计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2017.20160369
54(7):1557-1568,2017

使用锁分配图动态检测混合死锁

禹 振 苏小红 邱 景
(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)

Dynamically Detecting Multiple Types of Deadlocks Using Lock Allocation Graphs

Yu Zhen, Su Xiaohong, and Qiu Jing
(*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001)

Abstract Deadlock bugs are hard to expose, reproduce and diagnose. Once happening, they will cause increasing response time, decreasing throughput, or even crash to the target multithreaded programs. However, current deadlock detection techniques can detect only one mutex-caused deadlock at a time. In order to detect all possible deadlocks one time caused by multiple threads and multiple mutexes or rwlocks, this paper proposes the concept of multiple-type lock allocation graph (MLAG) and its construction method. Then a MLAG-based dynamic detection algorithm to detect multiple types of deadlocks is proposed. By instrumenting all lock/unlock operations on mutexes and rwlocks, our method dynamically constructs and real-time updates a MLAG which reflects the synchronization state of the target program. Our method detects deadlock bugs by detecting cycles on the MALG and checking whether or not a cycle is a deadlock cycle. When a deadlock is detected, the method outputs information about that bug to assist debugging. The experimental results on benchmarks show that our method is strong in deadlock detection for successfully detecting all 13 deadlock bugs with 5 types, and has a slight impact on target programs for incurring at most 10.15% slowdown to openldap-2.2.20's performance, and is scalable because the overhead increase gently along with an exponentially increasing number of threads.

Key words dynamic analysis; software testing; concurrency bug detection; deadlock detection; cycle detection

摘 要 死锁难以暴露、重演和调试.一旦发生,将导致多线程程序响应时间增长、吞吐量下降甚至宕机崩溃.现有死锁检测技术每次只能检测一个互斥锁死锁.为一次性检测由多个线程和多个互斥锁或读写锁造成的所有类型死锁,首先提出混合锁分配图的概念和构建方法,然后提出一种利用混合锁分配图动态检测混合死锁的方法.通过劫持所有互斥锁和读写锁的加锁解锁操作,以动态构建和实时更新一个反映目标程序同步状态的混合锁分配图.通过在锁分配图上检测环并判定该环是否为死锁环来检测死锁.当检测到死锁时,输出死锁信息来辅助调试.死锁检测实验、性能影响实验和可扩展性实验结果表明:该方法成功检测出所有13个共5种类型的死锁缺陷,检测能力强;给openldap-2.2.20带来至多10.15%的性能下降幅度,对目标程序造成的性能影响较小;性能开销随线程数目指数级增大而平缓增长,扩展性良好.

收稿日期:2016-05-25;修回日期:2017-02-06
基金项目:国家自然科学基金项目(61173021,61672191)
This work was supported by the National Natural Science Foundation of China (61173021, 61672191).
通信作者:苏小红(sxh@hit.edu.cn)

关键词 动态分析;软件测试;并发缺陷检测;死锁检测;环检测

中图法分类号 TP311

多线程程序中,一个线程集合中的每一个线程都在各自等待另一个线程占据的互斥性资源,由此导致的循环等待即为死锁.在所有并发缺陷中[1],死锁是其中最常见和最重要的一种:多方统计调查显示[2-4],死锁缺陷占所有并发缺陷的30%以上.死锁一旦发生,将可能导致多线程程序响应时间增长、吞吐量下降甚至宕机崩溃,严重威胁并发程序的可用性与可靠性.与其他并发缺陷一样,死锁具有难以暴露、重演和调试的特点[1],因此死锁动态检测技术应在死锁缺陷发生时立即将其检测出来,并同时输出关于缺陷的相关信息以辅助调试.

现有的死锁检测技术可以分为静态分析和动态分析2大类,其中静态分析又可以分为类型与效应分析和数据流分析2类,而动态分析则可以再分为在线和离线2类,如表1所示:

Table 1 Categories and Summaries of Deadlock Detection Techniques

表1 死锁检测技术分类与概述

Analysis Category	Technique Category	Summary
Static Analysis	Type and Effect Analysis[5-6]	Checking whether or not the target program requests locks according to a global locking order. If not, reporting that there may be potential deadlocks.
	Data Flow Analysis[7-9]	Combining multiple static analysis techniques to statically compute a lock-order graph for the target program and reporting cycls on the graph as potential deadlocks.
Dynamic Analysis	Online[10-17]	Monitoring the target program's run, creating realtime abstract representation for synchronization status of the target program and detecting cycles on the representation.
	Offline[18-21]	Obtaining and analyzing execution traces of the target program to create lock-order graphs and reporting cycles in the graphs as potential deadlocks.

类型与效应分析需要用户在源码中为变量或函数添加类型注释才能对程序进行类型推理和类型检查,故其人工干预度高、扩展性较差.另外该类技术没有考虑读写锁的语义,只能检查程序是否是免于互斥锁死锁的和检测是否存在互斥锁死锁.

数据流分析直接分析程序源码,综合使用调用图分析、上下文分析、指向分析和逃逸分析等静态分析技术,计算静态锁占用约束或锁占用顺序图,使用约束求解和环检测算法在其上检测环,将环作为可能的死锁报告出来.数据流分析缺乏精确的运行时信息,一般对变量值作保守估计,因此其误检率较高.同时该类技术的检测能力有限,有些技术如Jade[7]只能检测2线程互斥锁死锁,有些技术如SDD[8]虽能检测多线程互斥锁死锁,但不能检测读写锁死锁和由互斥锁和读写锁造成的混合死锁.

动态分析是目前死锁检测的主流技术,一般分为离线和在线2类.在线方法监视程序运行,实时获取感兴趣的信息,建立和更新目标程序同步状态的抽象表示,并在其上检测死锁.离线方法先对程序源码静态插桩,然后执行程序并获取执行轨迹,最后分析轨迹,建立锁占用顺序图,将其上检测到的环作为死锁报告出来.在线方法(GoodLock[15-16]和Sherlock[17]除外,它们可以在线预测可疑死锁)一般只能检测到本次执行中实际出现的死锁,而离线方法则还能"预测"在其他执行中可能出现的死锁.例如MagicFuzzer[18]和MagicLock[21]根据本次执行中预测到的死锁信息,在程序的下一次执行中,控制线程调度,试图使死锁暴露出来.相对于在线方法,离线方法误报率较高,需要存储执行轨迹,对长时间运行的并发程序不适用.在线分析不需要人工干预,扩展性好,没有误报,但其漏报率较高.尽管各有其优缺点,但它们都忽略了读写锁可能造成的死锁,只能检测互斥锁死锁.

