计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.190700079

基于采样技术的动态混合数据竞争检测算法

李梦珂　郑秋生　王　磊
中原工学院前沿信息技术研究院　郑州 450007
河南省网络舆情监测与智能分析重点实验室　郑州 450007
(2017007063@zzti.edu.cn)

摘　要　数据竞争是多线程程序并发错误的主要来源,目前已有许多静态和动态程序分析技术用于检测数据竞争,但这些检测器或者会产生巨大的检测开销,或者会漏掉许多真实的数据竞争错误。文中提出了一种基于优化的 FastTrack 算法和锁模式的动态混合数据竞争检测算法 AsampleLock。该算法利用采样技术,监控同一时刻同时运行的来自并发线程的函数对,通过预竞争检测获得真正涉及数据竞争的内存访问对,从而减小竞争检测分析开销;为了减弱线程调度对算法相关性能的影响,AsampleLock 算法采用 nolock-hb 关系来判断访问事件的并发关系;采用 map 记录所有共享变量的读写信息,并采用锁模式进行动态数据竞争检测,降低漏报率和误报率。基于上述方法实现了原型系统 AsampleLock,选择基准测试集 Parsec 对该系统进行评估,并与 FastTrack 算法、LiteRace 算法和 Multilock-HB 算法进行对比。实验结果表明,AsampleLock 算法与 FastTrack 算法相比整体时间开销平均降低了 8%;AsampleLock 算法的数据竞争检测率与 LiteRace 算法和 FastTrack 算法相比分别增加了 39%和 27%。

关键词:多线程程序;数据竞争检测;预竞争检测;锁模式
中图法分类号　TP311.53

Dynamic Hybrid Data Race Detection Algorithm Based on Sampling Technique

LI Meng-ke,ZHENG Qiu-sheng and WANG Lei
Research Institute of Front Information Technology,Zhongyuan University of Technology,Zhengzhou 450007,China
Henan Key Laboratory on Public Opinion Intelligent Analysis,Zhengzhou 450007,China

Abstract　Data race is a major source of concurrency bugs. Numerous static and dynamic program analysis techniques have been proposed to detect data races. However,some of detectors may cause a large detection overhead and some of detectors may miss lots of true races. In this paper,a dynamic hybrid data race detection algorithm AsampleLock is proposed,which is based on the optimized FastTrack algorithm and lock mode. It uses the sampling technique,monitoring the function pairs from concurrent threads running simultaneously at the same time,and obtains memory access pairs that really involve data race through the preliminary data race detection,thereby reducing analysis overhead of race detection. In order to reduce the influence of the algorithm on thread scheduling,AsampleLock adopts nolock-hb relation to judge the concurrency relationship of access events,adopts map to record read and write informations of shared variables,and adopts the locking patterns to perform dynamic data race detection,thereby reducing false positives and false negatives. On the basis of the above methods,this paper implements the prototype system,named AsampleLock,and chooses the Parsec benchmark suite to evaluate the race detectors. Experiments compared to FastTrack algorithm,LiteRace algorithm and Multilock-HB algorithm. The results show that the time overhead of AsampleLock algorithm is reduced by 8% compared with FastTrack algorithm. Compared with LiteRace algorithm and FastTrack algorithm,the data race detection rate of AsampleLock algorithm is increased by 39% and 27%,respectively.

Keywords　Multithreaded program,Data race detection,Preliminary data race,Locking patterns

1　引言

目前,多核、众核处理器广泛应用于高性能计算机、服务器、桌面计算机以及移动终端设备。为了更好地发挥多核、众核设备的优势,多线程程序得到了广泛的使用。但多线程程序在执行时具有不确定性,并且会产生大量可能的交错[1],从而导致严重的并发错误,对并发软件的可靠性构成了严重威胁[2]。常见的并发错误包括死锁、数据竞争、原子性违例、顺序违例。数据竞争是导致其他并发错误的主要原因之一,且 Lu 等发现 69%的原子性违例是由数据竞争引起的[3]。若不

到稿日期:2019-07-10　返修日期:2019-12-09　本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家重点研发计划项目(2016QY07X1503,162300410190)
This work was supported by the National Key Research and Development Project (2016QY07X1503,162300410190).
通信作者:王磊(wl1167@163.com)

同线程 t_1 和 t_2 同时访问同一个共享变量，至少有一个线程对共享变量执行写操作，且这两个访存操作的执行顺序没有被同步语句强制保证，此时会发生数据竞争。

虽然数据竞争的出现并不一定全都能威胁到程序的执行，但隐藏在其中的有害数据竞争会引起内存错误、程序崩溃、错误输出等现象。例如，2012 年带来数亿美元经济损失的 Facebook 公司的 IPO 故障[4]，2016 年被曝光的隐藏在 Linux 内核 2.6.22 以上所有版本中[5]的 DirtyCow 漏洞(CVE-2016-5195)等，都是由数据竞争引起的。这样的事件产生的后果往往难以估计，因此，如何高效准确地检测数据竞争，成为提高并发程序可靠性和安全性亟需解决的问题。

