高技术通讯 2014 年 第24 卷 第7 期:669 ~ 676

doi:10.3772/j.issn.1002-0470.2014.07.002

基于可行序的数据竞争检测①

李 磊②*,**,*** 陈云霁*,** 章隆兵*,**,**** 肖俊华*,**,****

(*计算机体系结构国家重点实验室 北京 100190)

(**中国科学院计算技术研究所 北京 100190)

(***中国科学院大学 北京 100049)

(****龙芯中科技术有限公司 北京 100190)

摘 要 为了在并行程序的单次执行中找到更多的数据竞争,提出了用可行序关系替代传统的"happens-before"序关系来动态地实现数据竞争预测的算法。该算法认为:从技术上讲,如果在观测到的执行轨迹中,两个临界区之间没有可行序的关系,那么这两个临界区的顺序可以被颠倒以构造出其他的执行轨迹;通过判断可行序关系来分析这些构造出来的执行轨迹,就可以找到单次执行中未暴露出来的可能的数据竞争;所有构造出来的执行轨迹中的数据竞争,可以在 $O(an)$ 的时间内全部检测出来,其中 n 为程序中所有访存操作的个数,a 为每个共享地址上的最大锁集合数。在 Java Grande 测试程序集上的实验结果说明,上述算法可以找到其他动态检测数据竞争的方法找不到的数据竞争,而且算法时间也完全符合理论上的 $O(an)$ 时间复杂度。

关键词 数据竞争,并行程序调试,发生前(HB),可行序

0 引 言

随着多核处理器的广泛使用,并行程序也日益被普遍采用,以享受由多核处理器带来的性能提升。然而,与串行程序不同,并行程序的执行具有不确定性,使得并行程序的调试(debugging)变得极其困难。在并行程序的众多种类的错误(bug)中,最常见的一种就是数据竞争[1]。数据竞争检测是检测并行程序 bug 过程中的重要一环。已有的数据竞争检测方法有静态方法和动态方法,静态方法可以找到所有的数据竞争,但会有极多的误报,而动态方法只能找到单次执行中暴露出的数据竞争。本研究的目标是在并行程序的单次执行中找到更多的数据竞争,为此引入了可行序的概念,并提出了用可行序关系替代传统的"发生前"(happens-before,HB)序关系来动态地实现数据竞争检测的算法,实验表明,该算法可找到其他动态方法找不到的数据竞争,而且时间复杂度很低。

1 基本概念

对于来自不同线程的一对访存操作,如果它们访问的地址相同,而且其中至少有一个是写操作,这对操作就称为一对冲突操作[2]。对于一个并行程序,如果该程序存在一个所有操作满足程序序的全序,在全序中有一对冲突操作时是邻的,那我们就认为该程序有一个数据竞争[3]。数据竞争被认为是有害的,因为它们时常给程序带来出乎程序员意料的行为[4]。

已有数据竞争检测方法中的静态检测方法[5-10]静态地分析并行程序的源码,并在程序所有可能的执行轨迹中寻找数据竞争。虽然这类方法可以找到所有真实的数据竞争,但是由于一个程序可能的执行轨迹的数量是随着锁的数量呈指数上涨的,因而这些方法很有可能会碰到状态爆炸的问题。此外,静态方法缺少动态的数据流和指令流信息,因而只能极其保守地推断访存指令之间的序关系,从来无

① 国家"核高基"科技重大专项课题(2009ZX01028-002-003,2009ZX01029-001-003, 2010ZX01036-001-002,2012ZX01029-001-002-002),国家自然科学基金(61221062, 61100163, 61133004, 61173001, 61232009, 61222204),863 计划(2012AA010901, 2012AA011002, 2012AA012202, 2013AA014301)资助项目。

② 男,1991 年生,博士生;研究方向:计算机系统结构,并行计算;E-mail: lilei-cpu@ict.ac.cn

(收稿日期:2013-09-27)

