doi:10.11959/j.issn.1000-436x.2016173

基于滑动窗口的多核程序数据竞争硬件检测算法

朱素霞[1,2]，陈德运[1,2]，季振洲[3]，孙广路[1]

(1. 哈尔滨理工大学计算机科学与技术学院，黑龙江 哈尔滨 150080；
2. 哈尔滨理工大学计算机科学与技术学院博士后流动站，黑龙江 哈尔滨 150080；
3. 哈尔滨工业大学计算机科学与技术学院，黑龙江 哈尔滨 150001)

摘 要：数据竞争是引起多核程序发生并发错误的主要原因。针对现有基于硬件的 happens-before 数据竞争检测方法硬件开销大的问题，提出了一种轻量级的内存竞争硬件检测算法，该算法利用滑动窗口技术动态检测程序执行过程中发生的距离较近、更易引发并发错误的数据竞争。考虑竞争距离的大小，将并发线程片段细分为加锁并发竞争域和包含线程近期执行序列的未加锁并发竞争域，用一对交替移动的可重写滑动窗口保存未加锁并发竞争域内的内存操作指令，用一个大小可变的可重写滑动窗口保存加锁并发竞争域内的内存操作指令，当来自远程的共享访问与窗口内的内存访问发生冲突时，检测到数据竞争。在硬件实现结构中，仅为每个处理器核添加 3 对较小尺寸的硬件签名寄存器来保存并发竞争域内的数据地址，无需更改原有的 cache 一致性协议，带来的带宽开销低，能够快速地检测多核程序并发执行过程中发生的动态数据竞争，为多核程序开发和生产运行阶段的并发错误诊断提供有效的指导信息。

关键词：数据竞争；滑动窗口；硬件签名；并发错误；多核程序
中图分类号：TP303 **文献标识码**：A

Hardware data race detection algorithm based on sliding windows

ZHU Su-xia[1,2], CHEN De-yun[1,2], JI Zhen-zhou[3], SUN Guang-lu[1]

(1. School of Computer Science and Technology, Harbin University of Technology, Harbin 150080, China;
2. Postdoctoral Research Station, School of Computer Science and Technology, Harbin University of Technology, Harbin 150080, China;
3. School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

Abstract: Data race is a major factor which causes multi-core programs to produce concurrent bugs. To address the high hardware cost in happens-before detection proposals, a light-weight hardware data race detection approach based on sliding window technology was proposed. It used sliding windows to save recent memory instructions in thread execution and dynamically detected data races with small race distance which more easily lead to concurrent bugs. Considering the race distance, parallel thread segments were subdivided into concurrent race regions with lock and concurrent race regions without lock. A pair of alternate rewritable sliding windows was used to store the memory instructions in concurrent race region without lock, and a sliding window with variable size was used to store the memory instructions in concurrent race region with lock. When there was a conflict between a remote sharing access and memory accesses in sliding windows, a data race was detected. In the hardware implementation, the addresses of the data in sliding windows were automatically encoded into three hardware signatures with small size. Data races can be detected quickly without modifying the L1 cache and cache coherence protocol messages. This approach supplies efficient guidance to help users to diagnose concurrency bugs occurred in the development and production run of multi-core programs, achieving smaller hardware and bandwidth overhead.

Key words: data race, sliding window, hardware signature, concurrency bug, multi-core program

收稿日期：2016-04-05；**修回日期**：2016-07-14
基金项目：国家自然科学青年基金资助项目(No.61502123)；黑龙江省青年科学基金资助项目(No.QC2015084)；中国博士后科学基金资助项目(No.2015M571429)；国家自然科学基金资助项目(No.61472100)；国家重点基础研究发展计划（“973”计划）基金资助项目(No.2011CB302501)

Foundation Items: The National Natural Science Foundation of China for Youths(No.61502123), Heilongjiang Province Science Foundation for Youths(No.QC2015084), The China Postdoctoral Science Foundation(No.2015M571429), The National Natural Science Foundation of China(No.61472100), The National Basic Research Program of China(973 Program)(No.2011CB302501)