当前国内外研究人员针对数据竞争错误提出了许多可行的检测方案。静态检测分析技术只分析源代码或中间代码的同步调用，漏报率较低，但误报率高。例如，RacerX[6]被用于在大型、复杂的多线程系统中查找错误，利用流敏感和过程间分析来检测数据竞争和死锁。Voung 等提出的静态检测工具 RELAY[7]的扩展性较强，能够扩展到数百万级别代码的程序。相反，动态检测分析技术在程序执行过程中进行检测，通常需要监控大量的内存访问记录、同步等操作，误报率较低，但漏报率高、开销大，且依赖输入。动态检测分析技术的代表性方法有：Djit+[8]，FastTrack[9]，LOFT[10]。Djit+使用向量时钟算法检测数据竞争，FastTrack 和 LOFT 在检测开销、漏报率和误报率方面改进了此算法。基于 Lockset 和 Happens-before 算法的混合检测技术，在单个执行跟踪中增加了检测覆盖率，可跟踪发现隐藏的数据竞争[11]，如 Acculock[12]，MultiLock-HB[13]，Simple-Lock[14]等。

为了减小动态数据竞争检测的运行时开销并降低漏报率和误报率，本文提出了一种基于采样技术，在 FastTrack 算法和锁模式[15]的基础上改进行进的动态混合数据竞争检测算法 AsampleLock。本文的主要工作如下：

(1)基于采样方法获得跨线程内存访问对，定义产生数据竞争条件并保留真正涉及数据竞争的访存信息对，从而减小检测分析的开销。

(2)为了降低算法对线程调度的敏感程度，AsampleLock 算法采用 nolock-hb 关系来判断访问事件之间的并发关系，并基于锁模式进行动态数据竞争检测。在此基础上，AsampleLock 算法采用 map 来记录读写信息，在提高查找速度的同时降低漏报率和误报率。

(3)基于上述算法实现原型系统 AsampleLock。采用多线程基准测试集 Parsec[16]进行评估，并与 FastTrack 算法、LiteRace 算法、Multilock-HB 算法进行实验对比。实验结果证明了所提算法可有效减小检测开销，并降低漏报率和误报率。

2 传统数据竞争检测工具存在的问题

由 Lamport 最先提出的基于向量时钟的 Happens-before 算法[17]，用于判断不同线程事件之间是否存在并发关系，成为检测数据竞争的最初算法之一。FastTrack 算法[9]改进了 Djit+[8]算法，Djit+算法的时间复杂度为 $O(n)$，而 FastTrack 算法采用轻量级 epoch 技术把时间复杂度降低到接近于 $O(1)$。FastTrack 算法对读操作采用自适应的向量时钟，在 epoch 之间进行切换，对写操作采用轻量级 epoch 来记录，但对于同步操作中的同步对象，向量时钟的更新操作的复杂度还是 $O(n)$，在数据竞争检测过程中同步操作相对于读写操作占有较少的比例，因此从整体来看，FastTrack 算法的数据竞争检测时间复杂度为 $O(1)$。但通过持续监视多线程程序的所有内存访问可知，FastTrack 的系统开销可达到 400%～800%[18]。

采样技术通过选择性地监控内存操作，可以有效地降低系统开销。Marino 等提出的采样算法 LiteRace[19]，只处理小部分的内存访问，可显著降低数据竞争检测的运行时开销，但对于现有研究已证实的数据竞争，漏报率为 30%[18]。PACER[20]做出了比例保证，保证了数据竞争检测率与采样率相等，在采样期间 PACER 使用精确的 FastTrack 算法来跟踪所有的访问，在非采样期间 PACER 简化同步操作和变量读写的分析。PACER 简化了 Happens-before 关系的跟踪，减小了竞争检测的空间和时间开销。Guo 等[18]提出跨线程采样函数对的方法来减小运行时及检测分析开销。

基于 Happens-before 和 Lockset 算法的混合检测方法通过监控每一条执行路径上的访存信息，来精确检测出程序执行轨迹中所有实际发生和隐藏的数据竞争。为了提高检测速度，Acculock[12]采用基于 epoch 的 Happens-before 技术保留少量的历史访存信息，但它不能准确地保留共享内存访问的锁集情况，这会引入误报现象。Multilock-HB[13]针对 Acculock 的缺点进行了改进，通过优化技术来维护所有共享内存位置的历史读写信息，从而消除 Acculock 的误报，然而高运行时开销是其最大的缺点[15]。SimpleLock 算法[14]和 SimpleLock+算法[21]分别使用标量变量和布尔变量来表示共享内存操作是否被锁保护，大大提高了检测的准确性和速度，但当数据竞争的两次访问受到不同锁的保护时，可能会出现漏报现象。

3 动态混合数据竞争检测方法

本文设计了一种动态混合数据竞争方法，该方法基于 Intel 公司开发动态二进制插桩框架 PIN[22]，在每条指令前后插入探测语句来获取程序的执行情况，根据跨线程采样算法和预竞争检测过程获得真正涉及数据竞争的内存访问对信息，基于锁模式和改进的 FastTrack 算法的混合检测算法来准确判断数据竞争。

3.1 检测框架

本文通过跨线程采样函数对的采样方法[18]监控在同一时刻执行的来自并发线程的函数对，并获取访问事件对；然后进行预竞争检测[23]，即通过定义数据竞争产生的条件过滤掉不可能产生数据竞争的访问事件对，保留真正涉及数据竞争的访问事件对；把真正涉及数据竞争的访问事件对输入到检测阶段，通过检测算法精确检测出数据竞争错误。数据竞争检测器的总体框架如图 1 所示。

