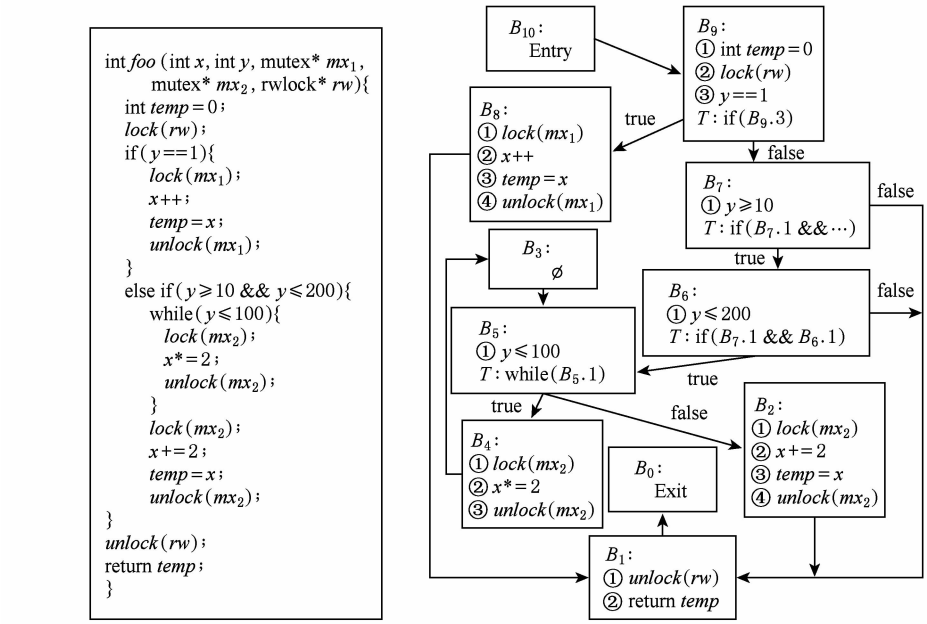
4 Flider 实现

我们在 Ubuntu 14.04(内核：Linux-3.13.0)上实现了 Flider. 如图 1 所示，Flider 主要包括预处理、静态分析和动态分析三大模块，分别负责完成过程内控制流图建立、锁效应计算与插桩以及未来锁集计算与规避逻辑执行等功能. 与 Flock 不同，Flider 使用 Clang 而不是 CIL[13] 进行预处理和锁效应插桩；另外 Flider 使用路径敏感分析技术而不是类型推理计算锁效应. 前者使得 Flider 更通用，能处理比 Flock 更多类型的源程序；而后者使得 Flider 的锁效应计算更简洁和易于理解.

4.1 过程内控制流图建立

使用 Clang 对多线程程序源码进行词法语法分析，建立抽象语法树并在此基础上建立过程内控制流图. Clang 建立的控制流图是一个由基本块组成的有向图，利用该图可以方便地判断程序结构，了解程序中各语句之间的先后执行顺序，获得程序的执行路径. 基本块代表一个可顺序执行的语句序列，其中包括块号、语句体、分支条件和前驱后继. 如图 6(a)是函数 $foo()$的源码，图 6(b)是 Clang 为之生成的控制流图.

图 6(b)中的每个节点都代表一个基本块，从 B_0 到 B_{10} 共 11 个基本块，其中 B_{10} 和 B_0 分别是 $f()$的入口块和出口块，这 2 个块对每个函数都有且只有一个；B_5，B_4，B_3 构成了一个环，对应图 6(a)中的 while 循环；B_3 是只有前驱和后继信息的空块，用于清晰地表示循环结构，每个 while 循环都至少有一个空块；非空块中的带标号文本为块中语句或表达式，与源码中的语句或表达式相对应；带有 T 的块表示分支块，表示源码中的分支结构，T 后的文本表示分支块的分支条件；有向边表示块之间的前驱后继关系，分支块的出边带有标签 true 或 false，分别指向条件为真或为假时的后继块.



(a) Function *foo*()　(b) Control flow graph generated by Clang for *foo*()

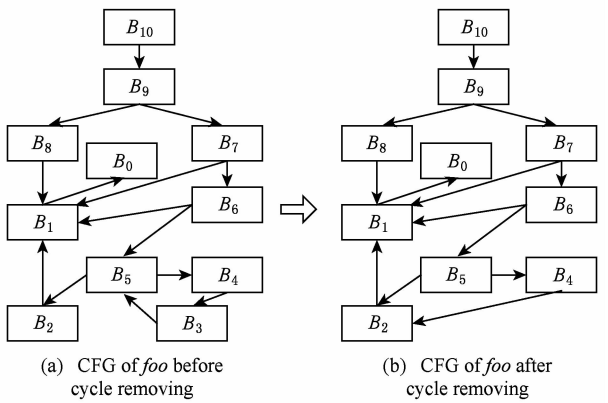
Fig. 6　Function *foo*() and its CFG generated by Clang

图 6　函数 *foo*()及其控制流图

4.2　过程内控制流图路径分析

Flider 针对过程内每条路径为加锁操作和自定义函数调用操作计算锁效应(定义 8),因此在计算锁效应之前需要在过程内控制流图上进行路径分析,获取从入口块到出口块的所有可能路径.

简单的深度或者广度优先遍历,只能分析有向图中 2 个节点的可达性,但不能计算出这 2 个节点之间的所有互异可达路径,因此我们提出算法 1 所示的路径分析算法求取从入口块到出口块的所有可能路径.路径分析算法不适用于带环的有向图,故在进行路径分析之前,需要先对过程内控制流图进行去环处理,即将循环结构改为循环体只执行一次的分支结构.图 7 展示了图 6 中控制流图的去环过程,其中控制流图的基本块节点以块号表示.可以看到图 7(a)中基本块 B_5,B_4,B_3 构成一个环,去环过程中,我们将链接 B_4 和 B_5 的空块 B_3 去除,将 B_4 的后继指向 B_5 的后继 B_2,从而得到图 7(b)所示的无环控制流图.



(a) CFG of *foo* before cycle removing　(b) CFG of *foo* after cycle removing

Fig. 7　Cycle removing on CFG of function *foo*()

图 7　对函数 *foo*()的控制流图进行去环处理

算法 1 以邻接矩阵 $Arcs[N][N]$为输入,它表示经过去环处理的控制流图,N 是所有节点的个数.算法 1 深度优先遍历控制流图,每个节点可能被多次遍历,使用栈 *nstack* 存储遍历过程中找到的可能构成路径的节点,使用 $VertexStatus[N]$ 和 $ArcStatus[N][N]$这 2 个数据结构控制节点的入栈和出栈.对于 $0\leqslant i\leqslant N$,$VertexStatus[i]=0$ 表示 i 号节点未进栈或者已出栈,否则 $VertexStatus[i]$为 1.对于 $0\leqslant i,j\leqslant N$,$ArcStatus[i][j]=0$ 当且仅当节点 i 和节点 j 都在栈外,否则 $ArcStatus[i][j]=1$.算法 1 中,$Traverse(nstack)$遍历栈 *nstack*,将从栈底到栈顶的块号按序排列组成一条所求路径.$UpdateArcStatus(VertexStatus, ArcStatus)$根据 $VertexStatus$ 更新 $ArcStatus$,使得所有 2 个顶点都不在栈内的边的状态为 0.