1　引言

随着多核处理器的广泛应用，多核编程也变得越来越普遍。然而，多核程序执行时因为线程间共享内存访问交互顺序的不确定性，导致并发错误频现，限制了多核程序的应用。多核程序运行时，当 2 个或多个线程并发访问同一个共享变量，没有采取正确的同步措施，并且至少有一个是写操作时，就可能引起数据竞争。数据竞争是一种常见的并发错误，检测数据竞争是多核程序开发、调试和诊断的重要手段，也是多核程序生产运行阶段的重要分析手段。因此，研究者们提出了一系列的数据竞争检测方法，有软件实现的[1~7]，有硬件实现的[8~13]，也有软硬结合的[14~16]，甚至还出现了商用的数据竞争检测工具[17]。本文针对基于硬件的数据竞争动态检测方法展开研究。

通常有 2 大类方法来检测数据竞争，一种是基于锁集合的，如文献[1]；一种是基于 happens-before 关系的，如文献[17]。基于锁集合的方法是依据所有访问同一个共享变量应该使用相同锁的思想，跟踪访问共享变量的锁集合，当 2 个访问同一个共享变量使用的锁集合的交集为空时，则认为存在数据竞争。Happens-before 方法基于线程片段，每个处理器核使用一个逻辑时钟来标记当前正在执行的线程片段，此外每个变量都有一个时戳记录它在处理器访问的哪个片段中，当另一个处理器访问这个变量时，将变量的时戳同自身的时钟进行比较，来决定这 2 个相应的片段是否存在逻辑上的 happens-before 关系，还是存在逻辑上的重叠，如果存在逻辑上的重叠，则认为存在竞争。

软件实现的数据竞争检测算法通常会以 10~200 倍降低程序运行的速度[10]，如此降速会影响程序运行的顺序或竞争发生的时间，使生产运行时出现的数据竞争更是难以发现。因此，有研究者提出了基于硬件的数据竞争检测方法。基于硬件的检测方法对程序的性能影响较小，对发现程序生产运行时的数据竞争比较有效，然而它们往往添加较多的硬件资源。比如文献[8]需要为 cache 块添加额外的时戳或锁信息，文献[9]改变 cache 一致性协议状态机，文献[10]采用了基于硬件签名的方式实现数据竞争的检测，但需要添加签名队列，硬件开销仍然过大，并且采用代价较高的回滚机制来定位竞争的位置。而且，大多数基于锁集合和 happens-before 的硬件检测方法需要在现有的 cache 一致性协议基础之上添加新的消息。然而，cache 和一致性协议部件都是处理器的关键部件，如果需要增加过多硬件资源或更改 cache，需要重新评估其对处理器性能的影响，不利于应用到实际中。虽然近期也有研究者提出的其他类型的数据竞争硬件检测方法硬件的开销较小[11~13]，但均只能检测某特定类型的数据竞争。

本文针对现有基于 happens-before 数据竞争检测方法硬件开销大的问题，鉴于线程的并发执行是导致竞争发生的主要原因，结合竞争距离大小，将并发的线程片段细分为加锁并发竞争域和未加锁并发竞争域，提出了一种轻量级的动态数据竞争检测方法。该方法基于在线数据流处理中常用的滑动窗口技术，保存线程近期执行的内存操作指令序列，动态地检测竞争距离较近的、更易引发并发错误的数据竞争。该方法无需更改 cache 和一致性协议机构，仅添加少量的硬件签名寄存器，带来的带宽开销小。

本文的研究针对采用锁同步方式的多核程序展开。

2　研究动机

数据竞争的检测是 NP 困难问题，已往的数据竞争检测方法大多旨在检测尽可能多的数据竞争。然而，现实情况存在以下问题。

1) happens-before 算法代价昂贵

基于 happens-before 的内存竞争检测方法需要考虑所有的同步操作，还需要使用向量时钟对不同线程中的内存访问进行标记和排序，无论是已有的软件实现方法还是硬件实现方法，都在内存或硬件开销方面付出了较大代价。