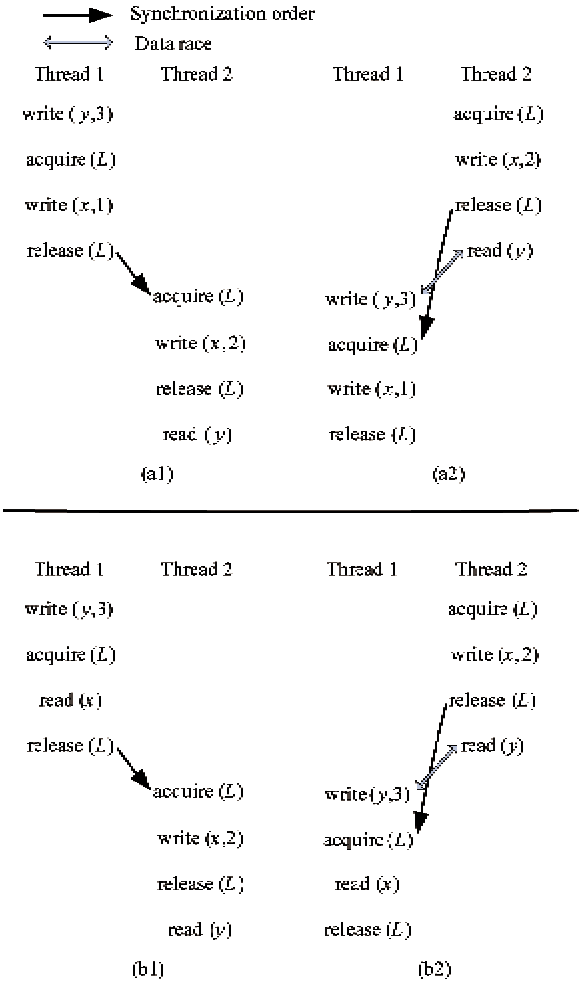
法避免地带来极多的误报[7,9,10]。

由于在静态方法给出的众多误报中去筛选真实的数据竞争是一个非常消耗人力的工作，于是研究者提出了动态的方法来避免这样的消耗[11-16]。具体来说，这些方法在程序的一次执行中收集这次执行的访存指令和同步指令的轨迹，然后根据收集到的执行轨迹找出在这次执行中没有被同步指令安排好顺序的冲突操作。虽然这些方法可以有效地找到单次执行中暴露出的数据竞争，但是在此次执行中没有暴露的数据竞争却无法被这些方法检测出来。然而，正如前面提到的，一个并行程序可能的执行轨迹是随着锁的数量呈指数上涨的，我们无法把一个并行程序所有的执行都检测一次。因此，如何在单次的执行中找到更多的数据竞争是目前动态方法面临的最重要的挑战。

本文提出，可通过观测到的执行轨迹来构造新的执行轨迹，以便在单次执行中找到更多的数据竞争。从构造出来的执行轨迹中，我们可以找到观测到的轨迹中未暴露出来的数据竞争。具体来说，本文的主要出发点是：在同一锁保护的两个关键区间内，如果没有写后读（read-after-write，RAW）的依赖，这两个关键区间的执行顺序就可以被颠倒以构造新的执行轨迹。即使一对冲突操作在观测到的执行轨迹中被两个关键区间隔开了，一旦这两个关键区间的访问次序颠倒，这对冲突操作有可能在颠倒后的执行路径中是一个数据竞争。

如图 1 所示，x 和 y 是两个共享变量，其初始值为 0，L 是一个锁变量。执行轨迹（a1）是一个观测的执行轨迹，其中两个关键区间包含写后写（write-after-write，WAW）的冲突事件。轨迹（a1）中是没有数据竞争的，因为任何一对冲突操作都被合理地同步了。然而，将（a1）中的关键区间执行的顺序颠倒，我们可以构造执行轨迹（a2）。在（a2）中，我们可以找到一个数据竞争 write（y,3）和 read（y）。同样，执行轨迹（b1）是一个观测的执行轨迹，其中两个关键区间包含读后写（write-after-read，WAR）的冲突事件。通过颠倒其中关键区间执行的顺序，我们可以构造执行轨迹（b2），并在（b2）中找到数据竞争 write（y,3）和 read（y）。

基于上述分析，本文提出了一种叫做可行序的概念。可行序是一种比 Lamport 提出的“发生前”（HB）更弱的序关系[17]。通过判断两个冲突操作直接是否有可行序的关系，我们可以判断这对冲突操作是否会在某些构造出的执行中暴露为一个数据竞争。



执行轨迹 a1 和 b1 是两个观测到的执行轨迹，都没有数据竞争，但是没有被 a1 和 b1 暴露的数据竞争可以通过构造执行轨迹 a2 和 b2 探索出来

图 1 数据竞争探索的两个例子

大致地说，一个操作 u 在另一个操作可行序之前，条件是下面任何一项成立：

1）操作 u 和 v 来自同一个线程，而且 u 在 v 的程序之前。

2）u′和 u 是一组锁获取和锁释放，v 和 v′是另外一组同锁地址的锁获取和锁释放。且关键区间 u′…u 直接有一个操作 w，关键区间 v…v′之间有一个操作 r；w 和 r 是一组冲突操作，且 w 为写，r 为读。