针对现有方法检测能力有限的问题,为提高现有方法的死锁检测能力,以便一次性检测由多个线程和多个互斥锁或读写锁造成的所有类型死锁,本文提出一种基于锁分配图的混合死锁动态检测方法,如图1所示.该方法主要由监视模块Monitor和检测模块Linter两部分组成.监视模块Monitor负责劫持所有互斥锁和读写锁的加锁解锁操作,根据劫持到的信息生成相应事件并将事件插入到线程的事件队列中.可能生成的事件共有5种:锁互斥请求

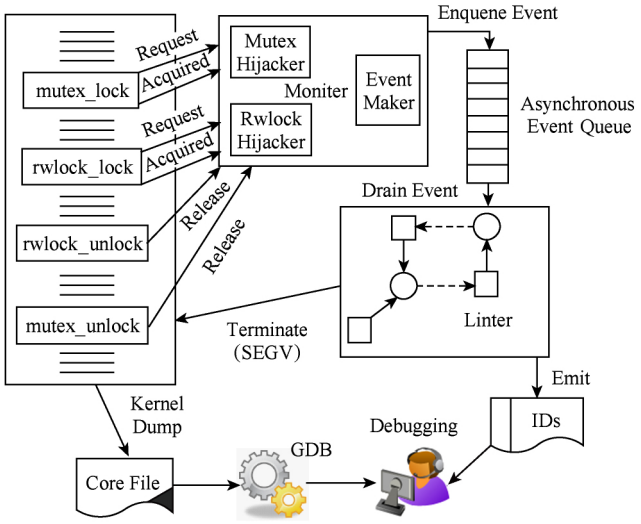

Fig. 1　Dynamic detection algorithm to detect multiple types of deadlocks based on MLAGs

图 1　基于锁分配图的混合死锁动态检测方法

事件、锁互斥占据事件、锁共享请求事件、锁共享占据事件和锁释放事件. 检测模块读取并根据这些事件构建和更新混合锁分配图(multiple-type lock allocation graph, MLAG),在其上使用强连通分量(strong connected component, SCC)算法检测环;如果有死锁环存在,则输出所有的环以及每个环中的所有线程 ID 和锁 ID,并发送 SIGSEGV 信号以中止目标程序. 操作系统将为该目标程序生成转储文件,程序员可以根据得到的死锁环信息和转储文件,使用 GDB 进行源码级别调试,快速理解和定位死锁的发生原因和位置.

我们提出的混合死锁检测方法有多重应用场景:在测试环境中,它可以配合死锁暴露技术来检测和验证死锁;在生产环境中,死锁规避技术可以使用它检测更多类型和更多数量的死锁,从而得到关于这些死锁的特征信息,以指导规避逻辑规避更多类型和更多数量的死锁.

1　检测对象与死锁环判定算法

多线程程序中,pthread 库中 2 种同步设施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但写操作互斥执行的同步场景,为提高性能程序员往往使用读写锁而不是互斥锁. 因此本文试图检测由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、互斥锁和读写锁造成的混合死锁混合死锁、互斥锁自锁和读写锁自锁,如图 2 所示:

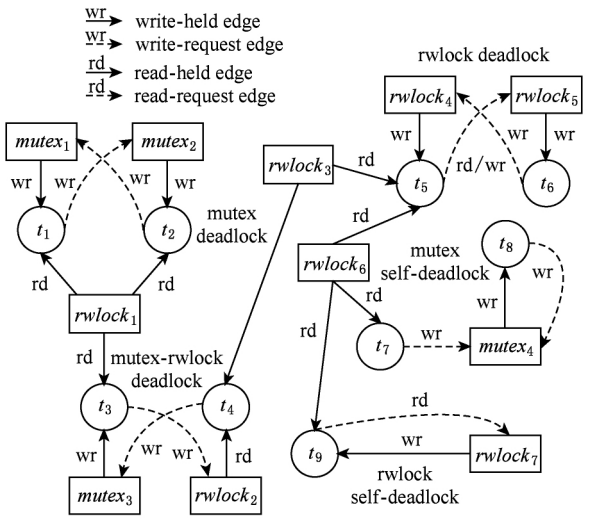

Fig. 2　Five deadlock scenarios caused by mutex and rwlock

图 2　互斥锁和读写锁造成的 5 种死锁场景

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和读请求. 图 2 中用不同的线型和标签标示这 4 种操作.

本文定义混合锁分配图 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 是一个三元组(v,thd,tp_1)或者(thd,v,tp_2),其中 $v\in M(t)\cup RW(t)$, $thd\in T(t)$, $tp_1\in\{wr\_held, rd\_held\}$, $tp_2\in\{wr\_request, rd\_request\}$. (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 的取值依规则而定:①若 $v\in M(t)$,则 tp_1 取值 wr_held,tp_2 取值 $wr\_request$;②若 $v\in RW(t)$,则 tp_1 取值 wr_held 当且仅当不存在(v,thr,rd_held)和(v,thr,wr_held),tp_1 取值 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 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 3 个条件:1)$k\geqslant 4$;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 3 个条件:1)$k\geqslant 4$;2)如果 i 是奇数,令 e_0 为 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 4 个条件:1)$k\geqslant 4$;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 为 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 3 个条件:1)$k=2$;2)$v_1\in M(t)$,$v_2\in T(t)$;3)$tp(e_1)=wr\_held$ 且 $tp(e_2)=wr\_request$.

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 3 个条件: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 加锁解锁劫持算法

监视模块 Monitor 劫持所有互斥锁和读写锁加锁解锁操作,如表 2 所示,并根据劫持到的操作信息生成相应的事件.