2) 数据竞争是否会引起并发错误受距离影响

数据竞争是引发并发错误的主要原因，但并不是所有的数据竞争都会引发并发错误，尤其是那些竞争双方距离较远的数据竞争。因为距离较远的数据竞争执行顺序发生反转的概率小，从而引发错误的可能性就小[10]。如图 1(a)所示，线程 j 访问共享变量 x 后，线程 i 过了很久才访问 x，这 2 个访问在时间上相隔很远，虽然线程 j 未添加同步操作，但该数据竞争执行顺序发生反转是一个小概率事件，引起错误的概率小，在一定条件下，可以不予以检测。

3) 纠正错误不一定要检测出所有的数据竞争

检测数据竞争可以有效地帮助多核程序的诊断和调试，然而，有时一个同步操作的错误使用或漏掉，可能会引发多个数据竞争，但只要检测出其中的部分竞争，就可以帮助用户找出同步错误的所在，从而修正程序。如图 1(b)所示，因线程 *j* 漏掉了一个同步操作，会引发①和②共 2 个数据竞争，而只要检测到①这一个数据竞争就可以修复程序。

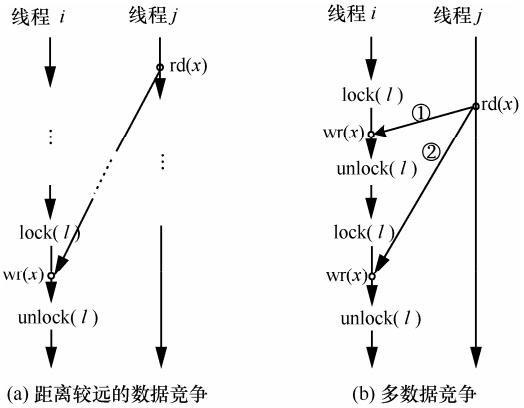

图 1 数据竞争示意

鉴于以上 3 点的分析，为了减小基于 happens-before 的硬件数据竞争检测方法带来的硬件开销，进一步降低检测算法的复杂度，并且能给用户或程序员提供诊断信息，尤其是提供生产运行阶段的诊断信息，本文提出了一种轻量级的数据竞争检测算法。该方法引入并发竞争域，用滑动窗口保存线程近期执行的内存操作，能够检测打断临界区操作的数据竞争和其他距离较近的数据竞争，对于距离较远、不易引起并发错误的数据竞争不予检测。

3 竞争距离

距离较远的数据竞争因其执行顺序发生反转的概率小，引起错误的可能性小，因此，在进行数据竞争检测时，可以更多地关注距离较近的数据竞争，从而为检测并发错误提供更加有效的诊断信息。为了描述数据竞争双方间的距离大小，本文提出竞争距离（race distance），并约定竞争距离表示：数据竞争的后发生方执行时，数据竞争的先发生方所在线程在执行完先发生方后又执行的内存操作指令数。如图 2 所示，圆圈表示内存操作，线程 *i*、*j* 间存在数据竞争 j:rd(x) → i:wr(x)，在线程 *j* 执行先发生方 rd(x)后，直到线程 *i* 执行后发生方 wr(x)时，线程 *j* 又执行了 3 条内存操作指令，称该数据竞争的竞争距离(*rl*)为 3。