3）存在一个操作 t，使得 u 在 t 可行序之前，且 t 在 v 可行序之前。

我们发现，如果两个冲突操作没有可行序的关系，且它们不被同一个锁保护，那么它们之间就极有可能有一个数据竞争或者死锁。因此，数据竞争可以通过判断可行序关系有效地探索出来。

我们的方法除了能有效找到观测执行中未出现的数据竞争，还有着很低的时间复杂度。我们的算

法探索数据竞争的复杂度仅为 $O(an)$,其中 a 是在一个给定的共享地址上最多的锁集合的个数,n 是所有访存操作的总数。综上所述,本文提出了一种称为可行序的新的序关系,通过判断冲突操作之间的可行序关系,我们可以探索出在单次执行中未曾暴露出的数据竞争。此外,我们的算法的复杂度足以能处理真实的大规模的并行程序。最后,我们实现了一个实时的软件数据竞争检测器,实验数据也说明了我们方法快速性和高效性。

2　可行序

这一节将形式化地描述可行序,并将可行序与已有的 happens-before 序关系进行对比。我们知道,一个并行程序是有数个线程组成的,每个操作都是在每个线程上独立执行的。操作的类型包括:线程创建/加入,对内存的写/读,对锁的获取/释放以及栅栏。在这里,我们只考虑内存的写/读操作以及锁的获取/释放操作,其他的操作可以直接视为可行序的边。

一个程序的执行轨迹就是一个满足程序序的所有线程的所有操作的全序。每个操作都可以表示成一个三元组(p,o,i),其中 p 表示该操作在的线程的 id,o 是该操作的类型(读、写、锁获取、锁释放),i 是这个该操作在全序中的索引。

定义 1:发生前(happens-before,HB)序关系是最小的闭包关系,它使得在一个执行轨迹中 u 在 v HB 序之前,条件是下面的任何一项成立:

1) 程序序:u 和 v 来自同一个线程,且在执行轨迹中 u 在 v 之前。

2) 锁释放获取序:u 是一个锁释放,v 是执行轨迹中接下来的对同地址锁的锁获取。

如果两个操作直接没有 HB 关系,那么这两个操作就是并发的。如果这两个操作在该轨迹中是并发的,我们称一个数据竞争在一个执行轨迹中被"暴露"出来。因此,只要对每一对冲突操作进行检查,看它们之间是否存在 HB 序关系,就可以找到单次执行中暴露的数据竞争。为了在单次执行中找到更多的数据竞争,我们提出了可行序关系。

定义 2:可行序关系是最小的闭包关系,它使得在一个执行轨迹中 u 在 v 可行序之前,条件是下面的任何一项成立:

1) 程序序:u 和 v 来自同一个线程,且在执行轨迹中 u 在 v 之前。

2) 写后读的锁释放获取序:u 是一个锁释放,v 是执行轨迹中在对同地址锁的锁获取。u′…u 和 v′…v 是对应的关键区间。存在一对冲突操作 w 和 r,w 是关键区间 u′…u 中的一个写操作,r 是关键区间 v′…v 中的一个读操作。简单来说,一个锁释放获取边是一个可行序的边,条件是对应两个关键区间中含有一对写后读(read-after-write,RAW)的冲突操作。

作为一个闭包的关系,可行序是一个偏序,所以可行序可以很容易地在执行中去判断。此外,从可行序的规则 2)中我们可以看出,锁释放获取边是一个可行序的边,仅当对应两个关键区间中含有一对 RAW 的冲突操作。因此,可行序是比 HB 序更弱的一种关系,也就是说,如果在可行序之前,那么 u 一定在 v HB 序之前。

在下节介绍重要的概念"可行数据竞争"之前,我们先给出以下两个定义:

定义 3:对于一个执行轨迹中的两个冲突操作 u 和 v,如果存在一个关键区间包含 u,并有另一个在同一个锁上的关键区间包含 v,我们就称 u 和 v 在该次执行中被合理保护。