Table 2 Lock/Unlock Operations on Mutexes and Rwlocks
表 2 读写锁和互斥锁加锁解锁操作

Lock Type	Lock/Unlock Operation
Mutex	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*);
Rwlock	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*);

2.1 互斥锁加锁解锁劫持算法

pthread 中互斥锁有 3 种类型:NORMAL, ERRORCHECK, RECURSIVE. 不同类型互斥锁的区别在于"未解锁时再次加锁"的执行结果不同:NORMAL 互斥锁将使当前线程陷入死锁,ERRORCHECK 互斥锁将返回错误代码以指示当前线程已经占据该互斥锁,而 RECURSIVE 互斥锁则在增加其锁计数后返回成功值.

Monitor 在劫持互斥锁的加锁解锁操作时,需要知道正被加锁或者解锁的互斥锁的类型,以根据不同的类型作出不同的反应. 然而在 pthread 中,互

斥锁的类型一旦被设置后就无法仅仅从该互斥锁获知其是何种类型.为绕开这个限制,我们参考了互斥锁的内部定义,并依据定义取互斥锁的第4个域的值为互斥锁的类型.这种方法在Linux系统上是可行的,然而可能不具有可移植性.我们建议在其下一版中,pthread应增加一个获取已初始化的互斥锁的类型属性的函数.

互斥锁 *lock* 操作的劫持算法分别如算法1所示,其中"*gettype*"负责如前所述的检索互斥锁类型.

算法1. 互斥锁加锁操作 *lock* 劫持算法.

输入: 互斥锁标识符 v 和线程标识符 t;

输出: 整型值,指示 t 是否成功获取 v.

① 设 t 为当前线程标识符;
② 设 v 为当前互斥锁标识符;
③ $type:=gettype(v)$;
④ if t 当前持有 v {
⑤　if $type$ is RECURSIVE{
⑥　$v.lock\_count++$;
⑦　return 0;}
⑧　else if $type$ is ERRORCHECK{
⑨　return $native\_mutex\_lock(v)$;}}
⑩ 向 $t.eq$ 插入事件(t,v,WR_REQUEST);
⑪ $ret:=native\_mutex\_lock(v)$;
⑫ if $ret=0$ {
⑬　if $type$ is RECURSIVE{
⑭　$v.lock\_count++$;}
⑮　向 $t.held\_locks$ 中插入 v;
⑯　向 $t.eq$ 中插入事件(t,v,WR_HELD);}
⑰ return ret.

在对互斥锁加锁操作的劫持算法(算法1)中,如果线程 t 当前没有占据互斥锁 v,则Monitor将生成一个写请求事件并将其插入到线程 t 的事件队列 $t.eq$(行⑩).劫持算法接着对 v 调用真正的加锁操作并检查它是否成功.如果成功,则 t 已成功占据 v,此时检测算法将 v 加入到线程 t 的已占据锁集合 $t.held\_locks$ 中,然后生成一个写占据事件并将其插入到 $t.eq$.在此过程中,如果 v 是递归互斥锁,则将其锁计数增加1(行⑪～⑯).

而如果线程 t 当前已经占据 v,则劫持算法根据 v 的类型进行下一步动作(行④～⑩):1)如果 $type$ 是RECURSIVE,则将 v 的锁计数增加1,然后返回0以指示目标程序的加锁调用执行成功(行⑤～⑦);2)如果 $type$ 是ERRORCHECK,则返回一个合适的错误代码以指示目标程序的加锁调用执行失败,这通过调用真正加锁操作并返回其结果值来实现(行⑧～⑨);3)如果 $type$ 是NORMAL,则线程 t 将陷入互斥锁自锁状态,故Monitor向 $t.eq$ 添加WR_REQUEST事件(行⑩),以便检测模块Linter在将来检测到该自锁.

互斥锁 *unlock* 操作的劫持算法分别如算法2所示,其中"*gettype*"同样负责检索互斥锁类型.

算法2. 互斥锁解锁操作 *unlock* 劫持算法.

输入: 互斥锁标识符 v 和线程标识符 t;

输出: 整型值,指示 t 是否成功释放 v.

① 设 t 为当前线程标识符;
② 设 v 为当前互斥锁标识符;
③ $type:=gettype(v)$;
④ if t 当前持有 v {
⑤　if $type$ is RECURSIVE{
⑥　$v.lock\_count--$;
⑦　if $v.lock\_count\neq 0$ {
⑧　return 0;}}
⑨　向 $t.eq$ 插入事件(t,v,RELEASE);
⑩　return $native\_mutex\_unlock(v)$;}
⑪ return $native\_mutex\_unlock(v)$.

在对互斥锁解锁操作的劫持算法(算法2)中,如果线程 t 试图释放一个不被它持有的互斥锁 v,则会引起"未占用解锁"错误.对此Monitor直接调用真正的解锁操作并返回其结果值,以让目标程序有机会处理该错误(行⑪).如果 t 试图释放一个已被它占据的RECURSIVE互斥锁 v,则劫持算法首先将 v 的锁计数减少1然后检查其是否为0.如果锁计数没有被减到0的话,则返回0以指示目标程序的锁释放操作执行成功(行⑥～⑧).如果锁计数已经是0,则Monitor向 $t.eq$ 插入一个互斥锁释放事件,并对 v 调用真正的释放操作以释放该锁(行⑨～⑩).当然,如果 v 不是RECURSIVE互斥锁,则Monitor直接向 $t.eq$ 插入一个RELEASE事件,并对 v 调用真正的释放操作以释放之(行⑨～⑩).

Monitor对互斥锁 *trylock* 和 *timedlock* 操作的劫持算法与算法1类似,为简洁起见不再赘述.

2.2　读写锁加锁解锁劫持算法

pthread中读写锁没有类型之分,因此对读写锁加锁解锁的劫持算法比对互斥锁加锁解锁的劫持算法更简单.读写锁 *wrlock*,*rdlock*,*unlock* 操作的劫持算法分别如算法3、算法4和算法5所示.

算法3. 读写锁互斥加锁操作 *wrlock* 劫持算法.

输入: 读写锁标识符 v 和线程标识符 t;

输出: 整型值,指示 t 是否成功互斥占据 v.

① 设 t 为当前线程标识符;

② 设 v 为当前互斥锁标识符;
③ 向 $t.eq$ 插入事件(t,v,WR_REQUEST);
④ ret:=$native\_rwlock\_wrlock(v)$;
⑤ if ret=0 {
⑥ 向 $t.eq$ 插入事件(t,v,WR_HELD);}
⑦ return ret.