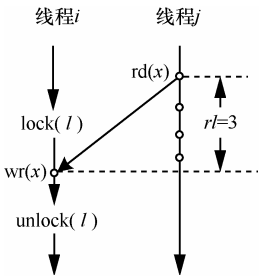

图 2 竞争距离

针对竞争距离在多大的情况下，数据竞争执行顺序发生反转的概率小，可以不需要检测的问题，本文对竞争距离和临界区的关系及其大小进行了分析和测试。通常情况下，若有共享变量访问，为避免发生竞争需要为其添加加锁和解锁操作。而且，临界区不应太大，因为临界区太大会降低程序的性能，这不是一种良好的编程习惯。如果某线程执行完加解锁操作合围的临界区后，其他线程再来访问由该临界区保护的共享变量就不会引起竞争，否则很可能会引起竞争。同理，如果漏掉加解锁操作，则在其原本应该有的临界区范围内有远程访问就可能会引发数据竞争，超出临界区范围则不会引起竞争。鉴于以上分析，可以发现数据竞争与临界区的大小有一定关系：如果竞争距离大于临界区，则发生数据竞争的可能性就变小。因此，竞争距离可以依据临界区的大小为依据来设定。本文对测试负载进行了临界区大小统计，详见 7.1 节，并给出了本方案中合理的竞争距离范围。

4 并发竞争域

基于 happens-before 的数据竞争检测方法通常将线程的执行序列依据同步操作划分为一个个的线程片段，通过比较向量时戳，可以找到不存在 happens-before 关系、可能并发执行的线程片段，如图 3（中括号内给出了线程片段的向量时戳）所示的 S_{i1} 和 S_{j1}、S_{i2} 和 S_{j1}、S_{i3} 和 S_{j1}、S_{i3} 和 S_{j2}、S_{i3} 和 S_{j3} 均不存在 happens-before 关系，在程序执行过程中可能会发生数据竞争。因此，可以通过监测程序执行过程中并发执行的线程片段来监测动态的数据竞争。如图 3 所示的执行顺序中，并发的线程片段 S_{i2} 和 S_{j1} 之间存在数据竞争 j:rd(x) → i:wr(x)；并发的线程片段 S_{i3} 和 S_{j1} 之间存在数据竞争 j:wr(x) → i:wr(x)和 j:rd(x) → i:wr(x)，而且数据竞争 j:rd(x) → i:wr(x)的竞争距离较近，更易引发并发错误；并发的线程片段 S_{i3} 和 S_{j2} 之间存在数据竞争 i:wr(x) → j:rd(x)。

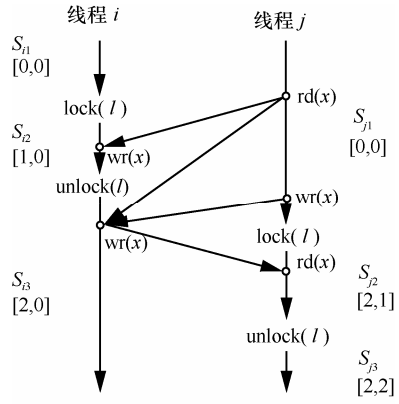

图 3　并发线程片段与数据竞争

为了降低 happens-before 算法的复杂度，本文结合上述分析仅检测程序执行过程中并发线程片段间距离较近、更易引发并发错误的数据竞争，并把可能引起数据竞争的、近期访问的一段线程片段称为并发竞争域(CRR, concurrent race region)。根据线程片段是否由加解锁操作合围，将并发竞争域又细分为 2 大类：一类是加锁并发竞争域，该域被加解锁操作合围起来，对应加锁线程片段；另一类是未加锁并发竞争域，该区域没有被加解锁操作包围，是未加锁线程片段的子集，仅包含未加锁线程片段中近期访问的指令执行序列。如图 4 所示，线程 i 中存在一个加锁并发竞争域 CRR_i，线程 j 中存在一个未加锁并发竞争域 CRR_j，2 个属于并发执行的程序片段，因为都访问了共享变量 x，因此存在数据竞争。为了能够定位检测到的数据竞争对应的内存地址，本文将来自远程的共享访问与并发竞争域中的访问有冲突时，认定存在数据竞争，如图 4 中存在数据竞争 i:wr(x)→j:rd(x)。