定义 4:对于一个执行轨迹中的两个冲突操作,如果它们之间没有可行序关系,而且没有被合理保护,我们就称这两个冲突操作在该次执行中是一个可行数据竞争。

如果在一个执行轨迹中,两个冲突操作之间没有 HB 的关系,那么这两个操作之间就有一个数据竞争。接下来,我们会发现,如果两个冲突操作之间没有可行序的关系,那么这两个操作极有可能在一个构造出来的执行轨迹中是一个数据竞争或者死锁。

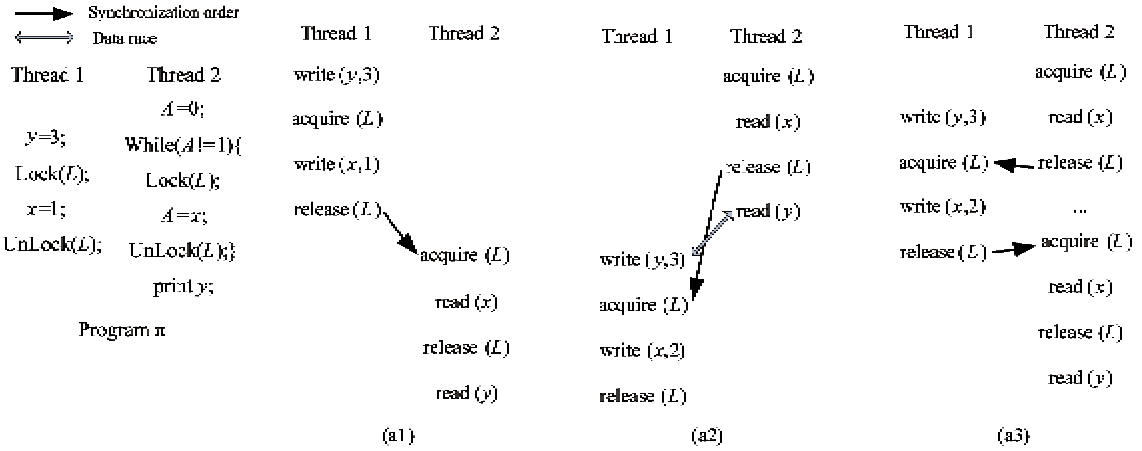
3　可行数据竞争分析

在本节中,我们给出一些较为复杂的执行轨迹的例子来更好地解释可行数据竞争。通过这些例子,我们可以看出,可行序定义中的每个规则都是不可缺少的。

正如在前面提到的,在一个执行轨迹中,如果两个关键区间之间没有可行序的关系,这两个关键区间可以颠倒执行的顺序来探索数据竞争。反之,如果两个关键区间有可行序,强行地颠倒它们执行的顺序很有可能带来数据竞争的误报。

如图 2 所示,执行轨迹(a1)是程序的一个观测

到的执行轨迹。在程序中,线程 2 重复地执行它的关键区间,直到共享变量被线程 1 写入。在(a1)中,两个关键区间之中有一对写后读(RAW)的冲突操作,根据定义 2 的规则 2),这两个关键区间是有可行序的。一旦将这两个关键区间的执行次序颠倒过来,构造的执行轨迹(a2)是不可能称为程序的执行轨迹的。因为线程 2 会不断地再次执行它的关键区间,并产生执行轨迹(a3)。而执行轨迹(a3)中是没有数据竞争的。所以,颠倒两个有着可行序的关键区间之间的执行顺序构造的执行轨迹(a2)中的数据竞争是个误报。故此,在通过可行序去判断数据竞争的过程中,我们只会通过颠倒没有可行序的关键区间来避免类似以上例子的误报。



颠倒 a1 中两个有可行序关系的关键区间的执行顺序会带来数据竞争的误报

图 2 执行轨迹 a1 是观测到的执行轨迹

除了通过找到构造执行轨迹找到观测执行中未暴露的数据竞争,可行序关系还能找到观测执行中未暴露出的死锁。如图 3 所示,在观测轨迹(b1)中,我们可以发现一个可行数据竞争。但实际上,无论如何去颠倒观测轨迹(b1)中的关键区间执行顺序,我们都无法让 x 地址上的冲突操作彼此相邻。但是,通过构造执行轨迹(b2),我们可以很容易地找到一个未暴露出的死锁。