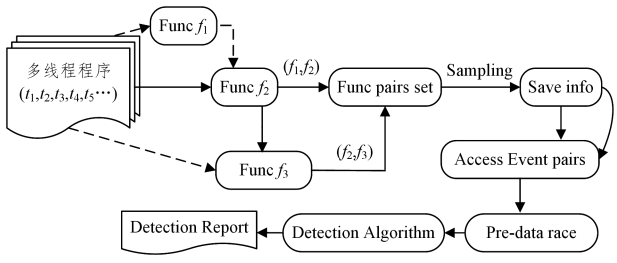

图 1 数据竞争检测器的总体框架

Fig. 1 Overall architecture of datarace detector

3.2 采样策略

本文通过对 Guo 等提出的 AtexRace 算法[18]进行优化，重新设计采样算法的相关策略，最终获得真正涉及数据竞争的访问事件对。

AtexRace 算法忽略数据竞争真正涉及的两个线程和重复执行被测程序的事实，采用一种跨线程和跨执行的采样内存访问的方法，根据给定采样率采用突发采样策略对来自两个并发线程 t_1 和 t_2 在同一时刻运行的两个不同的函数 f_1 和 f_2 进行采样，在整个函数集合中只保持 n 对函数对，即只保留最近经常观察到的函数对，根据监控到的函数对来获取内存访问信息[18]。如图 2 所示，若线程 t_1 的函数 f_1 和线程 t_2 的函数 f_4 同时执行，都访问了共享变量 s_1，且两个线程都对 s_1 执行了写操作，两个线程加了不同的锁 $lock_1$ 和 $lock_2$，则线程 t_1 和 t_2 在第 4 行和第 20 行会发生数据竞争，AtexRace 算法会把函数对〈f_1，f_4〉放进函数对集合中。同样地，若线程 t_1 的函数 f_2 和 t_2 的函数 f_3 在同一时刻执行，AtexRace 算法也会把〈f_2，f_3〉放进函数对集合中。

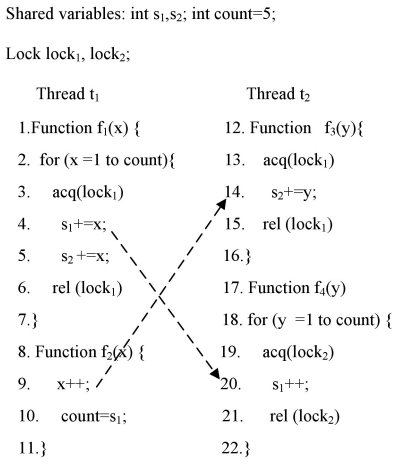

图 2 采样示例

Fig. 2 Example of sampling

AtexRace 算法在大大降低检测开销的同时，能够得到与 FastTrack 相同的数据竞争检测效果[18]，但在获得跨线程函数对的访存信息后，AtexRace 算法并没有做任何处理，这些访存信息是否真正涉及数据竞争，有待进一步判定。例如，在搜集到的两个函数对中，这两个函数没有访问共享变量或没有对共享变量执行写操作等，图 2 中线程 t_1 的函数 f_2 和线程 t_2 的函数 f_3 同时执行时，只有 f_3 对共享变量 s_2 进行了写操作，f_2 并没有访问共享变量，第 9 行和第 14 行不会发生数据竞争，但 AtexRace 算法仍会把函数对〈f_2，f_3〉保留到函数对集合中。把这些不可能涉及数据竞争的函数对输入到检测器会产生大量的检测分析开销。据此，本文提出了改进策略，即在 AtexRace 算法获取到的跨线程内存访问操作中只保留真正涉及数据竞争的事件对，把不涉及数据竞争的访存信息过滤掉，称该方法为预竞争检测[23]。本文把一个访问事件表示为：$Event_n$：{Tid，$variable$，$iswrite$，$lockset$}，Tid 表示线程号，$variable$ 表示是否访问同一个共享变量，$iswrite$ 表示是否是写操作，$lockset$ 表示获得锁集的情况。本文还定义了预数据竞争判定条件，即对于任意两个访问事件 $Event_t$ 和 $Event_{t'}$，存在如下条件：

(1) $Event_t.id \neq Event_{t'}.id$；

(2) $Event_t.variable = Event_{t'}.variable$；

(3) $Event_t.write \cup Event_{t'}.write \neq \emptyset$；

(4) $Event_t.lockset \cup Event_{t'}.lockset \neq \emptyset$。

若满足条件(1)－条件(3)，则 $Event_t$ 和 $Event_{t'}$ 可能存在数据竞争，在预竞争检测阶段不考虑条件(4)。其原因在于线程在进行访存操作时，拥有锁集的情况是受时序限制的。时序限制是指若两个访问事件要获得相同的锁，一个线程先获得锁进行访问变量操作，另一个线程只能等待该线程释放锁后再获取锁，然后对变量进行访问。本文为了简化预竞争检测过程，在不考虑时序限制的情况下进行预竞争检测，在真正检测阶段会详细分析锁集数量。对条件(1)－条件(3)进行判断即可得到所有可能引发真实数据竞争错误的访问事件对。