算法 4. 读写锁共享加锁操作 *rdlock* 劫持算法.
输入: 读写锁标识符 v 和线程标识符 t;
输出: 整型值,指示 t 是否成功共享占据 v.
① 设 t 为当前线程标识符;
② 设 v 为当前互斥锁标识符;
③ 向 $t.eq$ 插入事件(t,v,RD_REQUEST);
④ ret:=$native\_rwlock\_rdlock(v)$;
⑤ if ret=0 {
⑥ 向 $t.eq$ 插入事件(t,v,RD_HELD);}
⑦ return ret.

算法 5. 读写锁解锁操作 *unlock* 劫持算法.
输入: 读写锁标识符 v 和线程标识符 t;
输出: 整型值,指示 t 是否成功释放 v.
① 设 t 为当前线程标识符;
② 设 v 为当前互斥锁标识符;
③ ret:=$native\_rwlock\_unlock(v)$;
④ if ret=0 {
⑤ 向 $t.eq$ 插入事件(t,v,RELEASE);}
⑥ return ret.

读写锁互斥加锁操作的劫持算法(算法 3)首先向当前线程 t 的事件队列插入 t 对当前读写锁 v 的写请求事件,然后对 v 调用真正的读写锁互斥加锁操作,只有当其成功执行时才将一个写占据事件插入到 $t.eq$,最后向目标程序返回其返回值.

读写锁共享加锁操作的劫持算法(算法 4)与对互斥加锁操作的劫持算法类似,只是将插入事件队列 $t.eq$ 的写请求和写占据事件改为读请求和读占据事件.而读写锁释放操作的劫持算法(算法 5)直接在锁释放成功后向 $t.eq$ 插入一个锁释放事件.

另外,Monitor 对读写锁 *trywrlock* 和 *timedwrlock* 操作的劫持算法与算法 3 类似,对读写锁 *tryrdlock* 和 *timedrdlock* 的劫持算法与算法 4 类似,不再赘述.

3 混合锁分配图构建和环检测算法

检测模块 Linter 动态构建和更新混合锁分配图,并负责检测其上是否有死锁环.实际上,检测模块被实现为一个驻留在目标程序进程空间的独立线程,它周期性地(比如 0.1 s)休眠和苏醒,以减少对目标程序不必要的干扰.当 Linter 苏醒时,它从每一个在其休眠期间执行过加锁解锁操作的线程的事件队列中读取事件,并根据这些事件更新 MLAG.算法 6 给出了 Linter 处理事件和更新 MLAG 的整体过程.

算法 6. 混合锁分配图构建和环检测算法.
输入: 各个线程队列中的事件;
输出: 环集合.
① 设 *mlag* 为全局混合锁分配图;
② 令 *thrs* 为在 Linter 休眠期间进行过 *lock*/*unlock* 操作的所有线程构成的集合;
③ 令 *reqthrs* 为在 Linter 休眠期间发出过加锁请求但到目前为止仍没有占据相应锁的线程的集合;
④ foreach t in *thrs*{
⑤ 令 *num* 为当前 $t.eq$ 的长度;
⑥ while *num*≠0 {
⑦ *num*−−;
⑧ *event*=*dequeue*($t.eq$);
⑨ switch *event*.*type*{
⑩ case WR_REQUEST:
⑪ 向 *mlag* 添加一条从 *event*.t 到 *event*.v 的 WR_REQUEST 的边;
⑫ 将 t 添加到 *reqthrs*;break;
⑬ case RD_REQUEST:
⑭ 向 *mlag* 添加一条从 *event*.t 到 *event*.v 的 RD_REQUEST 类型的边;
⑮ 将 t 添加到 *reqthrs*;break;
⑯ case WR_HELD:
⑰ 从*mlag* 中删除从 *event*.t 到 *event*.v 的 WR_REQUEST 类型的边;
⑱ 向*mlag* 添加一条从 *event*.v 到 *event*.t 的 WR_HELD 类型的边;
⑲ 从 *reqthrs* 中删除 t;break;
⑳ case RD_HELD:
㉑ 从*mlag* 中删除从 *event*.t 到 *event*.v 的 RD_REQUEST 类型的边;
㉒ 向*mlag* 添加从 *event*.v 到 *event*.t 的 RD_HELD 类型的边;
㉓ 从 *reqthrs* 中删除 t;break;
㉔ case RELEASE:

㉕　从 $mlag$ 中删除从 $event.v$ 到 $event.t$ 的 WR_HELD(RD_HELD)类型的边;break;}}}

㉖ return $tarjan\_scc(mlag, reqthrs)$.

在算法 6 中,对于一个在其休眠期间执行过加锁解锁操作的线程 t,Linter 首先从 $t.eq$ 中读取 num 个事件.这个数字是在 Linter 正要读取 t 的事件时确定的(行⑤).由于 $t.eq$ 是无锁队列[①],因此它可能同时被线程 t 和线程 Linter 访问和修改.例如当 Linter 正从 $t.eq$ 的队尾读取事件时,t 可能正将一个新事件添加到 $t.eq$ 的队头.因此为避免冲突,不管线程 t 是否向 $t.eq$ 添加事件,Linter 只从队尾读取 num 个事件.如果 $t.eq$ 中还有其他事件,Linter 会在下一次苏醒期间处理它们.

Linter 读取事件后根据事件的类型决定如何更新 $mlag$(行⑧～㉕):如果 $event$ 是写请求或者读请求类型,则向 $mlag$ 添加从线程 t 到锁 v 的相应类型的边(行⑩～⑮);如果 $event$ 是写占据或者读占据事件,则先删除从 t 到 v 的相应请求边,然后添加从 v 到 t 的写占据或读占据边(行⑯～㉓);如果事件是释放事件,则删除从 v 到 t 的占据边(行㉔～㉕).

同时,在处理事件的过程中 Linter 将那些发出了锁请求但至今没有占据相应锁的线程记录在 $reqthrs$ 中.在读取事件并根据事件更新 $mlag$ 后,Linter 会从 $reqthrs$ 中线程所对应的顶点而不是 $mlag$ 中所有线程顶点开始检测是否存在环(算法 6 行㉖),这是因为若 $mlag$ 中存在环,则这些环必然包括 $reqthrs$ 中的线程所对应的顶点.