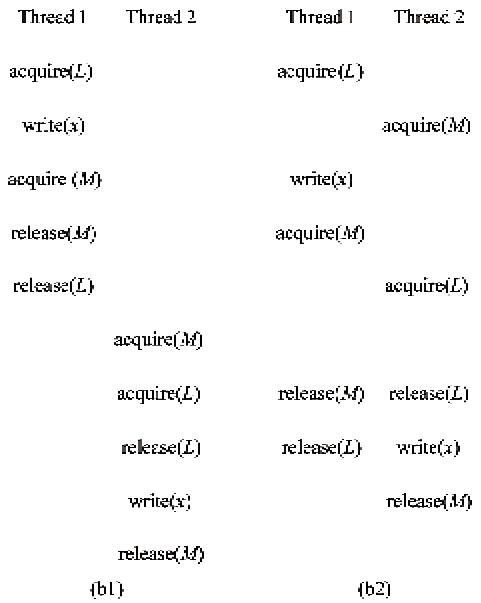


图 3 通过执行轨迹(b1)的可行数据竞争,可以通过构造轨迹(b2)来找到一个死锁

总的来说,上面的几个例子说明了可行序定义中的每条规则都是不可缺少的,并且,一个可行数据竞争除了是由数据竞争引起之外,还可能是由于死锁的缘故。

4 算法实现以及时间复杂度分析

根据定义 4,一个可行数据竞争是指未被合理保护的一对没有可行序关系的冲突操作。为了检测数据竞争,我们需要检测每一对未被合理保护的冲突操作之间的是否有可行序关系。跟大多数的检测数据竞争的方法类似,我们在实现上用向量时钟来实时地表示各个操作之间可行序的边。

一般来说,线程 p 的向量时钟对每个线程 i 分配有一个整数(记作 $C_p(i)$)。这个整数用来表示每个线程在线程 p 的当前执行指令可行序之前指令的逻辑时钟。初始的向量时钟 $C_p(p) = 1$, $C_p(i) = 0(i \neq p)$。自己线程的向量时钟 $C_p(p)$ 在该线程执

行完一条锁释放或者锁获取后加1。对于线程p的操作u和线程q的操作v来说,一旦有一个可行序的边u→v被找到,对任意的i,线程q的向量时钟需要更新为 $C_q(i) = \max(C_p(i), C_q(i))$。

除了每个线程有个执行过程中实时更新的向量时钟,每个操作u也有一个向量时钟(记作 VC_u),是u在执行时所在进程的向量时钟。对两个操作u和v来说,u→v成立当且仅当 $VC_u(i) \leqslant VC_v(i)$ 对任意的都成立。

根据可行序定义中的规则2),一个锁释放获取边是一个可行序的边当对应两个关键区间中含有一对写后读(read-after-write,RAW)的冲突操作。为了在执行过程中实时去找可行序的边,每个关键区间内的写操作的向量时钟都要被记录下来。而每次执行到关键区间内的读操作是,判断是否有一个同地址的写操作已经在同一个锁保护的关键区间内被执行。如果有,我们就找到了一条可行序,并且对应读操作所在线程的向量时钟就需要被更新。

Step 1: for each write u protected by lock L
{x=u.address; $U_x^L(i) = max(U_x^L(i), VC_u(i))$ for each i; } //record write events protected in critical sections.
Step 2: for each read u protected by lock L
{x=u.address; p=u.thread; $C_p(i) = max(U_x^L(i), VC_u(i))$ for each i; } //find edges between threads.
Step 3: for each read u protected by lockset $\mathcal{L}$ (including ø)
{x=u.address; $R_x^{\mathcal{L}}(i) = max(R_x^{\mathcal{L}}(i), VC_u(i))$ for each i; } //record R_x for each lockset.
for each write u protected by lockset $\mathcal{L}$ (including ø)
{x=u.address; $W_x^{\mathcal{L}}(i) = max(W_x^{\mathcal{L}}(i), VC_u(i))$ for each i; } //record W_x for each lockset.
Step 4: for each write u protected by lockset $\mathcal{L}$ (including ø)
{ x=u.address;
for each previous lockset $\mathcal{L}'$ (including ø) on x satisfying $\mathcal{L} \cap \mathcal{L}' ==$ ø{ //consider non-properly protected events only.
If $VC_u(i) <= R_x^{\mathcal{L}'}(i)$ does not hold for some i, then report a race; //an race with previous read conflicting events.
If $VC_u(i) <= W_x^{\mathcal{L}'}(i)$ does not hold for some i, then report a race;} //an race with previous write conflicting events.
}
Step 5: for each read u protected by lockset $\mathcal{L}$ (including ø)
{ x=u.address;
for each previous lockset $\mathcal{L}'$ (including null set) on x satisfying $\mathcal{L} \cap \mathcal{L}' ==$ ø{ //consider non-properly protected events only.
If $VC_u(i) <= W_x^{\mathcal{L}'}(i)$ does not hold for some i, then report a race;} //an race with previous write conflicting events.
}
Step 6: for each reported race (u, v) (u is executed before v)
{ p=v.thread; $C_p(i) = max(C_p(i), VC_u(i))$ for each i; } //Consider each race as a edge to handle multiple races.