3.3 检测方法

AsampleLock 算法是 FastTrack 和锁模式相结合的混合算法，FastTrack 是目前较精确的动态数据竞争检测工具，但也会忽略潜在的数据竞争。锁模式是由 Yu 等通过研究现实世界中大量真实存在的数据竞争错误而提出的[15]。两种算法相结合可有效降低误报率和漏报率。前文介绍了通过优化的采样技术可获取真正涉及数据竞争的跨线程内存访问对，将其作为检测阶段的输入，输出为一份检测报告。本节主要分析 map 记录读写信息过程、nolock-hb 关系和锁模式。

3.3.1 nolock-hb 关系

基于 Happens-Before(HB)关系的竞争检测器只能检测到当前执行路径上的数据竞争，若想确定其他线程交错序列中隐藏的数据竞争则需要重复执行程序。在处理同步操作时，HB 算法不仅对线程的创建、等待、阻塞等操作进行了向量时钟的更新，同时对锁获取和锁释放等同步操作也做了相同的处理，大大增加了 HB 算法对线程调度的敏感性[12]，同时也增加了程序运行结果的不确定性[24]。图 3 给出了相同程序的不同线程调度情况，在不同线程调度情况下两种加锁、解锁情况都可能发生，因此也会产生不同的数据竞争检测结果[24]。图 3(a)所示方式符合 HB 关系，但图 3(b)中就会产生写读、写写数据竞争，而且图 3(b)所示情况下最终打印输出的 x 值和图 3(a)所示情况下打印出的 x 值不同。

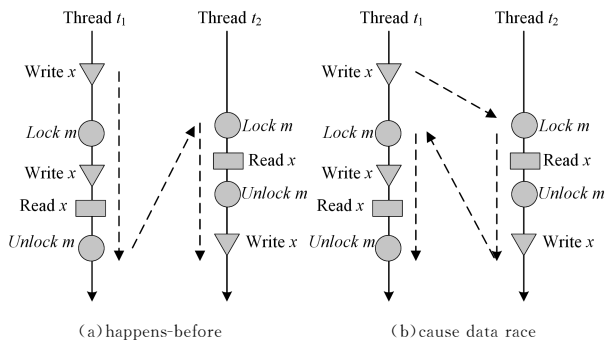

图 3 多线程程序加锁、解锁的不同顺序

Fig. 3 Multithreaded programs lock and unlock in different order

为了降低 AsampleLock 算法对线程调度的敏感性[12]，降低数据竞争检测的漏报率，AsampleLock 算法采用 nolock-hb 关系来跟踪线程，不把加锁、解锁的操作视为同步操作，即不执行锁获取和释放事件的向量时钟更新操作。图 4 给出了 nolock-hb 关系的示例，其不关注加锁、解锁操作，只关注线程的创建、等待、阻塞等同步操作。nolock-hb 关系降低了加锁、

解锁操作对 HB 关系确定性的影响。

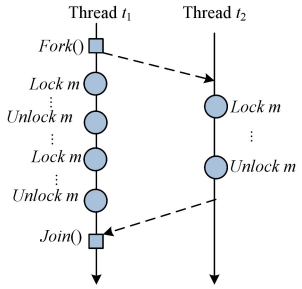

图 4 nolock-hb 关系

Fig. 4 nolock-hb relation

3.3.2 竞争检测

Yu 等通过对大型程序中真实存在的数据竞争错误进行分析，探究了锁模式，针对涉及数据竞争错误的两次访问的锁的数量进行研究，并得出结论：97.1%的数据竞争错误是由不受任何锁保护的访问引起的；2.9%的数据竞争错误是由两次访问受不同的锁保护引起的[15]。AsampleLock 算法就是基于此结论提出的。

当有内存访问事件对$\langle Event_t, Event_{t'}\rangle$输入时，AsampleLock 首先检查这两个访问事件之间是否存在 nolock-hb 关系，若不存在，则表示这两个线程之间的访问顺序是并发的。在不存在 nolock-hb 关系的情况下，检查锁模式情况。若两次对同一共享变量的访问（至少一次为写操作）没有被相同的锁保护，或没有被锁保护，或一次访问被锁保护而另一次没有被锁保护，则可以判断：在某一个特定的执行路径上，两次访存之间一定会发生数据竞争。

FastTrack 算法对读操作采用自适应的向量时钟并在 epoch 之间进行切换，对写操作采用轻量级的 epoch 来记录，很大程度上降低了系统开销。但由于共享内存单元中保存的历史访存信息不完整，存在漏报数据竞争的可能性。例如，在图 3(b)所示的情况下，按照 FastTrack 算法只对最后一次的写操作进行记录，则只记录线程 t_1 的第二次写操作，而忽略了第一次写操作，导致线程 t_2 读写共享变量时，无法获取 t_1 的第一次写访问时的信息，忽略了潜在隐藏的数据竞争。

本文针对上述问题做出改进，对于每一个共享内存单元的读写操作，都会保留全部的访存记录。但不像 Djit+一样通过对共享内存的读写操作去维护一个 n 维向量，而是对每个内存访问单元的读或写操作维护一个 $map\langle key, value\rangle$，$key$ 中存放线程号，$value$ 中存放一个数组指针，数组指针中第 1 个元素存放一个锁集变量 L_tcount，表示线程当前持有锁数量的情况。L_tcount 值可以为 0，1 或 m，分别代表当前线程拥有 0 个锁，1 个锁，m 个锁，这种情况下值为 m 的情况一般是多个线程同时访问某个共享读变量。从第 2 个元素开始存放线程对共享变量访问的时钟值，按访问先后顺序依次往后存放，当新的线程第一次访问此共享变量时，在共享内存单元中添加一条 map 信息，某个共享内存单元中 map 的条目总数与有多少个线程访问过此共享变量保持一致。AsampleLock 算法整体的时间复杂度约为 $O(2pk\log l)$，p 表示涉及数据竞争的访问事件对数量，k 为共享变量的总数，l 为锁集总数量。