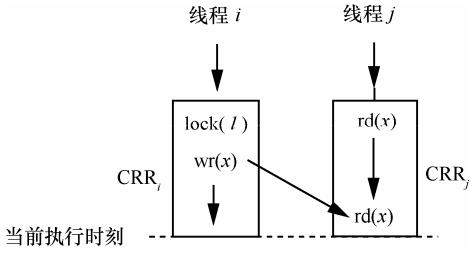

图 4　并发竞争域

引入并发竞争域，可以有效地检测引发并发错误的 2 类主要竞争类型。一类是来自远程并发竞争域的共享访问与加锁的并发竞争域内的访问存在冲突，则对该地址的访问存在竞争，这类竞争至少有一方进行了加解锁保护，通常被称为非对称竞争[11]，如图 5(a)和图 5(b)所示。此类竞争打断了临界区操作，是程序执行中坚决不允许出现的，本文记该类竞争为 LRace。另一类是来自远程并发竞争域的共享访问与未加锁并发竞争域内的访问发生冲突，则对该地址的访问存在竞争，而且竞争距离越小，越容易引发并发错误。如图 5(c)和图 5(d)所示，本文记为 ULRace，该类中也存在打断临界区的情况，如图 5(d)所示。

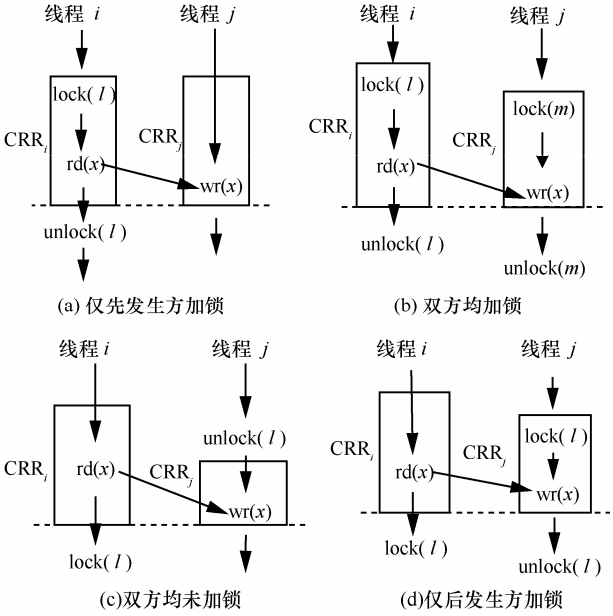

图 5　检测到的数据竞争类型

5　基于滑动窗口的动态数据竞争检测

对于第 1 类数据竞争，因其先发生方位于临界区内，而临界区的执行是不能被打断的，因此不管竞争距离远近都要检测。对于第 2 类数据竞争，因为其先发生方位于未加锁并发竞争域内，远距离的数据竞争可以不予考虑。因此对于这 2 类竞争的检测方法要区别对待。下面分别给出 2 类竞争的检测方法的具体描述。

5.1　未加锁并发竞争域

针对未加锁并发竞争域，为确保能够保存近期访问的执行序列，以便检测到竞争距离较近的数据竞争，本文借鉴在线数据流处理中常用的滑动窗口技术，引入一对交替移动的可重写滑动窗口：窗口 1 和窗口 2，用来存放未加锁并发竞争域内的内存操作指令，每个窗口最多能够容纳有限数量个内存操作。随着程序的执行，窗口可以不断交替下移，线程内的执行序列便不断加入到了滑动窗口中；当窗口 1、窗口 2 都满时，则清空并下移窗口 1 用来

存放新的内存操作；再次全满后，则清空并下移窗口 2 用来存放新的内存操作。

指令在滑动窗口中流动的过程如图 6 所示，箭头表示程序执行的顺序，矩形框分别表示窗口 1 和窗口 2，2 个窗口均只能存放有限数量的内存操作指令，未加粗实线矩形框表示工作窗口，加粗实线矩形表示已满工作窗口，虚线矩形框表示已清空并下移的窗口，加粗虚线矩形框表示待工作窗口。具体流动过程描述如下。