算法 1. 过程内控制流图上路径分析算法.

输入:过程内控制流图邻接矩阵形式 $Arcs[N][N]$;

输出:过程内控制流图上从基本块 N 到基本块 0 的所有路径集合 *paths*.

```
① paths={}; arg1=VertexStatus; arg2=
     ArcStatus;
② for i=1 to N,VertexStatus[i]=0;
③ for i=1 to N,j=1 to N,ArcStatus[i][j]=0;
④ nstack.push(N); /*块 N 入栈*/
⑤ VertexStatus[N]=1;
⑥ while !nstack.empty(){
⑦   elem=nstack.top(); /*获取栈顶元素*/
⑧   if elem==0{
⑨     path=Traverse(nstack);
⑩     paths.add(path);
⑪     VertexStatus[elem]=0;
⑫     UpdateArcStatus(arg1,arg2);
⑬     nstack.pop(); /*弹出栈顶元素*/}
⑭   else{
⑮     for i = N to 0{
⑯       let C1 be VertexStatus[i]==0;
⑰       let C2 be ArcStatus[elem][i]==0;
⑱       let C3 be Arcs[elem][i]==1;
⑲       if C1&&C2&&C3{
⑳         VertexStatus[i]=1;
㉑         ArcStatus[elem][i]=1;
㉒         nstack.push(i); /*块 i 入栈*/
㉓         break;}}
㉔     if i==0{
㉕       VertexStatus[elem]=0;
㉖       nstack.pop(); /*弹出栈顶元素*/
㉗       UpdateArcStatus(arg1,arg2);}}}
㉘ return paths.
```

算法 1 首先将起始块 N 入栈 $nstack$,并置 $VertexStatus[N]$ 为 1 表示块 N 已入栈(行④～⑤),然后检查栈是否为空,当栈不为空且栈顶元素是块 0 时,则栈中从栈底到栈顶的块号序列构成一条所求路径,生成路径并加入到路径集合中,将块 0 从栈顶弹出,并相应更新 $VertexStatus$ 和 $ArcStatus$,使得块 0 的入栈状态为 0 和以块 0 为其中一个顶点的 2 个顶点都不在栈中的边的入栈状态为 0(行⑦～⑬),如果栈不为空且栈顶元素不是块 0 时,则寻找序号最大的的块 i,该块不在栈中(条件 C_1)且栈顶块 $elem$ 与块 i 形成的边(如果有边的话,条件 C_3)也不在栈中(条件 C_2),将块 i 入栈,置边($elem$,i)的入栈状态为 1(行⑭～㉓),如果不存在这样的块 i,则说明在当前情况下,块 $elem$ 的所有出边已被访问过,此时向上回溯,将块 $elem$ 弹出并相应地更新 $VertexStatus$ 和 $ArcStatus$(行㉔～㉗). 最后,算法返回所有合法路径(行㉘).

需要注意的一点是,为保证正确性和深度优先遍历特性,算法 1 仅在弹出栈顶元素时调用 $UpdateArcStatus(VertexStatus, ArcStatus)$. 在图 7(b)上运行算法 1,得到所有可能执行路径:

$path_1: B_{10}\rightarrow B_9\rightarrow B_7\rightarrow B_6\rightarrow B_5\rightarrow B_2\rightarrow B_1\rightarrow B_0$,

$path_2: B_{10}\rightarrow B_9\rightarrow B_7\rightarrow B_6\rightarrow B_5\rightarrow B_4\rightarrow B_2\rightarrow B_1\rightarrow B_0$,

$path_3: B_{10}\rightarrow B_9\rightarrow B_7\rightarrow B_6\rightarrow B_1\rightarrow B_0$,

$path_4: B_{10}\rightarrow B_9\rightarrow B_7\rightarrow B_1\rightarrow B_0$,

$path_5: B_{10}\rightarrow B_9\rightarrow B_8\rightarrow B_1\rightarrow B_0$,

其中→表示直接可达.

4.3 锁效应计算与插桩

针对函数 $f()$的控制流图中的一条路径 $path$,Flider 按照定义 8 为其上加锁操作或者自定义函数调用操作计算锁效应. 计算出锁效应后,根据 $path$ 是否含有分支块,按照不同的插桩规则将锁效应信息插桩到相应操作前后,以便传递到动态分析模块供计算未来锁集之用. 假设函数 $f()$在线程 thd 中,针对锁效应中的一个元素(x,+,MTX),则在相应操作之前和之后的插桩语句分别是 $stack\sharp thd.push(x,1)$和 $stack\sharp thd.pop(x,1)$,针对元素(y,−,RWLOCK),在相应操作之前和之后的插桩语句是 $stack\sharp thd.push(y,0)$和 $stack\sharp thd.pop(y,0)$. 这里有 2 点需要注意:1)由于锁的类型信息在未来锁集计算和规避逻辑执行时无用(定理 1 和定理 2),我们省略对其的插桩,不将其传递到动态分析模块;2)由于静态分析阶段无法确定线程的标识,线程专用的栈 $stack\sharp thd$ 是动态阶段通过映射动态创建和查找的,我们静态插桩能够自动完成这种映射的代码.

如果 $path$ 不含有任何分支块,则 $f()$是不包含任何分支语句的顺序结构函数,将锁效应信息直接插桩到相应操作的前后. 图 8 展示了对图 5 中线程 T_1 执行的 2 个顺序结构函数 $f_T_1()$和 $g_T_1()$的插桩结果. 函数 $g_T_1()$中需要插桩的语句有 $lock(x)$和 $lock(y)$,根据它们的锁效应信息,在 $lock(x)$前后插桩语句 $stub_3$,$stub_4$ 和 $stub_5$,$stub_6$,在 $lock(y)$前后插桩语句 $stub_7$ 和 $stub_8$. 函数 $f_T_1()$中需要插桩的语句是自定义函数调用 $g\_T_1()$,在其前后插桩语句 $stub_1$ 和 $stub_2$.

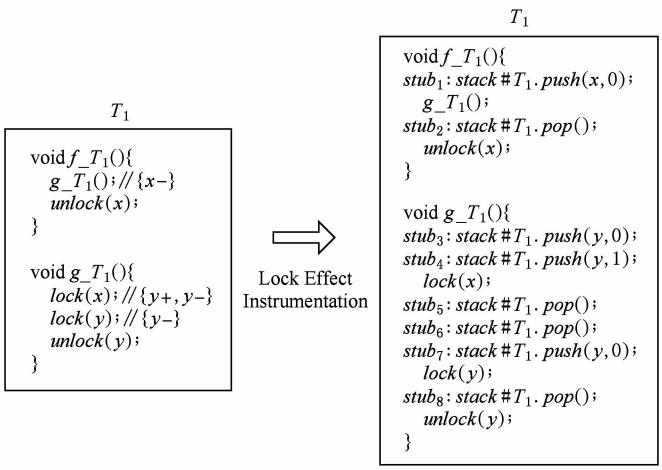


Fig. 8　Lock effect instrumentation for functions with sequential structures

图 8　顺序结构函数的锁效应插桩

顺序结构函数只含有一条路径,对其锁效应插桩较简单.但若函数 $f()$ 含有条件分支语句(循环控制语句是条件分支语句的一种),则其控制流图必定具有多条路径,且每条路径都至少包含一个分支块.对分支结构函数的锁效应插桩较复杂.