3.4 AsampleLock 算法的描述

跨线程采样处理的流程如算法 1 所示。

算法 1 跨线程采样处理

```
输入：多线程程序 m；采样率 r；频值 n，整个程序只监控 n 对函数对；
      最近 n-1 次执行的映射（从函数对到计数器）pairs
初始化：
T 表示一个线程到一个函数的空映射；
Bool 表示一个线程到一个布尔值的映射；
TP=pairs(深拷贝)// TP 是一个线程到 pairs 副本的映射
  for each thread t∈m do
    T(t):=∅
    Bool(t):=true
    TP(t):=pairs //深拷贝
  end for
Function onEnterFunc(Thread t,Func f)
  let T(t):=f and Bool(t)=false
    for each thread t'∈m,t≠t' do
        pair:=⟨f,T(t')⟩
        if pair∉TP(t) then
          Bool(t):=true
          Bool(t'):=true
          TP(t):=TP(t)∪{pair,1}
        else
  TP(t):=TP(t)∪{pair,Counter(TP,pair)+1}
    if counter(pair,TP(t)) satisfies r then
      Bool(t):=true
      Bool(t'):=true
    else
      Bool (t)=false
    end if
    end if
  end for
  for each thread t∈m do
    TPairs':=TPairs' ∪T(t)
  end for
  save TPairs'
end Function
输入：TPairs'
Function onPreDetect(event Event_t,Event_t')
  if Bool(t):=true then
    if ((Event_t.variable=Event_t'.variable)∩(Event_t.write∪
    Event_t'.write≠∅) == true)
    Save ⟨Event_t,Event_t'⟩
  endif
  endif
end Function
```

读访问处理的流程如算法 2 所示。

算法 2 读访问处理

```
输入：⟨Event_t,Event_t'⟩；线程 t 的向量时钟 C_t；本地时钟值 C_t[t]
      L_tcount：当前线程拥有锁的数量，可为 0,1,m；
      Wrmap_x：变量 x 的写访问记录 map；
      Rdmap_x：变量 x 的读访问记录 map；
      Rdlist，Wrlist：当前线程 t 读写变化的时钟向量的数组
for t'id= 0 to Wrmap_x.size in //遍历写 map 记录
  Wrlist=Wrmap_x·[t'id]//根据线程号获得一条 map 记录
    for i=1 to Wrlist.size in//遍历 map 中的 value 数组，即相关向
    量时钟的变化情况
```

```
        if tid≠t′id ∩ Ct[t′id]<Wrlist[i]
            if (Wrlist[0]=0 ∪ Lt count=0) ∪ (Wrlist[0]≠m ∪
            Ltcount≠m∪Wrlist[0] ≠Ltcount)
            Report a write-read race.
            end if
        end if
for t′id=0 to Rdmapx. size in //遍历读 map 记录
        Rdlist=Rdmapx.[t′id]
        for i=1 to Rdlist. size in//遍历 map 中的 value 数组,即相关向
        量时钟变化情况
      if tid≠ t′id ∩ Ct[t′id]<Rdlist[i]
      shareReadx=true;
      end if
      end for
end for
```

写访问处理的流程如算法 3 所示。

算法 3 写访问处理

```
for t′id=0 to Wrmapx. size in
  Wrlist=Wrmapx.[t′id]
    for i=1 to wrlist. size in
      if tid≠ t′id∩Ct[t′id]<Wrlist[i]
          if (Wrlist[0]=0 ∪ Lt count=0) ∪ (Wrlist[0]≠m ∪
            Ltcount≠m∪Wrlist[0]≠Ltcount)
          Report a write-write race.
          end if
        end if
    end for
end for
for t′id=0 to Rdmapx. size in //遍历读 map 记录
  Rdlist= Rdmapx.[t′id]
    for i=1 to Rdlist. size in//遍历 map 中的 value 数组,即相关向量
    时钟变化情况
      if tid≠t′id∩Ct[t′id] < Rdlist[i]
          if (Rdlist[0]=0 ∪ Lt count=0) ∪ (Rdlist[0]≠m ∪
          Ltcount≠m∪Rdlist[0]≠Ltcount)
          Report a read-writerace.
          end if
        end if
    end for
end for
```

算法 2 和算法 3 描述了读操作和写操作的处理过程。write-read 竞争、read-write 竞争、write-write 竞争虽然产生机理不同,但处理过程相同。处理过程以 write-read 竞争检测为例:当前线程为读操作,遍历共享内存单元写访问记录,比较向量时钟值的大小,如果 $tid\neq t'id\cap C_t[t'id]<Wrlist[i]$,则表示两次读写访问不具有 nolock-hb 关系,处于并发状态;然后判断处于并发状态的两个线程的锁模式情况,判断条件为 if $(Wrlist[0]=0\cup L_tcount=0)\cup(Wrlist[0]\neq m\cup L_tcount\neq m\cup Wrlist[0]\neq L_tcount)$,即比较写访问历史中的线程和当前线程各自拥有的锁集数量。若满足上述条件,则报告一个 write-read 数据竞争。若满足条件 $tid\neq t'id\cap C_t[T'id]<Rdlist[i]$,则标记变量 x 为共享读变量。