图4　可行数据竞争检测算法

基于可行序来检测数据竞争的具体算法见图4。每个地址 x 都有一个向量时钟 U_x^L,来表示被锁 L 保护的对 x 的写操作中最大的向量时钟。根据此算法的第1和第2步,U_x^L 被初始化成(0,…,0);每次有在 x 上一个写操作u被执行时,如果u是被锁 L 保护的,U_x^L 被更新为 $U_x^L(i) = \max(U_x^L(i), VC_u(i))$(对任意 i)。值得注意的是,如果u被多个锁保护,其他锁对应的 U_x 也需要被更新。另一方面,每次有在 x 上一个读操作u被执行时,如果u是被锁 L 保护的,u所在线程p的向量时钟需更新成 $C_p(i) = \max(C_p(i), U_x^L(i))$(对任意 i),因为之前的写操作到u有一条可行序的边。值得注意的是,如果u被多个锁保护,其所在线程的向量时钟需对每个锁进行更新。

除了根据每条可行序去更新向量时钟的同时,我们还要比较每对未被合理保护的冲突操作的向量时钟以检测可行数据竞争。为了判断一对冲突操作是否被合理保护,我们需要记录每个操作被哪几个锁保护起来了。对每个操作u,其锁集合 $\mathcal{L}$ 指的就是保护u的所有锁的地址组成的集合。如果u不被任何锁保护,其锁集合就是空集。如果一对冲突操作的锁集合的交集是空集,那么它们就未被合理保护。我们称一个锁集合 $\mathcal{L}$ 在共享地址 x 上,如果存在一个操作u在 x 上,且u的锁集合为 $\mathcal{L}$。

在我们的算法中,每个共享地址对其上的每个锁集合都有一个 $R_x^{\mathcal{L}}$ 和 $W_x^{\mathcal{L}}$,分别用来表示在 x 上的锁集合为 $\mathcal{L}$ 的读和写操作中最大的向量时钟。如算法的第3步所示,在 x 上读操作u被执行,如果u的锁集合为 $\mathcal{L}$,$R_x^{\mathcal{L}}$ 需更新为 $R_x^{\mathcal{L}}(i) = \max(R_x^{\mathcal{L}}(i), VC_u(i))$(对任意的 i)。同样,在 x 上写操作u被执行,如果的u锁集合为 $\mathcal{L}$,$W_x^{\mathcal{L}}$ 需更新为 $W_x^{\mathcal{L}}(i) = \max(W_x^{\mathcal{L}}(i), VC_u(i))$(对任意的 i)。这样,每个共享地址的锁集合上的最大的向量时钟都可以被记录下来。

更进一步,如算法的第4步和第5步所示,在x上写操作u被执行时,如果u锁集合为 $\mathcal{L}$,u的向量

时钟需要跟所有满足 $\mathcal{L}' \cap \mathcal{L} = \phi$ 的 $R_x^{\mathcal{L}'}(i)$ 和 $W_x^{\mathcal{L}'}(i)$ 进行比较。如果 $VC_u(i) \leqslant R_x^{\mathcal{L}'}(i)$ 或者 $VC_u(i) \leqslant W_x^{\mathcal{L}'}(i)$ 对某些 i 不成立，说明写操作 u 与之前某个未被合理保护的读或者写操作之间没有可行序。因此，一个可行的数据竞争就被找到了。同样，在 x 上读操作 u 被执行时，如果 u 锁集合为 $\mathcal{L}$，u 的向量时钟需要跟所有满足 $\mathcal{L}' \cap \mathcal{L} = \phi$ 的 $W_x^{\mathcal{L}'}(i)$ 进行比较。如果 $VC_u(i) \leqslant W_x^{\mathcal{L}'}(i)$ 对某些 i 不成立，说明读操作 u 与之前某个未被合理保护的写操作之间没有可行序。因此，一个可行的数据竞争就被找到了。