初始情况下，窗口 1 在前，窗口 2 在后，内存操作指令依次加入到窗口 1 中，如图 6(a)所示。

当窗口 1 满，则将后续内存操作指令依次加入到至窗口 2 中，如图 6(b)所示。

如果窗口 1 和窗口 2 全满，则窗口 1 清空并下移，用来存放后续内存操作，此时窗口 2 在前，窗口 1 在后，如图 6(c)所示。

当窗口 2、窗口 1 全满，则窗口 2 清空并下移，用来存放后续内存操作指令，此时窗口 1 在前，窗口 2 在后，如图 6(d)所示。

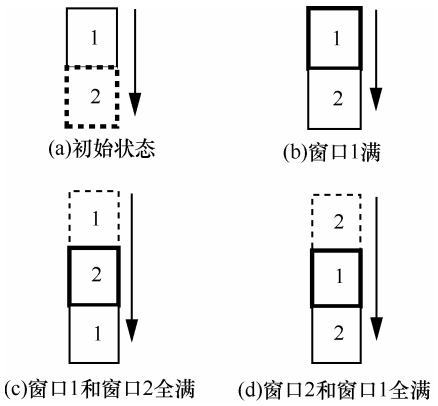

图 6 指令在滑动窗口的流动示意

设每个窗口最多可容纳 m 个内存操作，如此交替移动，便可存放线程最近执行的至少 m 个内存操作（初始情况除外）。这样，如果来自其他线程的远程访问同滑动窗口内的内存操作发生冲突，则认为存在竞争。从而能够检测到所有竞争距离在 0~m 的数据竞争，还可以检测部分 m~$2m$ 的内存竞争，距离大于 $2m$ 的不予以检测。如此，通过一对滑动窗口的交替移动和重写，有效地检测到先发生方位于未加锁并发竞争域、竞争距离较近的数据竞争。

该检测方法中距离较远的数据竞争不会被检测到，如图 7 中虚线指出的竞争。当该数据竞争后发生方执行时，线程 i 已经在执行完 wr(x)后至少又执行了 m 个内存操作，因为此时窗口 1、窗口 2 的前后顺序已经交替移动过，位于前面的窗口是满的。线程 j 的 wr(x)操作距离线程 i 的 wr(x)操作的距离大于 m，相对较远。假设存在临界区的话，wr(x)执行时，线程 i 的关于 wr(x)的临界区已经执行完毕，不会破坏线程 i 中 wr(x)操作相关的临界区。

图 7 中可以检测到线程 i 窗口 1 中的 rd(x)与线程 j 的 wr(x)之间的竞争。虽然该竞争距离大于 m 且接近 $2m$，但其仍在滑动窗口内，竞争的距离未超过 $2m$，仍然能够检测到。

滑动窗口的大小决定了所能检测到的数据竞争的距离，在后面的仿真测试中，本文给出了滑动窗口的合理尺寸。

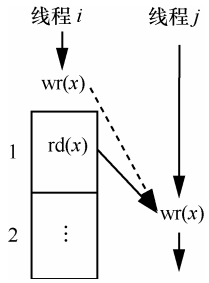

图 7 未加锁并发竞争域内的竞争示例

5.2 加锁并发竞争域

临界区的执行是不允许被打断的，因此，如果临界区内有来自其他线程的访问冲突，则必引发竞争，此竞争必须要检测到。因此，本文将加锁竞争域内的内存操作指令用一个大小可扩展的滑动窗口来保存，该滑动窗口可以容纳不同大小临界区内的所有内存操作，窗口随着临界区内内存操作数量的增加而增大。一旦来自远程线程的共享访问与窗口内的访问发生冲突，则检测到了数据竞争。如图 8 所示，线程 j 访问执行 wr(x)时，线程 i 还未执行完保护共享变量操作 wr(x)的临界区，则会检测到数据竞争 i:rd(x)→j:wr(x)。