Linter 使用 Tarjan 强连通分量算法[②]在 $mlag$ 上检测环.由于只为 $|reqthrs|$ 个顶点调用 $Tarjan$ 算法,故这里环检测算法时间复杂度是 $O(|reqthrs|+O|E(t)|)$. Linter 根据第 1 节给出的判定算法(定义 2～6)判断检测到的环是否是死锁环.之所以要进行判定,是因为在 $mlag$ 上检测出来的某些环不一定是死锁环.例如下面这样的环就不是一个死锁环:线程 t_1 已经共享占据读写锁 $rwlock_1$,正互斥请求锁定互斥锁 mtx_1,但 mtx_1 已经被线程 t_2 互斥占据,并且 t_2 正共享请求锁定 $rwlock_1$.

若已判定完所有环且检测到有死锁环存在,则 Linter 向目标程序发送 SIGSEGV 信号并输出所有的死锁环以及每个环中的线程 ID 和锁 ID.程序员可以根据这些信息和操作系统为目标程序生成的转储文件,使用 GDB 对死锁进行源码级别调试.

4　实验与分析

根据本文提出的检测方法,我们在 Linux-3.2.0 上开发了一个原型工具 Docklinter(实际上是一个动态链接库),以检测用 pthread 库编写的多线程 C/C++程序中的混合死锁.为评估 Docklinter 的死锁检测能力、对目标程序造成的性能影响和可扩展性,本节通过实验回答 3 个问题:

1) Docklinter 的检测能力如何,即能否准确检测所有互斥锁和读写锁造成的死锁.

2) Docklinter 的性能影响如何,即是否会给目标程序造成较大的性能下降.

3) Docklinter 的可扩展性如何,即能否在线程数目指数级增长的情况下,其检测开销保持平稳增长.

4.1　基准死锁程序集

为回答这些问题,本节使用如表 3 所示的基准死锁程序集:由作者编写的 5 个含有不同类型死锁的程序和在死锁检测和死锁规避领域[12-13,22-24]广泛使用的 8 个死锁程序构成.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 $usleep$ 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%的概率暴露出来.

表 3 列出所有死锁缺陷、相应死锁的程序、死锁缺陷的具体类型(见第 1 节所述的 5 类死锁)以及死锁缺陷中包含的死锁环的数量、互斥锁的数量和读写锁的数量. deadlock-mutex-1, deadlock-mutex-2, deadlock-mrwlock-1, deadlock-mrwlock-2, deadlock-mrwlock-3 是作者自己编写的死锁程序,分别包含一个互斥锁死锁、一个互斥锁自锁、一个混合死锁、一个读写锁死锁和一个读写锁自锁. bank-transaction, dining-philosophers, sqlite-3.3.3[③], hawknl-1.6b3 各自包含一个互斥锁死锁缺陷 bug#6, bug#7, bug#10, bug#11,其中 bug#6 中的互斥锁是 RECURSIVE 类型锁. tgrep 和 mysql-6.0.4-alpha[④]则各自包含一个读写锁死锁缺陷. sshfs-fuse-2.2 和 openldap-2.2.20[⑤]各自包含一个混合死锁缺陷.注

① http://preshing.com/20120612/an-introduction-to-lock-free-programming/

② https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

③ http://www.sqlite.org/src/info/a6c30be214

④ http://bugs.mysql.com/bug.php?id=37080

⑤ http://www.openldap.org/lists/openldap-bugs/200501/msg00101.html

意 bug#1,bug#3,bug#4,bug#9 都包含多个能够同时存在的死锁环,传统检测方法不能一次性检测到所有死锁环. 除 bug#10～bug#13 外,所有死锁缺陷都可以通过直接运行相应程序进行触发. 对 bug#10～bug#13,我们编写相应触发用例触发它们.

4.2 实验环境

所有评测和对比实验在下列环境下进行:4 核 Intel Core 2 Q8200 2.33 GHz CPU、2 GB 内存、Ubuntu 12.04 操作系统(内核 Linux-3.2.0)、GCC 4.6.3 编译器. Linter 的睡眠周期为 0.1 s.

4.3 检测能力评测与对比

为比较 Docklinter 与已有死锁检测工具的死锁检测能力,我们使用 Docklinter 和 Dimmunix 检测在表 3 中列出的 13 个死锁缺陷,并对它们的检测能力进行对比. 对任何一个死锁,我们对其分别使用 Docklinter 和 Dimmunix 检测 30 次. 只要有一次某个工具没有或者没有完全检测到某个死锁的所有死锁环,则我们认为该工具不能检测到该死锁.

为比较 Docklinter 与已有死锁预测工具的死锁检测能力,我们根据文献[16]重新实现了 GoodLock,用它来监视程序执行,劫持线程创建、线程汇合、互斥锁加锁/解锁和读写锁加锁/解锁共 4 类操作,并根据收集到的信息建立锁占用图(lock graph),最后在锁占用图上检测有效环(valid cycle). GoodLock 检测到的环就是它预测到的可疑死锁. 为让 GoodLock 顺利监视目标程序运行,我们删除相应缺陷程序中的 *usleep* 语句,使得它们的每次运行都不会触发死锁缺陷. 如果不这样做,GoodLock 可能收集不到足够的信息以进行死锁预测.

