可部署的数据竞争采样检测技术

蔡 彦、 张 健、曹玲微、刘 剑

摘要：
本文提出了一种可部署的数据竞争动态采样检测技术，首先提出了基于线程本地时序的数据竞争定义，之后基于硬件断点进行采样检测。在采样率为 1%时，时间开销约为 5%，且有效性得以保证。

正文：

背景介绍：
并发缺陷在多线程程序中普遍存在，由于其只在特定的线程交替运行中产生，故而很难检测到。即使一个 Well-tested 的程序被部署后，并发错误仍然会发生。因此，在已部署程序上的并发缺陷检测是非常重要的。

本文关注并发错误中数据竞争（Data Race）的动态检测。数据竞争是指发生了如下情况：一个程序中多个线程同时访问一个共享变量且其中的一个线程执行写操作。数据竞争的发生可能会导致内存不一致，进而导致程序崩溃，或者引发其他错误甚至安全问题。

传统的数据竞争检测通常会引起很大的时间开销，例如在 Java 程序上面，最新的技术也有 400%以上的时间开销，在 C/C++程序上面则有大于 10000%的时间开销。因此，这些技术无法直接用于部署程序的数据竞争检测，因为部署程序上的错误检测，其时间开销一般要求不超过 5%。

Pacer 将传统的 Happens-before 关系结合到数据竞争的采样中。Pacer 将程序的运行分为采样区间和非采样区间，在采样区间，Pacer 完全维护 Happens-before 关系；但在非采样区间，Pacer 则仅仅检测每个内存读写在该区间的首个操作，不维护 happens-before 关系。其主要贡献是，在采样率为 r (0< r < 1) 的时候，其可以检测到数据竞争的概率也为 r，而不是 r^2 ，并且 Pacer 检测到的数据竞争数量和采样率是线性相关的。

然而，happens-before 关系的维护需要对程序中的同步操作以及每个内存维护一定的数据结构，因此，即便是在 0%采样率的情况下，Pacer 的时间开销也至少需要 30%，这远远超出了一般要求的 5%的限制，故 Pacer 很难部署到真实的环境中。

除了完全基于软件的采样外，硬件也可以被用来采样内存访问，进而检测数据竞争，例如现代的处理器几乎都提供的一些性能监控相关的接口。一般处理器都会提供两种断点：数据断点(Data breakpoint 或者 Watch point) 和指令断点 (Instruction breakpoint)。利用数据断点，我们可以在内存某处设置一个断点，当程序中某个线程访问这个内存的时候就会触发一个回调。DataCollider 就是基于硬件断点来实现内存采样的，其首先利用指令断点来采样一些内存访问指令，之后在这些指令断点触发的时候，在其要访问的内存处设置相应的数据断点。为了达到数据竞争的检测，DataCollider 会将当前线程暂停一段时间，等待其设置的内存断点被触发。一旦某个内存断点触发，则表示有另外一个线程访问了这个内存地址，如果这两个线程的两次访问冲突（即满足数据竞争的条件），则一个数据竞争被检测到。

DataCollider 需要暂停 某个线程来等待某个断点的触发，因此其等待的时间将直接作用于时间开销。这导致 DataCollider 的时间开销很容易超过 5%。另外，由于 DataCollider 不依赖于 happens-before 等的推理能力，所以它只能检测已经发生的数据竞争，无法预测数据竞争的发生。

我们的方法：
基于对已有工作的分析，我们认为用于部署程序的采样检测技术应该具有如下特性：
1、不完全依赖于 happens-before 关系来检测数据竞争。
2、不直接暂停程序的运行。

基于此，我们首先提出了一种数据竞争的定义：Clock Race，并提出了 CRSampler 来采样 Clock Race。首先，线程的本地时序是一个数字，在该线程进行同步操作的时候增加。给定两个不同线程 t1 和 t2 对同一内存的两个访问事件 e1 和 e2，如果在 e1 和 e2 发生的时间区间内，线程 t1 或者线程 t2 的本地时序没有改变，则 e1 和 e2 构成一个 Clock Race。该定义包含两种情况：线程 t1 的本地时序没有发生改变和 t2 的本地时序没有发生改变，如图 1 所示。但是，从实现的角度来

说，如果 e2 发生在 e1 后面，则在 e2 发生的时候判断 t2 的本地时序是否自 e1 以来发生改变，需要在 e1 处记录全部线程的本地时序，这个操作是 O(n)的，即和线程数量相关。而在 e2 发生时判断 t1 的本地时序则只需在 e1 发生的时候记录本线程的本地时序，该操作是 O(1)的。因此，我们进一步使用后者来定义 Clock Race。