6 硬件实现

基于上述滑动窗口的数据竞争检测方法，实现了基于 CMP（chip multiprocessor）系统的数据竞争硬件检测算法。该检测算法对应的硬件结构中需要为每个处理器核添加一个内存竞争检测模块 RaceSW，如图 9 所示。其中包括 3 对读写签名寄

存器：RF0/WF0、RF1/WF1、RF2/WF2。RF0/WF0 和 RF1/WF1 这 2 对签名分别用于存放未加锁并发竞争域中滑动窗口 1 和窗口 2 存放的读写操作的数据地址，且每对读写签名最多能存放 m 个内存操作的数据地址。RF2/WF2 用来存放未加锁并发竞争域中滑动窗口 3 存放的内存操作的数据地址，存放数量不限。当窗口 1 下移时，签名对 RF0/WF0 清空，用来存放窗口 1 后续存放的内存操作的数据地址，当窗口 2 下移时，签名对 RF1/WF1 清空，用来存放窗口 2 后续存放的内存操作指令的数据地址。RF2/WF2 用来存放加锁并发竞争域中窗口 3 存放的内存操作的数据地址，并且存放的地址数量不受限制。在签名寄存器大小固定的情况下，滑动窗口设置越大，能够检测到具有更大竞争距离的数据竞争，但因更多的地址加入到签名寄存器中，带来误报也会增加。因此在第 7 节中对滑动窗口和签名寄存器的大小进行了仿真测试，选取了合适的参数。

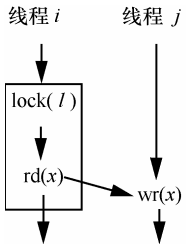

图 8　加锁并发竞争域竞争示例

除了 3 对读写签名寄存器外，RaceSW 还包括指令计数器 IC，用来记录窗口 1 和窗口 2 中的内存操作数量，以及一系列的标识触发器：Order(窗口顺序标识)、Full0（窗口 1 满标识）、Full1（窗口 2 满标识）、Lock（加锁标识）、Filter（过滤标识）。

同时，为了识别程序中的加锁、解锁操作，以及在检测竞争时过滤掉锁操作本身带来的竞争，还需要为处理器增加新的机器指令。机器指令的实现形式多样，可以为每个同步操作分别引入 2 条指令，一个是打开地址过滤功能，一个是关闭地址过滤功能；还可以综合应用更少数量的机器指令来识别不同操作。鉴于尽可能引入较少的机器指令，降低硬件复杂度，该硬件实现中仅增加了 Lock_on、Lock_off、Filter_off 3 个新的机器指令，如表 1 所示。这 3 条指令相当于硬件开关，Lock_on 指令既能结束一个未加锁并发竞争域又可以开启一个新的加锁并发竞争域，同时还开启了锁竞争过滤功能，即将锁操作自身带来的内存地址不添加到签名中。Lock_off 指令既能结束一个加锁并发竞争域又可以开启一个新的未加锁并发竞争域，同时开启锁竞争过滤功能。Lock_on、Lock_off 分别和 Filter_off 配合，可以对锁操作实施过滤功能，不将它们加入到滑动窗口中，从而过滤掉锁操作本身带来的数据竞争，使数据竞争检测的结果更加有意义。

表 1　　新增机器指令

指令	描述
Lock_on	开启加锁并发竞争域和地址过滤功能
Lock_off	开启未加锁并发竞争域和地址过滤功能
Filter_off	结束地址过滤功能

虽然增加了 3 条新的机器指令，但并不需修改用户程序，只要修改库函数即可。如表 2 所示，给出了针对 M4 macros[18]库的修改。其中对于 barrier 操作，成对使用 Lock_off 和 Filter_off，将其带来的内存地址给过滤掉。而且，还可以灵活应用这几条指令，将不想进行数据竞争检测的区域过滤掉。