对于路径 $path=B_N\rightarrow B_{b_1}\rightarrow B_{b_2}\rightarrow\cdots\rightarrow B_{b_n}\rightarrow B_0$,若 B_{b_i} 是分支块,则令 C_{b_i} 表示 B_{b_i} 的分支条件,其中 $b_1,b_2,\cdots,b_n\in\{0,1,2,\cdots,N\}$,$n\geqslant 0$,$N\geqslant 1$.令 $B_p\rightarrow^+B_q$ 表示 B_p 经过至少一个块可达 B_q,$0\leqslant p,q\leqslant N$.假设 $path$ 内的所有分支块 $B_{v_1},B_{v_2},\cdots,B_{v_r}$ 构成 $B_{v_1}\rightarrow^+B_{v_2}\cdots\rightarrow^+B_{v_r}$,$C_{v_1},C_{v_2},\cdots,C_{v_r}$ 分别是各自块的分支条件,其中 $v_1,v_2,\cdots,v_r\in\{b_1,b_2,\cdots,b_n\}$,$r\geqslant 0$.Flider 按下列规则为路径 $path$ 上的加锁操作和自定义函数调用操作计算和插桩锁效应信息:

1) 按照定义 8 计算锁效应.

2) 若某个加锁操作与相应的解锁操作(这 2 个指令与它们之间的指令构成一个关键区)都在一个块内,则直接在该加锁操作前后插入相应的 $push$ 和 pop 语句.

3) 设存在 $u\in\{b_1,b_2,\cdots,b_n\}$,使得 B_u 不是分支块且 $B_u\rightarrow^+B_{v_1}\rightarrow^+B_{v_2}\cdots\rightarrow^+B_{v_r}$.若加锁操作指令 ins_l 在 B_u 内,且其相应的解锁指令不在 B_u 而在 B_{v_1} 或者 B_u 与 B_{v_1} 之间的非分支块中,则直接在 ins_l 前后插入相应插桩语句.若 ins_l 的相应解锁指令不在 B_u 而在 B_{v_s},$2\leqslant s\leqslant r$,则插桩在 ins_l 前后的 $push$ 和 pop 语句集各自位于条件语句 if(C_{v_1} && C_{v_2} && … && $C_{v_{s-1}}$)中.若 ins_l 的相应解锁指令不在 B_u 而在 B_{v_s} 与 $B_{v_{s+1}}$(规定 $B_{v_{r+1}}$ 为出口块)之间的某个非分支块中,$2\leqslant s\leqslant r$,则插桩在 ins_l 前后的 $push$ 和 pop 语句集各自位于条件语句 if(C_{v_1} && C_{v_2} && … && C_{v_s})中.若自定义函数调用指令 ins_f 在 B_u 内,则插桩在 ins_f 前后的 $push$ 和 pop 语句集各自位于条件语句 if(C_{v_1} && C_{v_2} && … && C_{v_r})中.

4) 设存在 $u\in\{b_1,b_2,\cdots,b_n\}$,使得 B_u 不是分支块且 $B_{v_1}\rightarrow^+B_{v_2}\cdots\rightarrow^+B_{v_r}\rightarrow^+B_u$.若 ins_l 和 ins_f 分别是 B_u 中的加锁指令和自定义函数调用指令,则直接将锁效应信息插桩到相应指令前后.

5) 对于分支块 B_{v_w} 或位于 $B_{v_{w-1}}$(规定 B_{v_0} 为入口块)与 B_{v_w} 之间的非分支块 B_u,$u\in\{b_1,b_2,\cdots,b_n\}$,$1\leqslant w\leqslant r$,若 ins_l 是 B_{v_w} 或 B_u 中的加锁指令,且其相应解锁指令在 $B_{v_{s+1}}$ 或者 B_{v_s} 与 $B_{v_{s+1}}$ 之间的非分支块中,则插桩在 ins_l 前后的 $push$ 和 pop 语句集各自位于条件语句 if(C_{v_w} && $C_{v_{w+1}}$ && … && C_{v_s})中.若 ins_f 是 B_{v_w} 或 B_u 中的自定义函数调用指令,则插桩在 ins_f 前后的 $push$ 和 pop 语句集各自位于条件语句 if(C_{v_w} && $C_{v_{w+1}}$ && … && C_{v_r})中.

图 9 展示了使用上述规则对分支结构函数的锁效应插桩结果.图 9 控制流图上存在 2 条路径 $path_1:B_5\rightarrow B_4\rightarrow B_3\rightarrow B_1\rightarrow B_0$ 和 $path_2:B_5\rightarrow B_4\rightarrow B_2\rightarrow B_1\rightarrow B_0$,$B_4$ 是分支块.对 $path_1$ 和 $path_2$ 中 B_4 的加锁指令 $lock(mtx_1)$ 运用规则 5 插桩锁效应,对 B_3 和 B_2 运用规则 2 插桩锁效应,得到插桩后基本块.

插桩规则 3～5 对于在分支块或者在位于分支块之前的非分支块中的加锁操作和自定义函数调用操作进行锁效应插桩时,会将相应分支条件提前计算,以保证插桩的锁效应是针对当前路径的.但是有时分支条件不能提前计算,如分支条件中包含加锁

操作,分支条件中使用的变量在加锁操作或自定义函数调用操作之后被定义或修改(图 10).在这种情况下,Flider 使用路径不敏感分析方法插桩锁效应,即将从加锁操作或自定义函数调用操作到函数结束的所有语句视为一条路径上的语句,使用针对不含有分支块的路径的锁效应插桩方法插桩.这种方法可能会串行化一些不必要串行的关键区,也可能会引入活锁.

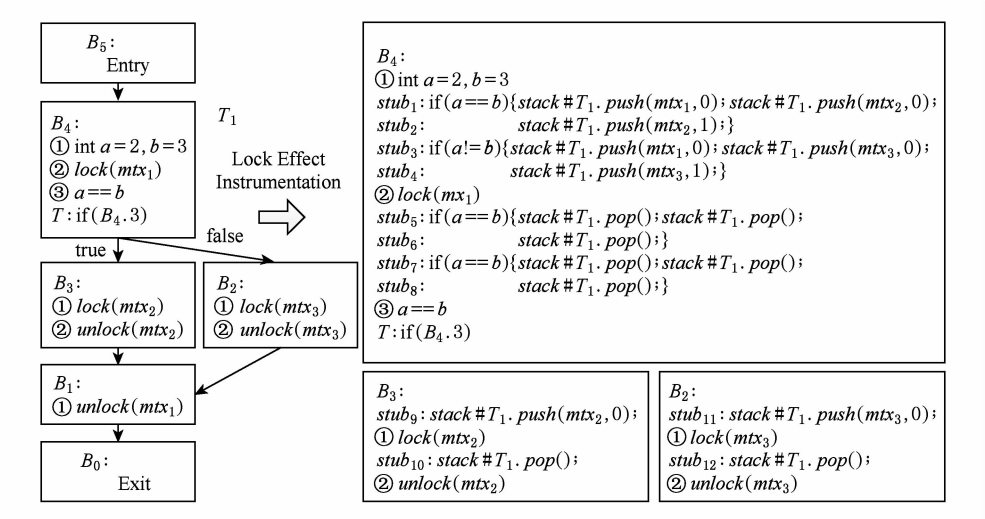


Fig. 9 Lock effect instrumentation for functions with conditional structures

图 9 分支结构函数的锁效应插桩

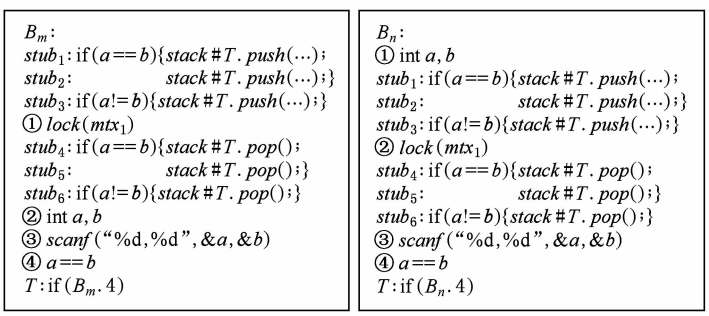


Fig. 10 Examples of buggy instrumentation

图 10 错误插桩的例子

对于循环结构,由于其已在控制流上转化成分支结构,因此对循环结构的锁效应插桩自动按照对分支结构的插桩方法处理.