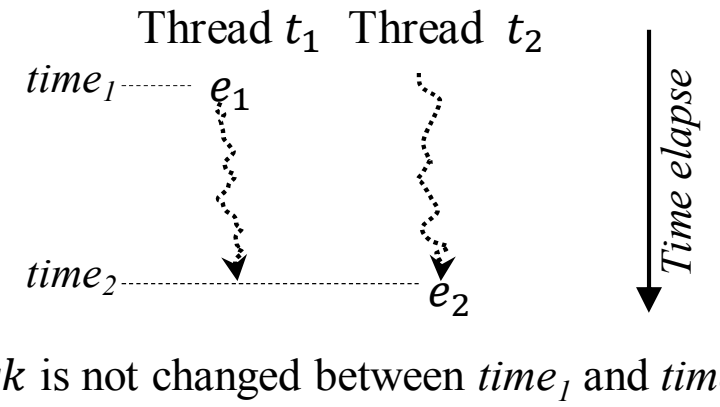

图 1：Clock Race 示意图

从 Clock Race 的定义我们可以看出，（1）其不需要完整的 Happens-before 关系，只需要维护每个线程的本地时序；（2）Clock Race 不需要暂停某个线程。当然，很容易证明，暂停某个线程（例如图 1 中的 t1）则是 Clock Race 的一个特殊情况。

有了 Clock Race 的定义，我们进一步使用硬件数据断点来采样内存访问并检测数据竞争。大致算法如下（如图 2 所示）：首先按照给定的采样率 r 随机选择程序 P 中的内存访问指令（该选择可以通过插桩程序 P 来完成，P'为插桩之后的程序）。之后，在程序的执行过程中，我们维护程序本地的时序。当某个这种内存被执行时，在其要访问的内存处设置内存断点，并记录当前线程的本地时序和内存地址。若某个数据断点触发，则判断之前设置该断点的线程当前的本地时序是否发生改变，若没有，且两次访问中至少有一个是写操作，则检测到一个数据竞争。

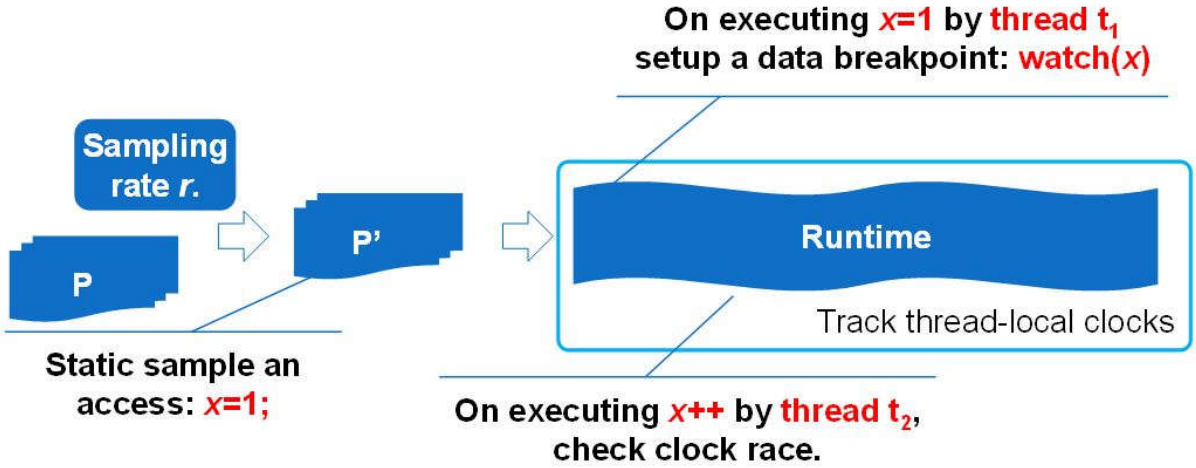

图 2：Clock Race 的检测示意图

当然，和 DataCollider 类似，我们的方案在实践中必须考虑当某个断点设置以后，在多长时间内这个断点有效。因为：如果这个时间设置的太长，即便是有另外一个内存访问触发了断点，也没有 Clock race 被检测到，因为所有线程的本地时序已经被改变了，且目前的硬件中断点都是很少的（一般 PC 机的都只有 4 个）；如果这个时间设置的太短，那些需要较长时间的 Clock race 则无法被检测到。很遗憾的是，这方面很难有一个确定的时间，因为有的 race 只需要很短的时间就会被检测到，但是有的 race 则需要很长的时间。其他人的研究也表明了这一点。