以图 5 为例说明数据竞争的判定方法。对于每一个共享内存单元来说,会保留全部的访存记录,即保留全部的 map 记录。线程 t、锁集 l、时钟 c 三者的组合为一条 map 记录,记作 $map\{t,\langle l,c\rangle\}$。表示线程 t 当前的时钟值为 c,且线程 t 当前拥有 l 个锁。当且仅当 map 中的时钟值小于或等于相应的向量时钟值时,map 与向量时钟 V 具有 nolock-hb 关系:

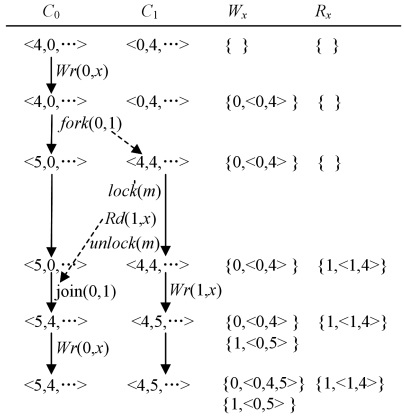

图 5 基于 map 的读写更新

Fig. 5 Read-write update based on map

$map\{t,\langle l,c\rangle\}\leq V(t)$ iff $c\leqslant V(t)$

对于 x 的第一次写操作,Wx 中更新一条写访问记录为 $\{0,\langle 0,4\rangle\}$,表示线程 C_0 当前的向量时钟值为 4,且没有被锁保护。当 C_1 对 x 进行读操作时,遍历写访问记录,此时 $Wrlist[1]=4\leqslant C_1[0]=4$,$Wrlist[1]$代表 Wx 中的$\{0,\langle 0,4\rangle\}$这条 map 记录中 *value* 值中的第 2 个元素值 4,即表示两次读写访问存在 nolock-hb 关系,不会产生写读数据竞争,在 R_x 中添加一条记录$\{1,\langle 1,4\rangle\}$。C_1 对 x 进行写操作,遍历 x 的读访问记录,线程号相同,处于非并发状态。再遍历写访问记录$\{0,\langle 0,4\rangle\}$,此时 $Wrlist[1]=4\leqslant C_1[0]=4$,表示两次写访问存在 nolock-hb 关系,不会产生写写数据竞争,然后线程 C_1 更新共享变量写访问记录,在 W_x 中添加一条记录$\{1,\langle 0,5\rangle\}$。C_0第二次写 x,遍历写访问操作,写访问记录为$\{1,\langle 0,5\rangle\}$,$C_0[1]=4<5=Wrlist[1]$,两次访问不存在 nolock-hb 关系,再判断锁的数量情况,C_0 和 C_1 都没有被锁保护,满足数据竞争判定条件则可报告写写竞争,线程 C_0 更新共享变量,写访问记录为$\{0,\langle 0,4,5\rangle\}$。遍历读访问记录$\{1,\langle 1,4\rangle\}$,$Rdlist[1]=4\leqslant C_0[1]=4$,不会发生读写数据竞争。

4 实验结果与分析

第 3 节分析了 AsampleLock 算法,在此基础上实现了原型系统 AsampleLock。本节将该系统与具有代表性的 FastTrack 算法[9]、LiteRace 算法[19]、Multilock-HB[13]算法进行时间开销以及数据竞争检测率的对比实验分析。

4.1 实验环境与测试程序

硬件环境如下:CPU 为 Intel(R)Core(TM) i7-7700 @ 3.60GHz(8 核,8 线程),内存为 32.00GB。软件环境如下:操作系统为 Ubuntu 12.04 64 位,插桩框架为 Intel pin-2.12-58423-gcc.4.4.7-linux,编译环境为 gcc-4.9.4,g++-4.9.4,编程语言为 C++11。

从文献[25]中选择了 3 个测试集和本文中图 2 的示例程序,从 Parsec-3.0[16]中选择了 6 个大型多线程程序测试集,以上 10 个程序组成了 Benchmarks。本实验采用此测试集在双

线程以及更多线程上对本文算法的性能和有效性进行测试评估，每个测试程序插桩运行 20 次，以准确收集运行时开销和数据竞争检测率。

4.2 实验结果与对比分析

我们在 PIN 上分别实现了 AsampleLock(AL)算法、FastTrack(FT)算法、LiteRace(LR)算法、Multilock-HB 算法(ML)，并运用表 1 中的数据集来验证 AL 算法的性能，对检测时间开销和竞争检测率进行评估。将 AL 算法的采样率和 n 值(确定 n 个频繁函数对)分别设置为 10/100 和 2，LR 的采样率同样也设置为 10/100。从表 2 可以看出，在 10 个测试集中，无论是大型程序还是小程序，AL 算法的表现与 FT 一样稳定。从检测时间上来看，AL 算法的时间开销与 FT 算法相比整体平均降低了 8%，与 LR 算法相比整体上平均降低了 6%。对于大规模的程序，ML 需要大量的内存和时间，原因是对于 n 个线程的程序，ML 算法的时间复杂度较大[26]，为 $O(nm\log l)$，在本实验中，ML 算法的检测时间开销约是 FT 算法和 LR 算法的 1.6 倍，约是 AL 算法的 1.7 倍。实验结果表明，本文算法的检测时间开销明显优于其他 3 种算法。