最后，我们将每个可行数据竞争都当做可行序在一次执行中实时地去处理多个数据竞争。如算法的第 6 步所示，每个找到的可行数据竞争都被当做可行序并更新其对应的向量时钟。

算法复杂度分析：算法最消耗时间的部分是第 4 步和第 5 步。让参数 a 表示每个共享地址上的最大的锁集合数，p 表示总共的线程数，n 表示总共的访存操作数。算法第 4 步和第 5 步总共消耗了 $O(pan)$ 时间，因为每个操作的向量时钟最多被比较了 a 次。又由于总共的操作数是 n 个，每次比较向量时钟需要 p 的时间，我们得到算法 1 的时间复杂度是 $O(pan)$。另外，每个地址 x 上的锁集合 $\mathcal{L}$，我们需要 $O(1)$ 的空间来存 $R_x^{\mathcal{L}}$ 和 $W_x^{\mathcal{L}}$，所以算法的空间复杂度也是 $O(pan)$。

5 实验结果

我们在开源的实验平台 RoadRunner[18] 上完成了基于可行序数据竞争检测算法的实现。RoadRunner 通过在 Java 测试程序里面动态地加入字节码来对程序执行轨迹进行观测。本文算法被实现在 RoadRunner 上以动态地检测可行数据竞争。为了验证基于可行序的数据竞争检测的有效性，我们还实现了一个基于 HB 序的数据竞争检测器进行了对比。

我们的测试程序都是来自 Java Grande[19] 测试程序集，包括 crypt, lufact, sparse, series, sor, moldyn, montecarlo，和 raytracer。所有的测试程序都配置为 8 线程，并用提供的最大的数据集作为输入。在这些测试程序集中，montecarlo 和 raytracer 各有一个数据竞争，而其他的测试程序没有数据竞争。我们随机地在每个测试程序中插入了 10 个额外的数据竞争，来比较可行序和 HB 序的有效性。所有的实验都在一个 8 核主频 2.13G 的 Intel Xeon 处理器上完成。实验结果参见表 1。

表 1 与 HB 序数据竞争检测的对比

测试程序	HB（平均）	HB（10 次）	可行序（平均）	可行序（10 次）	最大锁集合数（参数）
crypt	3.2	5	10	10	2
lufact	2.6	4	10	10	2
sparse	2.8	7	8.2	10	2
series	2.2	4	9.8	10	2
sor	4.4	7	9.3	10	2
moldyn	3	7	9.0	10	2
montecarlo	3.7	6	10.5	11	2
raytracer	4.3	6	9.5	11	7

我们列出了在 10 次执行中，用 HB 序和可行序检测到的数据竞争的平均数和最大数。测试程序 montecarlo 和 raytracer 中原本就有一个数据竞争。在 10 次执行中，基于可行序的数据竞争检测可以找到所有的原有的数据竞争和后来随机插入的数据竞争。然而，基于 HB 序的数据竞争在所有的测试程序上都会漏掉了其中一些。此外，我们统计了在每个共享地址上的最大的锁集合数（算法复杂度中的参数），发现这个参数在所有的测试程序中都非常小（<10）。

由于每个测试程序都只提供了 2 – 3 个输入数据集，我们随机地在每个测试程序中都插入了一些访存操作以验证本文算法的时间复杂度。我们列出了我们在测试程序 sparse 和 sor 上插入 8000000，到 72000000 个操作的结果。图 5 为程序执行时间与插入操作数的关系图。如图 5 所示，随着操作数的

增加,每个测试程序的执行时间线性地上涨,从而验证了本文算法的复杂度的分析。

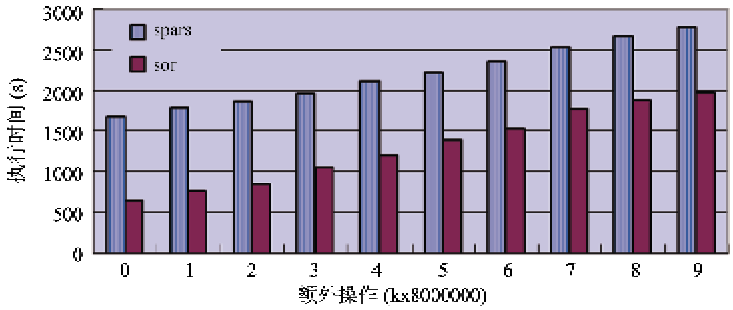


图5　程序执行时间与插入操作数的关系图(用 sparse 和 sor 作为例子)