实验：

选取了 Dacapo 的 java 程序测试集，在 JikesRVM 上实现。在实验中，我们对比了 Pacer、DataCollider 和我们的 CRSampler。对 DataCollider 和 CRSampler 各自设置了 15ms 和 30ms 的等待时间，分别简称为 DC_{15}、DC_{30}、CR_{15} 和 CR_{30}。详细实验结果见论文，此处简单比较性能。

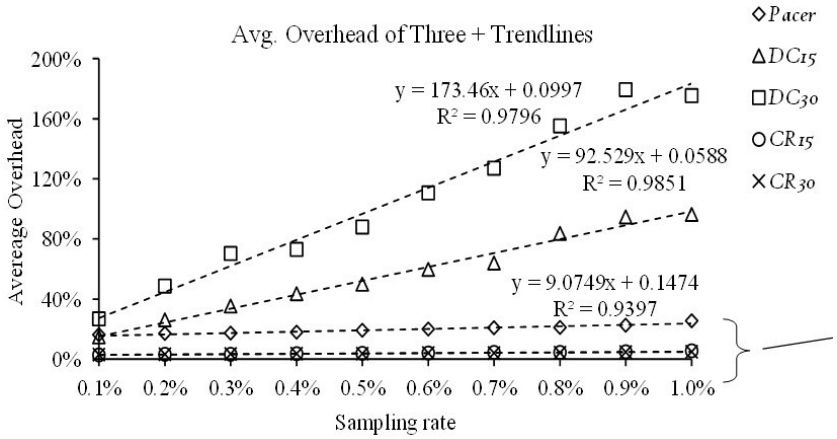

图 3：性能比较

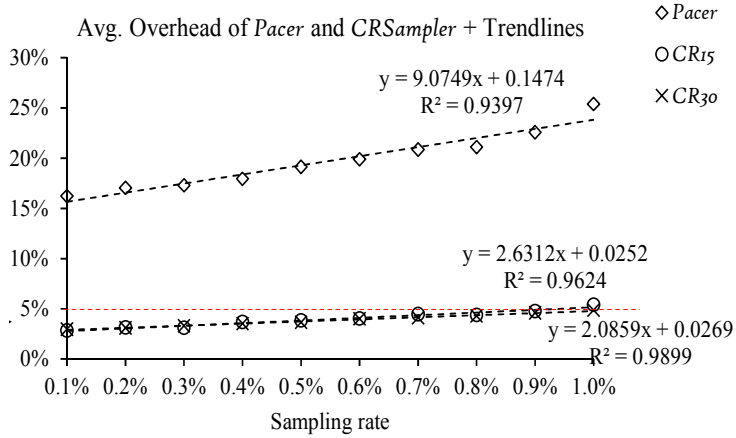

图 4：Pacer 和 CRSampler 的性能比较

图 3 显示了三种采样检测技术的性能对比。从图中我们可以看出，DC 具有最高的 Overhead，Pacer 和 CRSampler 的 Overhead 远低于 DC。但是比较 Pacer 和 CRSampler（如图 4 所示），我们可以发现，Pacer 的 Overhead 增加稍微大于 CRSampler 且其有个基本的 Overhead，约为 15%（这个基本的 Overhead 为 Pacer 作者所测的值的一半）。

另外，比较 DataCollider 和 CRSampler 我们可以发现，当时间间隔从 15ms 增加到 30ms 时，DataCollider 的 Overhead 几乎增加了相应的倍数，但是 CRSampler 的 Overhead 不仅没有增加，反而稍微有点减小（我们在 paper 中详细讨论了这一点）。因此，在实际应用中，CRSampler 可以在任意长的时间间隔下工作而不用担心其性能问题，但是 DataCollider 很难做到这一点。

作者简介：

蔡彦，2014 年于香港城市大学获得博士学位，之后在中国科学院软件研究所（计算机科学国家重点实验室）任副研究员。主要研究并发程序的测试问题，并注重在大规模真实程序中的应用。近五年来发表文章约 20 篇，包括 CCF A 类顶级期刊和会议 IEEE TSE、ICSE、FSE、ASE、IEEE TPDS 上以第一作者身份发表十余篇。其多次受邀为 IEEE TSE、IEEE TC, IEEE TSC、IEEE TR、JSS、The Computer Journal 等著名期刊审稿、担任国际会议 PC 等，任 FCS 青年 AE 并获 FCS 2016 Excellent Young AE Award。更多信息见 http://lcs.ios.ac.cn/~yancai 。

本文作者还包括：中科院软件所 张健研究员、中国科学院大学研二学生 曹玲微、中科院信工所 刘剑副研究员。