插桩完成后,Flider 利用 Clang 的代码重写器读取已插桩的控制流图,生成插桩过锁效应的源码.

4.4 加锁解锁操作劫持

我们将 Flider 的动态分析模块实现为一个动态链接库.在插桩过锁效应信息的目标程序执行时,该动态库以预加载机制作为第一个被加载的动态库加载到目标程序的进程空间内.通过使用 Linux 提供的预加载机制 LD_PRELOAD,Flider 劫持表 1 中所有作用于互斥锁和读写锁的加锁解锁操作,以便构建和更新全局混合锁分配图 G(定义 1)、为加锁操作计算未来锁集(定义 9)、根据未来锁集执行规避逻辑(定理 1).

混合锁分配图根据加锁解锁操作的执行效果而实时更新.若线程 T 执行加锁操作 $lock(x)$ 请求加锁 x 但目前还没有成功占据 x,则建立从 T 到 x 的(读/写)请求关系;若线程 T 执行加锁操作 $lock(x)$ 成功,则删除从 T 到 x 的(读/写)请求关系并建立从 x 到 T 的(读/写)占据关系;若线程 T 执行解锁操作 $unlock(x)$ 成功,则删除从 x 到 T 的(读/写)占据关系.在 Flider 中,混合锁分配图以映射数组的形式实现.

4.5 规避逻辑的原子执行

由于有锁效应信息的辅助,加锁操作的未来锁集计算直接按照定义 9 进行即可.而规避逻辑检查未来锁集中的锁的状态,只有当所有锁都未被占据

Table 1　Lock/Unlock Operations on Mutual Exclusive Locks and Read/Write Locks

表 1　读写锁和互斥锁加锁解锁操作

Lock Type	Lock/Unlock Operations
pthread_mutex_t	int *pthread_mutex_lock*(pthread_mutex_t*); int *pthread_mutex_trylock*(pthread_mutex_t*); int *pthread_mutex_timedlock*(pthread_mutex_t*,const struct timespec*); int *pthread_mutex_unlock*(pthread_mutex_t*).
pthread_rwlock_t	int *pthread_rwlock_rdlock*(pthread_rwlock_t*); int *pthread_rwlock_tryrdlock*(pthread_rwlock_t*); int *pthread_rwlock_wrlock*(pthread_rwlock_t*) int *pthread_rwlock_trywrlock*(pthread_rwlock_t*); int *pthread_rwlock_timedrdlock*(pthread_rwlock_t*,const struct timespec*); int *pthread_rwlock_timedwrlock*(pthread_rwlock_t*,const struct timespec*); int *pthread_rwlock_unlock*(pthread_rwlock_t*).

时,才允许当前加锁操作执行.其中 2 个动作,即检查锁状态和决定加锁操作是否执行,必须原子性执行,否则规避逻辑不能正确运行.假设线程 T_1 在执行加锁操作 $lock(x)$前,Flider 计算出该加锁操作的未来锁集是$\{x,y\}$,然后 Flider 检查发现 x 和 y 都未被占据,在它进一步决定允许 T_1 执行 $lock(x)$时,操作系统调度 T_2 执行 $lock(y)$,假设 $lock(y)$的未来锁集也是$\{x,y\}$,则 T_2 会被运行执行 $lock(y)$并成功获取锁 y.当 T_1 再次执行时,它仍然认为 x 和 y 都是空闲的,从而执行 $lock(x)$以获取 x.这种规避逻辑可能不能规避死锁.因此为保证正确性,规避逻辑必须原子性执行.为此,我们为所有的规避逻辑加上全局同一锁保护,使得所有的规避逻辑串行执行.具体地,锁状态检查在全局锁的保护下进行,如果发现未来锁集中的锁不是全都空闲,则释放全局锁,等待一段时间,然后重新获取全局锁,检查锁状态.若在全局锁的保护下,发现未来锁集中的锁都未被占据,则执行加锁操作,然后释放全局锁.这样就保证了规避逻辑的正确性.此外,为规避互斥锁自锁和读写锁自锁,当 Flider 检查发现未来锁集中的锁已被当前线程占据时,就将当前加锁操作和后续执行中的相应解锁操作视为空操作.

5　实验与分析

本节使用表 2 和表 3 列出的 2 个基准程序集和引言部分介绍的 Dimmunix[7],Grace[10],Flock[11],Scrider[12]这 4 个具有代表性的死锁规避工具,实验

Table 2　The Deadlock Benchmark Composed of Ten May-Deadlock Programs

表 2　由 10 个死锁程序构成的死锁基准程序集

Program	LOC(K)	Bug ID	Deadlock Type	Deadlock Description
Deadlock-Mutex	0.1	Bug#1	Mutex Deadlock	Two threads acquire four mutexes in different order
		Bug#2	Mutex Self-Deadlock	One thread acquires a mutex without first releasing it
Deadlock-Mrwlock	0.1	Bug#3	Mixed Deadlock	Two threads acquire two rwlocks and two mutexes in different order
		Bug#4	Rwlock Deadlock	Two threads acquire two rwlocks in different order
		Bug#5	Rwlock Self-Deadlock	One thread acquires a rwlock without first releasing it
Bank-Transaction	0.1	Bug#6	Mutex Deadlock	Deposit and withdraw are separated and protected by different mutexes
Dining-Philosophers	0.1	Bug#7	Mutex Deadlock	Each philosopher first picks up the left stick,then the right stick
Tgrep	1.7	Bug#8	Rwlock Deadlock	Reverse lock acquisition on *work_q_lk* and *running_lk*
Sshfs-fuse-2.2	3.8	Bug#9	Mixed Deadlock	Reverse lock acquisition on a mutex and a rwlock
SQLite-3.3.3	50.7	Bug#10	Mutex Deadlock	*sqlite3UnixEnterMutex*() and *sqlite3UnixLeaveMutex*()
HawkNL-1.6b3	8.7	Bug#11	Mutex Deadlock	*nlshutdown*() and *nlclose*()
Openldap-2.2.20-kernel	0.2	Bug#12	Mixed Deadlock	*bdb_cache_add*() and *bdb_cache_find_id*()
MySQL-6.0.4-kernel	0.2	Bug#13	Rwlock Deadlock	Two threads perform insert and truncate operations on the same table with the falcon engine

Table 3 The Deadlock-Free Benchmark Composed of Four Kernel Programs from SPLASH-2

表 3 由 4 个 SPLASH-2 核心程序组成的基准非死锁程序集

Program	Parameter Setup	Program Description
LU(contiguous)	-n6144-p1/2/4/8/16/32-b16	Matrix decomposition using Doolittle algorithm
FFT	-p1/2/4/8/16/32-m24-n65536-14	Fast Fourier transformation
RADIX	-p1/2/4/8/16/32-n67108864-r8-m524288	Radix sort for integer numbers
CHOLESKY	-p1/2/4/8/16/32-b32-c16384./inputs/tk17.O	Matrix decomposition using Cholesky algorithm

回答以下问题.