Table 3 The Deadlock Program Benchmark Suite

表 3 基准死锁程序集

Buggy Program	Bug #	Deadlock Type	DN	MN	RWN	Description
deadlock-mutex-1	1	mutex deadlock	2	4	0	Two pairs of threads, each pair acquires two mutexes in different order
deadlock-mutex-2	2	mutex self-deadlock	1	1	0	One thread acquires a mutex without first releasing it
deadlock-mrwlock-1	3	mixed deadlock	3	6	3	Three pairs of threads, each pair acquires one rwlock and two mutexes in different order
deadlock-mrwlock-2	4	rwlock deadlock	2	0	4	Two pairs of threads, each pair acquires two rwlocks in different order
deadlock-mrwlock-3	5	rwlock self-deadlock	1	0	1	One thread acquires a rwlock without first releasing it
bank-transaction	6	mutex deadlock	1	2	0	Deposit and withdraw are separated and protected by different mutexes
dining-philosophers	7	mutex deadlock	1	5	0	Each philosopher first picks up the left stick, then the right stick
tgrep	8	rwlock deadlock	1	0	2	Reverse lock acquisition on *work_q_lk* and *running_lk*
sshfs-fuse-2.2	9	mixed deadlock	3	3	3	three pairs of threads, each pair acquires one mutex and one rwlock in reverse order
sqlite-3.3.3	10	mutex deadlock	1	2	0	*sqlite3UnixEnterMutex*() and *sqlite3UnixLeaveMutex*()
hawknl-1.6b3	11	mutex deadlock	1	2	0	*nlshutdown*() and *nlclose*()
openldap-2.2.20	12	mixed deadlock	1	1	1	*bdb_cache_add*() and *bdb_cache_find_id*()
mysql-6.0.4-alpha	13	rwlock deadlock	1	0	2	Two threads perform insert and truncate operations on the same table with the falcon engine

DN: # of deadlock cycles, MN: # of mutexes, RWN: # of rwlocks.

死锁检测结果如表 4 所示,其中√或×后面的数字($m/n/l$)分别表示相应死锁缺陷中的全部死锁环数目 m、检测工具检测到的死锁环数目 n 和检测结果经人工检查确认为真的死锁环数目 l. 从表 4 可以看出,Docklinter 能成功检测到所有 13 个不同类型的死锁,包括多个死锁环同时出现的死锁 bug#1,bug#3,bug#4,bug#9. Dimmunix 不能检测互斥锁自锁(bug#2)、读写锁死锁(bug#4,bug#8,bug#13)、读写锁自锁(bug#5)和混合死锁(bug#3,bug#9,bug#12),且只能检测到部分互斥锁死锁如 bug#6,bug#10,bug#11. Dimmunix 不能检测到互斥锁自锁 bug#2,是因为它的环检测算法无

法检测自环.Dimmunix不能检测读写锁死锁、读写锁自锁和混合死锁,是因为它没有劫持相关读写锁操作,没有能够建立关于程序相对于互斥锁和读写锁的同步状态的抽象表示,也没有在此种抽象表示上检测相应死锁环的算法.

尽管bug#1和bug#7与bug#6,bug#10,bug#11一样都是互斥锁死锁,但Dimmunix无法检测到它们.对于含有2个死锁环的bug#1,Dimmunix只检测出其中的一个死锁环,因此我们认为它不能检测到bug#1.对于单环死锁bug#7,Dimmunix有时能检测到它而有时检测不到它.为查明出现这种现象的原因,我们仔细检查了Dimmunix的源码,发现Dimmunix对用于存储事件的无锁队列的实现存在数据竞争错误.在这种情况下,当多个线程同时读取和修改无锁队列时,一些加锁解锁事件会由于数据竞争而丢失,从而根据无锁队列中的事件构造出来的锁分配图会因缺少或者多出某些边而不能反映目标程序的真实同步状态,这样Dimmunix就可能检测不到某些已经发生的死锁.

Table 4　Deadlock Detection Results of Docklinter, Dimmunix, GoodLock

表4　Docklinter,Dimmunix,GoodLock的死锁检测结果

Bug	Docklinter		Dimmunix		GoodLock	
	DR	$m/n/l$	DR	$m/n/l$	DR	$m/n/l$
bug#1	√	2/2/2	×	2/1/1	√	2/2/2
bug#2	√	1/1/1	×	1/0/0	×	1/0/0
bug#3	√	3/3/3	×	3/0/0	√	3/6/3
bug#4	√	2/2/2	×	2/0/0	√	2/2/2
bug#5	√	1/1/1	×	1/0/0	×	1/0/0
bug#6	√	1/1/1	√	1/1/1	√	1/1/1
bug#7	√	1/1/1	×	1/0/0	√	1/1/1
bug#8	√	1/1/1	×	1/0/0	√	1/1/1
bug#9	√	3/3/3	×	3/0/0	√	3/3/3
bug#10	√	1/1/1	√	1/1/1	√	1/1/1
bug#11	√	1/1/1	√	1/1/1	√	1/1/1
bug#12	√	1/1/1	×	1/0/0	√	1/1/1
bug#13	√	1/1/1	×	1/0/0	√	1/1/1

Notes: DR: Detection Results; √: yes; ×: no; $m/n/l$: # of total/detected/confirmed cycles in a deadlock bug.

Dimmunix的检测模块检测不到bug#1～bug#5,bug#7～bug#9,bug#12,bug#13,从而无法为这些死锁生成特征签名,这导致Dimmunix的规避模块也不能规避这些死锁缺陷.我们修改Dimmunix源码,令其监视表2中列出的所有读写锁加锁解锁操作,建立并更新混合锁分配图MLAG,然后使用本文提出的混合死锁检测方法检测死锁,这样Dimmunix就能检测到表3中列出的所有死锁并生成关于它们的特征签名.根据这些签名,Dimmunix就能够规避除bug#2和bug#5外的其他11个死锁缺陷了.对于互斥锁自锁bug#2和读写锁自锁bug#5,由于它们的发生是在单个线程内进行的,与线程间调度执行顺序没有关系,因此基于线程调度的Dimmunix无法规避它们.

从表4可以看出,GoodLock能根据相应目标程序的一次无死锁执行而预测出除bug#2和bug#5外的所有死锁缺陷.GoodLock不能预测bug#2和bug#5也是因为其环检测算法无法检测自环.GoodLock能预测到包括读写锁死锁和混合死锁在内的其他11个死锁缺陷,令人出乎意料.实际上它并未区分互斥锁和读写锁的不同语义,而是将读写锁等同于互斥锁,将所有对读写锁的共享请求(即rdlock)和互斥请求(即wrlock)都视为互斥请求,这种简单处理反而导致它能检测到bug#3,bug#4,bug#8,bug#9,bug#12,bug#13等读写锁死锁和混合死锁.