表 1 测试程序及属性

Table 1 Benchmarks and their properties

测试程序	线程数	代码行数
example	2	68
mysql_169_extract	2	106
circular_list	4	176
log_proc_sweep	4	99
bodytrack	8	143288
blackscholes	7	2747
fluidanimate	4	5795
streamcluster	5	2531
fft	8	1074
radix	7	968

表 2 实验结果

Table 2 Overall experimental results

Benchmark	时间开销(插桩后时间/插桩前时间)				竞争数量			
	FT	LR	ML	AL	FT	LR	ML	AL
example	308	231	367	217	6	7	8	8
mysql_169_extract	166	162	182	150	11	8	12	10
circular_list	175	173	1387	159	37	60	78	37
log_proc_sweep	144	138	1215	136	15	14	27	28
bodytrack	5608	6118	9689	4867	30	33	25	36
blackscholes	286	266	1620	240	0	0	0	0
fluidanimate	91039	88328	140714	84145	48	0	101	62
streamcluster	17	16	51	13	8	9	308	12
fft	18	18	108	15	12	20	27	26
radix	37	34	178	30	48	46	80	59

从漏报率方面来看，fluidanimate 是用于产生动画效果的程序，分析源码得出此程序对 malloc 的调用很少，并消除了大量内存分配中不必要的同步。在设置相同参数检测此程序时，LR 算法检测出的数据竞争的数目为 0，AL 算法的数据竞争检测率比 FT 算法高 29%。bodytrack 是有关计算机视觉领域方面的程序，针对此程序，AL 算法以较低的时间开销检测出比 ML 算法多 44%的数据竞争错误。针对程序 circular_list，ML 算法和 LR 算法的数据竞争检测率比 AL 算法和 FT 算法高，AL 算法以较低的时间开销检测出与 FT 算法相同的竞争数，ML 算法的时间开销最大。分析相关代码可知，AL 和 FT 算法确实存在一定漏报现象。FT 算法采用纯动态检测算法，只能检测某一特定执行路径上的数据竞争，检测覆盖率较低，且不保存完整的历史访存记录，因而导致漏报。AL 算法采用 nolock-hb 关系来降低算法对线程调度的敏感程度，并对 FT 算法进行了优化，保留每一个共享单元的全部访存记录，从而减少对潜在数据竞争的漏判。但 AL 算法在采样内存访问对时，保留的函数对采用突发采样策略，这会导致一定的漏报。分析 circular_list 源程序可知，4 个线程都访问了 process(list)函数，若其中两个线程连续多次访问此函数，为了减少检测数据竞争的时间开销，AL 算法不对已经收集到的函数对进行连续采样，若发现一函数对已存在于线程局部映射中，则由采样率和突发采样策略共同判定舍弃或保留，当判断为舍弃时，存在漏判数据竞争的现象。此漏判现象是一种特殊的情况，例如：来自并发线程 t_1 和线程 t_2 的函数对 $\langle f_1, f_2\rangle$ 已经在函数对集合中，若下次线程 t_1 和线程 t_2 又并行地访问了函数 f_1，f_2，则在 AL 算法依据采样率和突发采样策略舍弃此函数对的情况下，AL 无法再次统计此函数对，但 AL 算法会把第一次保留的 $\langle f_1, f_2\rangle$ 输入到检测阶段，若存在数据竞争，AL 会报告函数 f_1 的 m 行和 f_2 的 n 行存在的数据竞争错误。在上述情况下，LR，FT，ML 算法均存在重复报告该数据竞争的可能性。若在 m 行和 n 行这两条语句上采用同步机制，则可消除此数据竞争，且 AL 算法漏报的重复数据竞争也可一并消除。从本文中的 10 个测试集来看，circular_list 程序的漏报并不影响整体的数据竞争检测率，本文算法检测出的数据竞争总数与 FT 算法相比增加了 27%，与 LR 算法相比增加了 39%。

从误报率方面来看，本文算法经过预竞争检测阶段的判断可减少大量误报，在检测阶段通过 map 保留所有读写记录，以更加精确地检测出数据竞争。在检测程序 streamcluster 时，AL 与 LR 算法检测出的竞争数分别为 12 和 8。但 ML 算法检测出 308 个数据竞争，分析相关源码和 ML 算法检测出的数据竞争报告可知，检测报告中存在对相同的共享内存位置发出重复数据竞争警告和误报现象。分析 AL，FT，LR 的检测报告和相关源码可知，符合数据竞争定义的真实数据竞争数为 12。

结束语 本文提出了一种利用跨线程采样技术，基于锁模式和 FastTrack 算法的动态数据竞争混合检测算法 AsampleLock。该方法采用 nolock-hb 关系来判断访问事件的并发关系，并基于锁模式进行动态数据竞争检测。在此基础上，AsampleLock 算法采用 map 记录共享变量的读写信息以降低漏报率和误报率。采用多线程测试集 Parsec 测试其有效性，并与 FastTrack 算法、LiteRace 算法、Multilock-HB 算法进

行对比分析。结果显示，AsampleLock 算法消耗的总时间开销比 FastTrack 算法少 8%，比 Multilock-HB 算法少 42%，且检测出的数据竞争数目比 LiteRace 算法增加了 39%。本文算法在时间复杂度和检测率上都优于其他 3 种算法，但本文算法未能区分出恶性或良性数据竞争，这是下一步的研究方向。

参 考 文 献

[1] LU S W, ZHOU Y. A study of interleaving co-verage criteria [C]// The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. ACM, 2007: 533-536.