问题 1. Flider 的规避能力如何,即能否主动地规避多种类型的死锁.

问题 2. Flider 的规避效率如何,即能否高效智能地规避死锁.

问题 3. Flider 的可扩展性如何,即能否在线程数目指数级增长情况下,其规避开销保持平稳增长.

5.1 死锁基准程序集和非死锁基准程序集

为回答问题 1 和 2,本节使用的死锁基准程序集由作者编写的 2 个含有多个不同类型死锁的程序与在死锁检测和死锁规避领域[7,11,14]广泛使用的 8 个死锁程序构成,如表 2 所示. 由于死锁缺陷在正常情况下难以暴露,我们在这 10 个程序的关键代码点插入 *sleep*()语句,以影响程序的线程调度和执行路径,使得死锁以几乎 100%的概率被触发.

表 2 列出死锁程序的规模、死锁程序中包含的死锁缺陷、死锁缺陷的具体类型和死锁缺陷的发生原因或触发场景. Deadlock-Mutex 和 Deadlock-Mrwlock 是作者自己编写的死锁程序,分别包含 2 个和 3 个不同种类的死锁缺陷;Bank-Transaction, SQLite-3. 3. 3, Dining-Philosophers, HawkNL-1. 6b3 各自包含一个互斥锁死锁缺陷;Tgrep 和 MySQL-6. 0. 4-kernel 则各自包含一个读写锁死锁缺陷;Openldap-2. 2. 20-kernel 和 Sshfs-fuse-2. 2 各自包含一个混合死锁缺陷. 我们使用 Bug#12 和 Bug#13 模拟真实死锁缺陷 MySQL#37080① 和 OpenLDAP#3494②,故包含这 2 个缺陷的程序 MySQL-6. 0. 4-kernel 和 Openldap-2. 2. 20-kernel 程序规模较小.

为回答问题 2 和 3,本节使用的非死锁基准程序集由 4 个 SPLASH-2③ 核心程序构成. 表 3 给出各个程序的简要描述和在实验中使用的参数设置.

5.2 实验环境与死锁规避工具

所有评测和对比实验在下列实验环境下进行:Intel Core i5 M430 2. 27 GHz CPU, 6 GB 内存, Ubuntu 14. 04 32 位操作系统,Clang-3. 6 和 Gcc-4. 9. 1 编译器.

为评价 Flider 的死锁规避能力、死锁规避效率和可扩展性,使用表 4 中死锁规避工具监视表 2 和表 3 中程序运行. 之所以选用 Dimmunix, Grace, Flock, Slider 这 4 个死锁规避工具,是因为它们代表了死锁规避领域的最新水平,且源码开放,能在我们的实验环境下编译运行. Sammati[9] 源码开放,但只能在 64 位操作系统下编译运行,我们的实验环境不支持. 虽然没有使用 Sammati 做对比实验,但根据文献[9]可知,Sammati 只能规避互斥锁死锁.

Table 4 Deadlock Avoidance Tools Used in Our Experiments

表 4 实验中使用的死锁规避工具

Tool	Author	Year	Key Technique
Flider	Yu	2017	Future Lockset
Scrider	Yu	2014	Critical Section Serialization
Flock	Gerakios	2011	Future Lockset
Grace	Berger	2009	Rollback/Re-execute
Dimmunix	Jula	2008	Deadlock Signature

5.3 问题 1:Flider 死锁规避能力

为评价 Flider 的死锁规避能力,按如下步骤进行实验:1)分别在 Flider, Scrider, Dimmunix, Grace, Flock 下运行表 2 中的各个死锁程序,每个死锁程序在每个规避工具下运行 30 次;2)针对一个死锁缺陷,如果某个工具在 30 次运行中只要有一次没有成功规避该缺陷,则认为该工具不能规避该缺陷;3)统计每个工具对每个死锁缺陷的规避结果.

实验结果如表 5 所示,其中√表示规避成功,×表示规避失败,*表示规避成功但输出错误. 针对所

① http://bugs. mysql. com/bug. php?id=37080

② http://www. openldap. org/lists/openldap-bugs/200501/msg00101. html

③ http://www. capsl. udel. edu/splash/, http://kbarr. net/splash2

有 13 个不同类型的死锁缺陷,Flider,Scrider,Grace 各自成功规避其中 11 个,Flock 成功规避 5 个,Dimmunix 只成功规避了 4 个.

Table 5　Deadlock Avoiding Results of Five Avoidance Tools on The Deadlock Benchmark

表 5　5 个死锁规避工具在死锁基准程序集上的规避结果

Bug ID	Flider	Scrider	Dimmunix	Grace	Flock
Bug#1	√	√	√	*	√
Bug#2	√	×	×	√	×
Bug#3	√	√	×	*	×
Bug#4	√	√	×	*	×
Bug#5	√	×	×	√	×
Bug#6	√	√	√	*	√
Bug#7	√	√	×	×	√
Bug#8	√	√	×	×	×
Bug#9	√	√	×	√	×
Bug#10	√	√	√	*	√
Bug#11	×	√	√	*	√
Bug#12	√	√	×	*	×
Bug#13	×	√	×	*	×

Note: √,Succeed; ×,Fail; *,Wrongly Output

各个死锁规避工具的规避结果基本符合预期. Scrider 成功规避了除互斥锁自锁(Bug#2)和读写锁自锁(Bug#5)的所有类型死锁. Dimmunix 和 Flock 只能规避互斥锁死锁(Bug#1,Bug#6,Bug#7,Bug#10,Bug#11),其他类型死锁都不能规避. 但令人惊奇的是,与文献[7]相悖,Dimmunix 没有能够规避互斥锁死锁缺陷 Bug#7. Bug#7 是程序 Dining-Philosophers 中的一个简单互斥锁死锁,当 5 个哲学家不停地进餐/思考,并在进餐时每个哲学家(用线程模拟)都先拿到左手边筷子(用互斥锁模拟)再去拿右手边筷子时,Bug#7 就会发生. Bug#7 涉及 5 个线程和 5 个互斥锁,每个线程都占据一个互斥锁并请求另一个线程占据的互斥锁. 我们仔细检查了 Dimmunix 的源码,发现其用于存储加锁解锁事件的无锁队列实现错误,这一错误可能导致某些加锁解锁事件丢失,从而 Dimmunix 不能察觉到构成死锁的条件正在一个个发生,更不能控制线程调度以规避将要发生的死锁.