然而,不对共享请求和互斥请求进行区分会导致GoodLock报告虚假死锁环信息,例如对bug#3所在的程序deadlock-mrwlock-1,GoodLock报告检测到6个可疑死锁环,然而经分析确认后,只有3个死锁环会真正造成死锁,其他3个死锁环并不会真正导致死锁.图3(a)给出了deadlock-mrwlock-1中一对线程t_1和t_2的伪代码,当这2个线程按照先t_1后t_2的顺序执行后,GoodLock会为它们生成如图3(b)所示的锁图,并在其上检测有效环[23],得到2个死锁环:$cycle_1=\langle(rw_1,t_1,mtx_1),(mtx_1,t_2,rw_1)\rangle$和$cycle_2=\langle(mtx_1,t_1,mtx_2),(mtx_2,t_2,mtx_1)\rangle$.其中$cycle_1$对应的执行场景是:$t_2$先执行$L_{07}$和$L_{08}$,然后$t_1$执行$L_{01}$和$L_{02}$,最后$t_2$执行$L_{09}$.对于此执行场景,Docklinter会为其生成如图3(c)所示的混合锁分配图,显然其上存在一个环,然而由于环中读写锁顶点rw_1的出边和入边都是rd类型的边,故该环不是死锁环,从而$cycle_1$对应的执行场景不会发生死锁,也即$cycle_1$是虚假死锁环.

另外,即使GoodLock能够意外地预测到读写锁死锁和混合死锁,如果没有Docklinter这样的多类型死锁检测方法,GoodLock也无法确认预测到

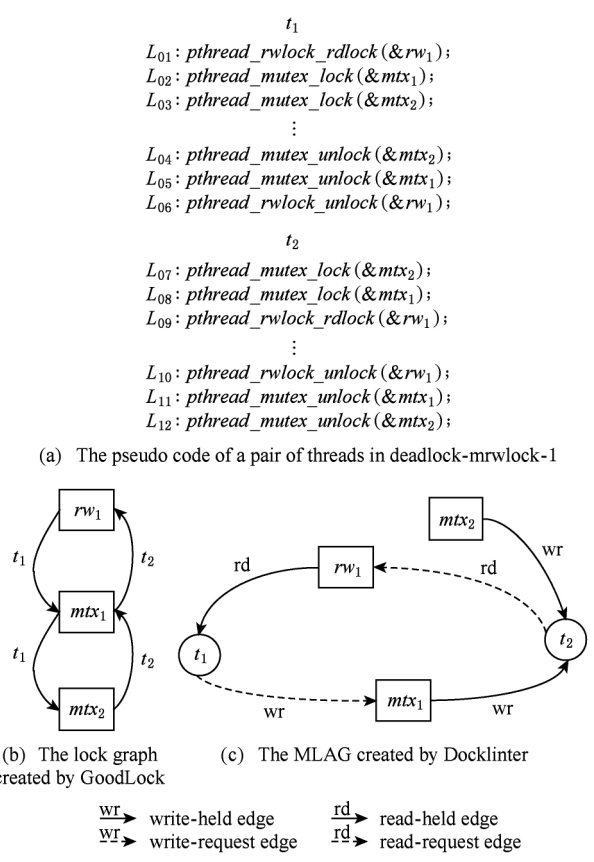

Fig. 3 A false deadlock cycle reported by GoodLock

图 3 GoodLock 报告的一个虚假死锁环

的死锁到底是否是真正的死锁. 因此我们提出的检测死锁方法与 GoodLock 等预测死锁方法在应用场景上是互补的.

4.4 性能影响评测与分析

为评估 Docklinter 对目标程序性能的影响,我们选择 openldap-2.2.20 作为目标程序,编写 2 个测试用例 addent 和 delent,并分别在对 openldap-2.2.20 使用和不使用 Docklinter 的情况下运行这 2 个测试用例. 所有测试用例的运行都不会令 openldap-2.2.20 的守护进程 slapd 陷入死锁:虽然 openldap-2.2.20 中存在并发缺陷 bug#12,但我们特意让测试用例按序向 slapd 发送不同类型的请求,从而有意识地避开了该死锁.

在实验中,我们先运行 addent 以向 slapd 发送 *num* 个 INSERT 请求,然后运行 delent 向 slapd 发送同等数量同样内容的 DELETE 请求. 我们称 addent 和 delent 每次发送的请求数量 *num* 为工作量,其可以从 2 310 变化到 111 210,如表 5 所示. 我们为每一个工作量按照如上所述运行 addent 和 delent 30 次,统计它们完成工作量所需的平均 CPU 时间,如表 5 所示. 其中 Original 表示未使用 Docklinter 监视运行的原始 slapd,Docklinted 表示在 Docklinter 监视下运行的 slapd.

Table 5 The Performance Overhead Incurred by Docklinter for openldap-2.2.20

表 5 Docklinter 对 openldap-2.2.20 造成的性能开销

Workload	Original Time/s		Mocklinted Time/s		Overhead/%	
	addent	delent	addent	delent	addent	delent
2 310	0.118	0.102	0.117	0.106	−1.02	4.31
3 410	0.173	0.150	0.177	0.154	2.31	2.93
5 610	0.290	0.248	0.300	0.263	3.59	5.80
10 010	0.517	0.438	0.559	0.480	8.04	9.49
18 810	0.996	0.835	1.054	0.904	5.87	8.29
36 410	1.954	1.613	2.042	1.747	4.46	8.28
45 210	2.426	1.978	2.538	2.179	4.62	10.15
67 210	3.603	2.961	3.867	3.226	7.32	8.96
89 210	4.812	3.944	5.145	4.322	6.92	9.57
111 210	6.015	4.902	6.346	5.328	5.51	8.71

从表 5 可以看出,当 slapd 在 Docklinter 监视下运行时,addent 和 delent 一般需要更多时间(相对于 slapd 单独运行时)才能完成某一工作量. 但是 Docklinter 对目标程序的性能影响是可接受的:实验结果表明 delent 的最大性能下降是 10.15%,而 addent 的最大性能下降仅为 8.04%. 另外,Docklinter 对目标程序的性能影响也不会随着目标程序工作量的增大而单调增长. 这是因为 Docklinter 以异步方式来检测死锁环,而且仅仅为那些发出加锁请求且还没有占据互斥锁的线程检测死锁. 只有当大量线程频繁地发出加锁请求而没有占据锁时,Docklinter 才会导致较大的开销.