[2] LU S. How to deal with concurrent bugs in multithreaded programs[J]. Communication of China Computer Society, 2013, 9(2): 20-27.

[3] LU S, PARK E, SEO, et al. Learning from mistake-s: A comprehensive study on real world concurrency bug characteristics [C]// Proc. ASPLOS. 2008: 329-339.

[4] JACKSON J. Nasdaq's Facebook glitch came from 'RaceConditions'[EB/OL]. [2012-03-24]. https://www.computerworld.com/article/2504676/nasdaq-s-facebook-glitch-came-from--race-conditions-. html.

[5] JIMENEZ M, PAPADAKIS M, LE TRAON Y. Vulnerability prediction models: A case study on the Linux kernel[C]// 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation(SCAM). 2016: 1-10.

[6] ENGLER D, ASHCRAFT K. RacerX: effective, static detection of race conditions and deadlocks[J]. ACM SIGOPS Operating Systems Review, 2003, 37(5): 237-252.

[7] VOUNG J, JHALA R, LERNER S. RELAY: Static Race Detection on Millions of Lines of Code[C]// Joint Meeting of the European Software Engineering Conference & the ACM Sigsoft Symposium on the Foundations of Software Engineering. ACM, 2007: 205-214.

[8] POZNIANSKI E, SCHUSTER A. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs[C]// International Parallel & Distributed Processing Symposium. IEEE, 2003: 179-190.

[9] CORMAC F S N F. FastTrack: Efficient and precise dynamic race detection[J]. ACM SIGPLAN Notices, 2009, 44(6): 121-133.

[10] CAI Y, CHANW K. LOFT: Redundant Synchronization Event Removal for Data Race Detection[C]// IEEE International Symposium on Software Reliability Engineering. IEEE, 2011: 160-169.

[11] YANG Z, YU Z, SU X, et al. RaceTracker: Effec-tive and efficient detection of data races[C]// 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). IEEE Computer Society, 2016: 293-300.

[12] XIE X, XUE J, ZHANG J. Acculock: Accurate and Efficient Detection of Data Races[C]// IEEE/ACM International Symposium on Code Generation & Optimization. IEEE, 2011: 201-212.

[13] XIE X, XUE J, ZHANG J. Acculock: accurate and efficient detection of data races[J]. Software: Practice and Experience, 2013, 43(5): 543-576.

[14] YU M, YOO S K, BAED H. SimpleLock: Fast and Accurate Hybrid Data Race Detector[C]// International Conference on Parallel & Distributed Computing. IEEE, 2014: 50-56.

[15] YU M, LEE J, BAE D, et al. AdaptiveLock: Efficient Hybrid Data Race Detection Based on Real-World Locking Patterns[J]. International Journal of Parallel Programming, 2019, 47(7): 805-837.

[16] BIENIA C. Thesis: Benchmarking modern multiprocessors[D]. Princeton University, 2011.

[17] LAMPORT L. Time, clocks, and the orde-ring of events in a distributed system[J]. Communications of the ACM, 1978, 21(7): 558-565.

[18] GUO Y, CAI Y, YANG Z. AtexRace: across thread and execution sampling for in house race detection[C]// Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017: 315-325.

[19] MARINO D, MUSUVATHI M, NARAYANASAMY S. LiteRace: effective sampling for lightweight data race detection[C]// ACM Sigplan Notices. ACM, 2009: 134-143.

[20] BOND M D, COONS K E, MCKINLEY K S. PACER: proportional detection of data races[C]// ACM Sigplan Notices. ACM, 2010: 255-268.

[21] YU M, BAE D H. SimpleLock+: fast and accurate hybrid data race detection[J]. The Computer Journal, 2016, 59(6): 793-809.

[22] LUK C K, COHN R S, MUTH R, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation [C]// ACM Sigplan Notices. ACM, 2005: 190-200.

[23] ZHANG Y, LIANG Y N, ZHANG D W, et al. An Effective Approach of Detecting Data Race in Concurrent Programs[J]. Journal of Computer Applications, 2019, 39(1): 67-71.

[24] WANG X F. Research on Dynamic Data Race Detection[D]. Wuhan: Huazhong University of Science And Technology, 2016.

[25] YU J, NARAYANASAMY S, PEREIRA C, et al. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs [C]// Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. ACM, 2012: 485-502.

[26] YU M, PARK S M, CHUN I, et al. Experimen-tal Performance Comparison of Dynamic Dat-a RaceDetectionTechniques [J]. ETRI Journal, 2017, 39(1): 124-134.

LI Meng-ke, born in 1996, postgraduate. Her main research interests include high performance computing and so on.

WANG Lei, born in 1977, professor, master supervisor. His main research interests include research and development of high performance computing and domestic independent and controllable basic software.