Grace 使用回滚/重新执行技术虽能成功规避 11 个死锁缺陷,但在规避其中 8 个死锁时,由于不能回滚 IO 操作,导致包含死锁的程序输出发生错误,影响了程序正确性. 此外,Grace 不能规避 Bug#7 和 Bug#8. 对于 Bug#7,原因可能是由于参与到 Bug#7 中的 5 个线程频繁读写共享变量,Grace 未能成功隔离对共享变量的修改而导致回滚时使程序进入到一个非法状态;对于 Bug#8,原因是由于参与到 Bug#8 中的线程使用条件变量进行同步,而 Grace 不能处理条件变量操作.

Flider 成功规避了除 Bug#11 和 Bug#13 外的所有死锁,其规避能力与 Scrider 相当. Flider 不能规避这 2 个缺陷是因为不精确的过程内锁效应计算和插桩会导致 Flider"看不到"某些本来应该属于未来锁集中的锁. 例如,Bug#11 和 Bug#13 可以抽象为如图 11 所示的代码,由于静态阶段没有进行过程间分析,Flider 为 $f_1()$的 $lock(x)$计算出的锁效应是$\{x-\}$,实际上,线程 T_1 在执行 $lock(x)$后还会调用 $g_1()$进而执行 $lock(y)$. 对于 $f_2()$中的 $lock(y)$也存在这样的情况. 由于锁效应信息不准确,Flider 为加锁操作计算出的未来锁集也不准确. 当 T_1 执行 $lock(x)$时,Flider 为其计算的未来锁集是$\{x\}$,x 此时未被占用,故 Flider 允许 T_1 执行 $lock(x)$并占据 x. 当 T_2 执行 $lock(y)$时,Flider 也允许其执行并占据 y. 这样当程序继续执行时,死锁必定会发生,而且不能被 Flider 规避掉.

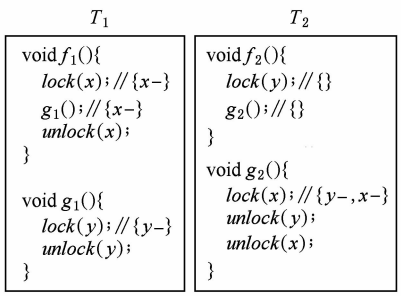


Fig. 11　A deadlock example that Flider fails to avoid

图 11　一个 Flider 不能规避的死锁例子

虽然存在以上不足,但从死锁规避数量、死锁规避类型方面评价,Flider 的死锁规避能力在 5 个规避工具中是最高的. Flider 能规避所有 5 种类型的死锁缺陷,而 Scrider 和 Flock 分别只能规避 3 种和 1 种类型的死锁缺陷. Flider 主动规避死锁,在 Flider 的规避逻辑下,死锁不可能发生,而 Dimmunix 则只有在死锁发生后才能规避死锁. Flider 允许执行的每个加锁操作都是安全的,因此无需回滚程序执行,也就不会造成输出错误. 只要目标程序不包含如图 11 所示的死锁缺陷,则 Flider 不会对目标程序正确性造成影响.

5.4 问题 2:Flider 死锁规避效率

为评价 Flider 的死锁规避效率,对比分析多个规避工具在死锁程序集和非死锁程序集上的规避开销.

1) Flider 和 Flock 在死锁程序集上的静态分析效率对比

由于 Flider 和 Flock 在进行动态规避之前,需要先进行源码分析以计算锁效应信息和插桩代码将锁效应信息传递给动态规避逻辑,因此在对比分析规避工具的动态规避开销之前,使用 Flider 和 Flock 对死锁程序集中的每个程序分别进行 10 次静态分析(即锁效应计算和代码插桩),统计得到平均分析时间,如表 6 所示:

Table 6 Comparison in Static Analysis Efficiency of Flider and Flock on Ten Deadlock Programs

表 6 Flider 和 Flock 在死锁程序集上静态分析效率 s

Program	Flider	Flock
Deadlock-Mutex	0.052 4	0.052 8
Deadlock-Mrwlock	0.050 8	0.049 2
Bank-Transaction	0.052 4	0.052 0
Dining-Philosophers	0.134 8	0.131 2
Tgrep	505.838 7	481.505 7
Sshfs-fuse-2.2	9.071 0	8.849 8
SQLite-3.3.3	0.036 0	0.036 0
HawkNL-1.6b3	0.056 0	0.053 6
Openldap-2.2.20-kernel	0.064 4	0.064 4
MySQL-6.0.4-kernel	0.040 8	0.036 0

在表 6 中,由于只对 SQLite-3.3.3 和 HawkNL-1.6b3 中涉及到死锁的文件进行静态分析,因此 Flider 和 Flock 对这 2 个程序很快完成了锁效应计算和插桩. Flider 和 Flock 对 Tgrep 和 Sshfs-fuse-2.2 的静态分析耗费时间较长,是由于其中存在具有大量条件分支语句的函数,而路径敏感的算法 1 在这种情况下会逐条遍历每条路径,导致静态分析速度变慢.另外,从表 6 中还可以看出,对于只含有互斥锁死锁的程序,如 SQLite-3.3.3 等,Flider 和 Flock 的静态分析时间相同,而对于含有读写锁死锁或者混合死锁的程序,如 Tgrep,Flider 的分析时间一般较 Flock 长,因为针对这些程序,Flider 不仅要为互斥锁加锁操作计算锁效应和插桩,也要为读写锁加锁操作进行锁效应分析.根据表 6,只要并发程序中的函数不具有过分巨大的条件分支,如一个函数中含有 50 个以上的分支条件,则 Flider 会在合理时间内完成静态分析.

2) 多个规避工具在死锁程序集上的规避效率

统计 Flider,Scrider,Grace 在规避 Bug#1,Bug#3,Bug#4,Bug#6,Bug#9,Bug#10,Bug#12 时,包含这些死锁缺陷的程序 Deadlock-Mutex,Deadlock-Mrwlock,Bank-Transaction,Sshfs-fuse-2.2,SQLite-3.3.3,Openldap-2.2.20-kernel 的运行时间.我们没有选用 Dimmunix 和 Flock 是因为 Dimmunix 不能在线规避死锁,规避效率较差,且 Dimmunix 和 Flock 规避死锁数量和类型都较少.另外,我们修改 Deadlock-Mutex 以保证其执行时 Bug#2 不会发生,对 Deadlock-Mrwlock 也做了类似修改.

用箱式图统计分析各个工具在各个程序上的规避时间,如图 12 所示.对所有 6 个程序中的 4 个,即 Deadlock-Mrwlock,Deadlock-Mutex,SQLite-3.3.3,Openldap-2.2.20-kernel,Flider 的规避时间都比 Scrider 小,其中对 Deadlock-Mrwlock,Scrider 的平均规避时间甚至达到了 Flider 的 53 倍.Grace 的规避开销较小,对所有 6 个程序中的 3 个,即 SQLite-3.3.3,Deadlock-Mutex,Openldap-2.2.20-kernel,Grace 的规避时间是 3 个工具中最小的,但 Grace 在规避过程中造成 6 个程序中的 5 个,即 Deadlock-Mutex,Deadlock-Mrwlock,SQLite-3.3.3,Bank-Transaction,Openldap-2.2.20-kernel,输出错误.