4.5 可扩展性评测与分析

为评估 Docklinter 的可扩展性,我们仍选择 openldap-2.2.20 作为目标程序,编写一个测试用例 searchent,并分别在对 openldap-2.2.20 使用和不使用 Docklinter 的情况下运行这个测试用例. searchent 只向 slapd 发送检索请求,不会触发 openldap-2.2.20 中的 bug#12.

在实验中,我们运行 searchent 以模拟 *num_client* 个客户端向 slapd 发送 20 480 个 SEARCH 请求的应用场景. 每个客户端发送请求的数量为 20 480/*num_client*. 这里一个客户端用一个线程表示,该线程会像独立的客户端一样,通过单独的连接与 slapd 通信. 我们针对每个 *num_client* 的取值,运行 searchent 30 次,统计其平均运行时间,结果如表 6 所示:

Table 6　The Scalability Results of Docklinter on openldap-2.2.20

表 6　Docklinter 在 openldap-2.2.20 上的可扩展性实验结果

#Client	Original Time/s	Docklinted Time/s	Overhead/%
2	4.175	5.495	31.61
4	3.843	5.088	32.40
8	3.693	5.135	39.06
16	3.703	5.284	42.69
32	3.781	5.388	42.49
64	4.568	8.508	86.24
128	4.503	9.223	104.81
256	4.455	9.107	104.42
512	4.060	8.889	118.97
1 002	3.650	8.230	125.49

根据表 6,我们可知 Docklinter 的扩展性良好:Docklinter 对目标程序的性能影响随客户端数目的增多而增大,但是增速十分平缓.实验结果表明当 2 个客户端发送请求时,Docklinter 带来 31.61%的性能下降,而当客户端数目是 32 时,Docklinter 才导致 42.49%的性能下降,甚至当 *num_client* 增长到 1 002 时,Docklinter 带来的性能下降也仅仅是 125.49%.

5　结　　论

本文提出混合锁分配图的概念和构建方法,提出混合锁分配图上的死锁环判定算法,并提出一种基于锁分配图的混合死锁动态检测算法,在此基础上开发了一个原型工具 Docklinter,以在混合死锁发生时一次性检测由多个线程和多个互斥锁或者读写锁造成的所有类型死锁.针对 13 个 5 种类型的死锁缺陷的死锁缺陷检测结果实验以及在 openldap-2.2.20 进行的性能影响评估实验和可扩展性测试实验结果表明:Docklinter 死锁检测能力强,对目标程序的性能影响较小且扩展性良好.

参　考　文　献

[1] 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)

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

[3] Fonseca P, Li Cheng, 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. Piscataway, NJ: IEEE, 2010: 221-230

[4] Lu Shan, 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

[5] Boyapati C, Lee R, Rinard M. Ownership types for safe programming: Preventing data races and deadlocks [J]. ACM SIGPLAN Notices, 2002: 37(11): 211-230

[6] Gordon C S, Ernst M D, Grossman D. Static lock capabilities for deadlock freedom [C] //Proc of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. New York: ACM, 2012: 67-78

[7] Naik M, Park C S, Sen K, et al. Effective static deadlock detection [C] //Proc of the 31st Int Conf on Software Engineering. New York: ACM, 2009: 386-396

[8] Williams A, Thies W, Ernst M D. Static deadlock detection for Java libraries [C] //Proc of the 19th European Conf on Object Oriented Programming. New York: ACM, 2005: 602-629

[9] Engler D, Ashcraft K. RacerX: Efficient, static detection of race conditions and deadlocks [J]. ACM SIGOPS Operating Systems Review, 2005, 37(5): 237-252

[10] Pradel M, Gross T R. Fully automatic and precise detection of thread safety violations [J]. ACM SIGPLAN Notices, 2012, 47(6): 521-530

[11] Joshi P, Park C S, Sen K, et al. A randomized dynamic program analysis technique for detecting real deadlocks [C] //Proc of the 2009 ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2009: 110-120

[12] Jula H, Tralamazza D, 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 Association, 2008: 295-308

[13] Yu Zhen, Su Xiaohong, Wang Tiantian, et al. Mocklinter: Linting mutual exclusive deadlocks with lock allocation graphs [J]. International Journal of Hybrid Information Technology, 2016, 9(3): 355-374

[14] Koskinen E, Herlihy M. Dreadlocks: Efficient deadlock detection [C] //Proc of the 20th Annual Symp on Parallelism in Algorithms and Architectures. New York: ACM, 2008: 297-303

[15] Havelund K. Using runtime analysis to guide model checking of Java programs [C] //Proc of 7th Int SPIN Workshop on Model Checking of Software. Berlin: Springer, 2000: 245-264

[16] Bensalem S, Havelund K. Dynamic deadlock analysis of multi-threaded programs [C] //Proc of the 1st Int Haifa Verification Conf on Hardware and Software Verification and Testing. Berlin: Springer, 2005: 208-223

[17] Eslamimehr, M, Palsberg, J. Sherlock: Scalable deadlock detection for concurrent programs [C] //Proc of the 22nd ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2014: 353-365

[18] Cai Yan, Chan W K. MagicFuzzer: Scalable deadlock detection for large-scale applications [C] //Proc of the 2012 Int Conf on Software Engineering. New York: ACM, 2012: 606-616

[19] Bensalem S, Havelund K. Scalable dynamic deadlock analysis of multi-threaded programs [C] //Proc of the 3rd Int Workshop on Parallel and Distributed Systems: Testing and Debugging. New York: ACM, 2005: 1-16

[20] Luo Zhida, Das R, Qi Yao. MulticoreSDK: A practical and efficient deadlock detector for real-world applications [C] //Proc of the 4th Int Conf on Software Testing, Verification and Validation. Piscataway, NJ: IEEE, 2011: 309-318

[21] Cai Yan, Chan Wing Kwong. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs [J]. IEEE Trans on Software Engineering, 2014, 40(3): 266-281

[22] 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

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

[24] 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

Yu Zhen, born in 1987. PhD candidate in the School of Computer Sience and Technology at Harbin Institute of Technology. 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 Harbin Institute of Technology. Senior member of CCF. Her main research interests include software engineering, information fusion, image processing and computer graphics.

Qiu Jing, born in 1982. Received his PhD degree from Harbin Institute of Technology in 2015. His main research interests include binary code analysis and binary code de-obfuscation.