6　结　论

数据竞争检测是检测并行程序 bug 过程中重要的一环。已有的静态检测方法可以找到所有的数据竞争但会有极多的误报,而动态方法只能找到单次执行中暴露出的数据竞争,如何在单次执行中找到更多的未暴露出的数据竞争,是动态检测方法面临的最重要的挑战。本文为了在单次执行中找到更多的数据竞争而提出的基于可行序概念的数据竞争检测方法是一种有效的方法,它通过可行序可以找到一些单次执行中其它动态方法找不到的数据竞争。这种基于可行序的数据竞争仅为 $O(n)$,n 指程序中所有操作的个数。低复杂度说明该方法可以解决真实的大规模程序。在 Java Grande 测试程序集上的实验验证了该方法的有效性。

参考文献

[1] Lu S, Park S, Seo E, et al. Learning from mistakes-a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, USA, 2008. 329-339

[2] Shasha D, Snir M. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans on Programming Language and Systems*, 1988, 10(2): 282-312

[3] Adve S V, Boehm H. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 2010, 53(8): 90-101

[4] Adve S V. Data races are evil with no exceptions: technicalperspective. *Communications of the ACM*, 2010, 53(11): 84

[5] Yahav E. Verifying safety properties of concurrent Java programs using 3-valued logic. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, 2001. 27-40

[6] Musuvathi M, Qadeer S, Ball T, et al. Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation, San Diego, USA, 2008. 267-280

[7] Engler D R, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, USA, 2003. 237-252

[8] Dwyer M B, Clarke L A. Data flow analysis for verifying properties of concurrent programs. Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst, 1994

[9] Naik M, Aiken A, Whaley J. Effective static race detection for Java. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Canada, 2006. 308-319

[10] Voung J W, Jhala R, Lerner S. Relay: static race detection on millions of lines of code. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Dubrovnik, Croatia, 2007. 205-214

[11] Christiaens M, Bosschere K D. TRaDe: Data Race Detection for Java. In: Proceedings of the International Conference on Computational Science, San Francisco, USA, 2001. 761-770

[12] Schonberg E. On-the-fly detection of access anomalies. In: Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon, USA, 1989. 285-297

[13] Flanagan C, Freund S N. FastTrack: efficient and precise dynamic race detection. In: Proceedings of the 2009

ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, 2009. 121-133

[14] Pozniansky E, Schuster A. Efficient on-the-fly data race detection in multithreaded C++ programs. In: Proceedings of 17th International Parallel and Distributed Processing Symposium, Nice, France, 2003. 287

[15] Pozniansky E, Schuster A. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 2007, 19(3):327-340

[16] Bond M D, Coons K E, McKinley K S. PACER: proportional detection of data races. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada, 2010. 255-268

[17] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of ACM*, 1978,21(7): 558-565

[18] Flanagan C, Freund S N. The roadrunner dynamic analysis framework for concurrent programs. In: Proceedings of the 9th ACM SIGPLAN – SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Toronto, Canada, 2010. 1-8

[19] Java Grande Forum. Java Grande benchmark suite. Available at http://www.javagrande.org/, 2008

Data race detection via feasible-ahead relation

Li Lei[*,**,***], Chen Yunji[*,**], Zhang Longbing[*,**,****], Xiao Junhua[*,**,****]

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

([**] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([***] University of Chinese Academy of Sciences, Beijing 100049)

([****] Loongson Technology Corporation Limited, Beijing 100190)

Abstract

To catch more data races in one-time execution of concurrent programs, a new algorithm using the feasible-ahead relation to substitute the happens-before relation to dynamially explore data races is put forward. Technically, the algorithm follows the rules below: Two critical sections without a feasible-ahead relation in an inspected execution trace can be reordered to construct conceived execution traces; The data races invoked in these conceived execution traces can be effectively explored by reasoning about the feasible-ahead relation in the inspected execution trace; The data race exploration via the feasible-ahead relation can be efficiently done with the time complexity of $O(an)$, where a is the maximum number of locksets over a shared location, and n is the total number of memory events. Both the effectiveness and efficiency of the above-mentioned algorithm were verified by the experiments on the Java Grande benchmark, and the results demonstrate that the algorithm can detect the data races other appreaches can not detect, and has a lower time complexity.

Key words: data race, concurrent program debugging, happens-before (HB), feasible-ahead