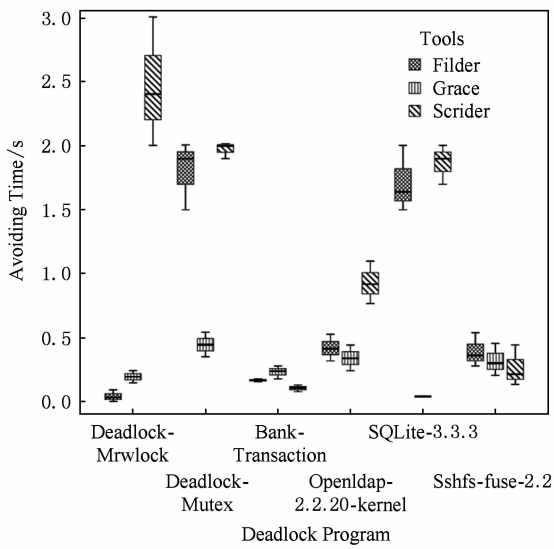


Fig. 12 Comparison in avoiding efficiency of three tools on six deadlock programs

图 12 3 个规避工具在 6 个死锁程序上的规避效率对比

另外,对所有 6 个程序中的 4 个,即 Deadlock-Mrwlock,Bank-Transaction,Sshfs-fuse-2.2,Openldap-2.2.20-kernel,Flider 的规避开销与 Grace 相近,

平均约是 Grace 的 0.85 倍.图 12 还直观地反映出 3 个工具的稳定性:Grace 的稳定性较好,Flider 稳定性一般,Scrider 稳定性较差.

根据表 5 可知,Flider 和 Flock 都能规避的死锁缺陷是 Bug#1,Bug#6,Bug#7,Bug#10.我们对比 Flider 和 Flock 对包含这些死锁缺陷的并发程序的规避开销,如表 7 所示.从表 7 中可以看出,Flider 和 Flock 对这 4 个死锁缺陷的平均、最大和最小规避开销在数值上几乎相同,而在百分比上平均规避开销差异在−9.1%～6%之间,最大规避开销差异在 0～12.5%之间,最小规避开销在 0.2%～14.3%之间.Flider 和 Flock 对这 4 个死锁缺陷的规避开销相差很小,是因为这 4 个死锁都是互斥锁死锁,而且包含这 4 个死锁缺陷的并发程序只调用互斥锁函数进行加锁解锁操作,这就使得 Flider 和 Flock 进行的静态分析(包括锁效应计算和代码插桩)完全一样,进而导致并发程序执行的规避逻辑完全一样.这样,Flider 和 Flock 对每个死锁缺陷的规避开销就几乎相同.

Table 7　Comparison in Avoiding Efficiency of Flider and Flock on Four Deadlock Programs

表 7　Flider 和 Flock 在 4 个死锁程序上规避效率对比　s

Program	Flider			Flock		
	avg.	max.	min.	avg.	max.	min.
Deadlock-Mutex	2.001	2.004	2.000	2.001	2.008	2.000
Bank-Transaction	0.130	0.160	0.096	0.123	0.168	0.084
SQLite-3.3.3	3.000	3.004	2.996	2.999	3.004	2.992
Dining-Philosophers	0.017	0.040	0.012	0.019	0.048	0.012

3) 多个规避工具在非死锁程序集上的规避效率

实验方法:①分别在 Flider 和 Scrider 下运行表 3 中的 4 个非死锁程序各 30 次,每个程序的线程数目参数设置为 2;②记录每个程序的运行时间并求平均值.由于 4 个非死锁程序存在使用共享变量进行线程间通信的情况,用 Grace 对它们进行规避可能导致活锁,因此本实验不选用 Grace.实验结果如图 13 所示.

Flider 分别对 LU,FFT,RADIX,CHOLESKY 造成 3.6%,−0.1%,3.0%,21%的性能下降,平均规避开销是 6.9%;而 Scrider 分别对上述 4 个程序造成 20.5%,178.7%,25.7%,100.5%的性能下降,平均规避开销是 81.4%.相比来说,Flider 对目标程序的性能影响较微小,而 Scrider 几乎使目标程序的运行速度变慢为原来的 1/2.这 2 个工具都采用

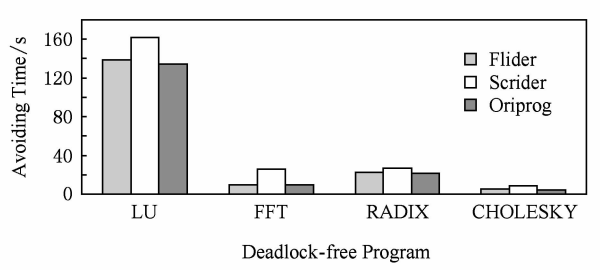


Fig. 13　Comparison in avoiding efficiency of two tools on four deadlock-free programs

图 13　2 个规避工具在 4 个非死锁程序上的规避效率对比

串行关键区的方法来规避死锁,但与 Flider 不同,由于没有锁效应信息的辅助,Scrider 串行所有的关键区,不论关键区之间是否可能造成死锁,这种盲目的方法使得 Scrider 的规避开销急剧上升.Flider 只串行可能导致死锁的关键区,而不影响绝大部分关键区的并行执行,这种智能的方法大大降低了其规避开销.

综上,Flider 对死锁程序集和非死锁程序集的规避开销都较低,且能高效智能地规避死锁程序集中的死锁缺陷.

5.5　问题 3:Flider 死锁规避可扩展性

为评价 Flider 的死锁规避可扩展性,按如下步骤进行实验:1)在线程数目参数设置为 1,2,4,8,16,32 的情况下,分别在 Flider 和 Scrider 下运行表 3 中的各个非死锁程序 30 次;2)记录并计算每个程序在每个工具下对每个线程数目参数设置的平均运行时间.

实验结果如图 14 所示.对 LU,当线程数目从 1 增加到 32 时,Flider 的开销在 4.6%～9.7%之间,而 Scrider 的开销则在 13.0%～38.4%之间.对 FFT,当线程数目从 2 增加到 16 时,Flider 的开销从 1.2%增加到 4.8%,而 Scrider 的开销从 50%迅速增加到 195%.对 CHOLESKY,当线程数目从 1 增加到 32 时,Flider 的开销在 16%～85.3%之间,而 Scrider 的开销在 100.5%～802.0%之间,虽然两者的开销变化都较大,但在特定的线程数目时,Flider 对目标程序的开销相对 Scrider 很小.对 RADIX,Flider 和 Scrider 的规避开销与对其他 3 个程序的开销类似.

因此,从图 14 中 4 个折线图可以看出,总体上,随着线程数目指数增长,Flider 的规避开销总体保持平稳增长;而 Scrider 对线程数目的变化较敏感,其规避开销随线程数目的增多而迅速增加.

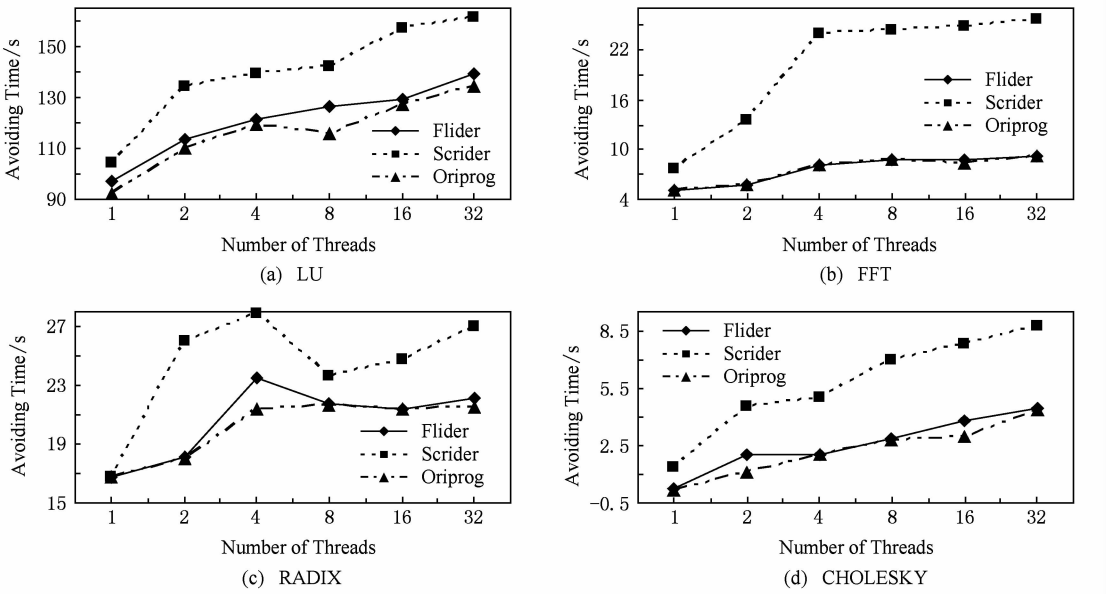


Fig. 14 Comparison in avoiding efficiency of two tools on four deadlock-free programs

图 14 2 个规避工具在 4 个非死锁程序上的规避效率对比

6 结 论

现有动态规避方法存在能力有限、被动盲目、开销较大和影响目标程序正确性等问题,为解决这些问题,本文提出一种动静结合的基于未来锁集的死锁规避方案 Flider. 首先形式化描述该方案并在理论上证明该方案的正确性;然后从 3 个方面,即过程内控制流图建立、路径敏感的锁效应计算与插桩和未来锁集计算与规避逻辑执行,来具体介绍该方案的实现细节;最后进行测评和对比实验,回答以下 3 个研究问题:

1) 与问题 1 相关的实验中,Flider 主动规避了所有 13 个死锁缺陷中的 11 个(这 11 个死锁缺陷分别属于 5 种不同类型的死锁),且在规避所有 11 个死锁缺陷时,没有对目标程序的正确性造成影响. 实验结果表明 Flider 能主动规避多种类型的死锁,且不会影响目标程序的正确性.

2) 与问题 2 相关的实验中,Flider 借助于静态时插桩的锁效应信息,智能地只串行化可能发生死锁的关键区,在执行规避逻辑时对目标程序造成的影响小. 实验结果表明 Flider 能智能高效地规避死锁.

3) 与问题 3 相关的实验中,Flider 在目标程序线程数目从 1 指数级增长到 32 时,规避开销的增长相对于线程数目的增长较平稳. 实验结果表明 Flider 的可扩展性良好.

综上所述,本文提出的死锁规避方案达到了"智能、主动、高效地规避多种类型死锁,不影响程序正确性"的研究目标.

参 考 文 献

[1] Agarwal R, Stoller S D. Runtime detection of potential deadlocks for programs with locks, semaphores and conditional variables [C] //Proc of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging. New York: ACM, 2006: 51-60

[2] Wang Zhaofei, Huang Chun. Static detection of deadlocks in OpenMP Fortran programs [J]. Journal of Computer Research and Development, 2007, 44(3): 536-543 (in Chinese)
(王昭飞,黄春. OpenMP Fortran 程序中死锁的静态检测 [J]. 计算机研究与发展, 2007, 44(3): 536-543)

[3] Shimomura T, Ikeda K. Two types of deadlock detection: Cyclic and acyclic [J]. Intelligent Systems for Science and Information, 2014, 54(2): 233-259

[4] Fonseca P, Li C, Singhal V, et al. A study of the internal and external effects of concurrency bugs [C] //Proc of the 2010 IEEE/IFIP Int Conf on Dependable Systems and Networks (DSN). Piscataway, NJ: IEEE, 2010: 221-230

[5] Lu S, Park S, Seo E, et al. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics [J]. ACM SIGARCH Computer Architecture News, 2008, 36(1): 329-339

[6] Su Xiaohong, Yu Zhen, Wang Tiantian, et al. A survey on exposing, detecting and avoiding concurrency bugs [J]. Chinese Journal of Computers, 2015, 38(11): 2215-2233 (in Chinese)

(苏小红,禹振,王甜甜,等. 并发缺陷暴露、检测与规避研究综述[J]. 计算机学报,2015,38(11):2215-2233)

[7] Jula H, Tralamazza D M, Zamfir C, et al. Deadlock immunity: Enabling systems to defend against deadlocks [C] //Proc of the 8th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX, 2008: 295-308

[8] Nir-Buchbinder Y, Tzoref R, Ur S. Deadlocks: From exhibiting to healing [C] //Proc of the 8th Int Conf on Runtime Verification. Berlin: Springer, 2008: 104-118

[9] Pyla H K, Varadarajan S. Avoiding deadlock avoidance [C] //Proc of the 19th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2010: 75-86

[10] Berger E D, Yang T, Liu T, et al. Grace: Safe multithreaded programming for C/C++ [J]. ACM SIGPLAN Notices, 2009, 44(10): 81-96

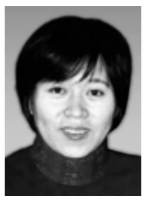
[11] Gerakios P, Papaspyrou N, Sagonas K. A type and effect system for deadlock avoidance in low-level languages [C] // Proc of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation. New York: ACM, 2011: 15-28

[12] Yu Zhen, Su Xiaohong, Ma Peijun. Scrider: Using single critical sections to avoid deadlocks [C] //Proc of the 4th IEEE Int Conf on Instrumentation and Measurement, Computer, Communication and Control. Piscataway, NJ: IEEE, 2014: 1000-1005

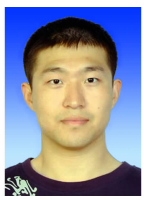
[13] Necula G C, McPeak S, Rahul S P, et al. CIL: Intermediate language and tools for analysis and transformation of C programs [C] //Proc of the 11th Int Conf on Compiler Construction. Berlin: Springer, 2002: 213-228

[14] Wang Y, Kelly T, Kudlur M, et al. Gadara: Dynamic deadlock avoidance for multithreaded programs [C] //Proc of the 8th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX, 2008: 281-294

Yu Zhen, born in 1987. PhD candidate in the School of Computer Sience and Technology at Harbin Institute of Technology (HIT). Student member of CCF. His main research interests include software static or dynamic analysis, implicit rules mining from large-scale software and concurrency bug (including deadlock, data race and atomicity violation) detection and avoidance.



Su Xiaohong, born in 1966. Professor and PhD supervisor in the School of Computer Science and Technology at HIT. Senior member of CCF. Her main research interests include software engineering, information fusion, image processing and computer graphics.



Qi Peng, born in 1990. Master in the School of Computer Science and Technology at HIT. His main research interests include software engineering and deadlock bug detection/avoidance (qipengsemail@163.com).

Ma Peijun, born in 1963. Professor and PhD supervisor in the School of Computer Science and Technology at HIT. His main research interests include software engineering, information fusion, image processing, embedded system simulation, and so on (ma@hit.edu.cn).