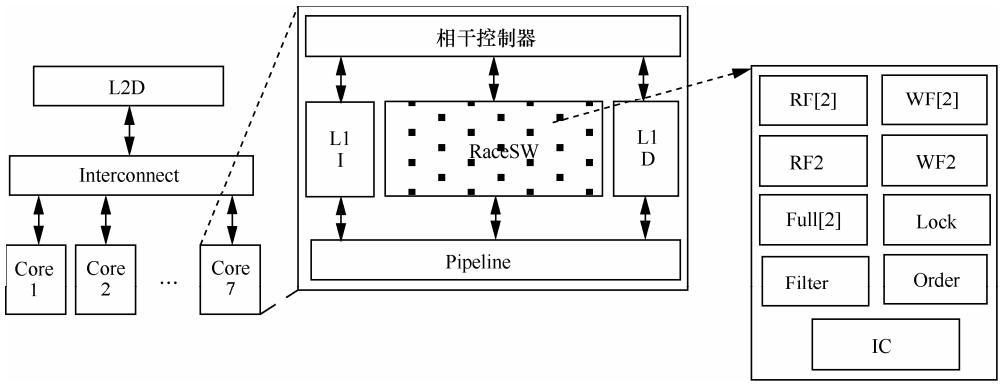

图 9　硬件实现结构

表 2 对库函数的修改

操作	方法	代码

lock	原函数	define(LOCK, `{
		lock($1);
		}')

	RaceSW	define(LOCK, `{
		Lock_on;
		lock($1);
		Filter_off;
		}')

unlock	原函数	define(UNLOCK, `{
		lock($1);
		}')

	RaceSW	define(UNLOCK, `{
		Lock_off;
		lock($1);
		Filter_off;
		}')

barrier	原函数	define(BARRIER, `{
		barrier(($1),($2));
		}')

	RaceSW	define(BARRIER, `{
		Lock_off;
		barrier(($1),($2));
		Filter_off;
		}')

本文提出的基于滑动窗口的数据竞争检测算法基于硬件的描述如下。它详细描述了每个处理器核的动作。

```
On commit of an instruction{
if (instr is Filter_on)
    Filter=true;
else if (instr is Filter_off)
    Filter=false;
}
if(instr is a memory instruction){
    IC++;
    If((Lock==false)&&(Filter==false))
{//concurrent race section without lock
        if(Order==false){//window 0 is before
window 1
            if(Full0==false){
                if(instr is a store)
                WF0.insert (instr.mem.address);
            else
                RF0.insert(instr.mem.address);
        }else if((Full0==true)&&(Full1== false)) {
            if(instr is a store)
                WF1.insert(instr.mem.address);
            else
                RF1.insert(instr.mem.address);
        }else if((Full0==true)&&(Full1== true)){
            RF0.clear();
            WF0.clear();
            Order=true;
            if(instr is a store)
                WF0.insert(instr.mem.address);
            else
                RF0.insert(instr.mem.address);
        }
    }else{//window 1 is before window 0
        if(Full0==false){
            if(instr is a store)
                WF0.insert(instr.mem.address);
            else
                RF0.insert(instr.mem.address);
        }else if(Full0==true) {
            RF1.clear();
            WF1.clear();
            Order=false;
            if(instr is a store)
                WF1.insert(instr.mem.address);
            else
                RF1.insert(instr.mem.address);
        }
    }
}elseif((Lock==true)&&(Filter==false))
{//Concurrent race section with lock
        if(instr is a store)
            WF2.insert(instr.mem.address);
        else
            RF2.insert(instr.mem.address);
        }else{
            RF0.clear();
            WF0.clear();
            RF1.clear();
            WF1.clear();
            RF2.clear();
            WF2.clear():
        }
    }
}
On sending an ack for block b from proc j {
```

```
if(WF0.find(b.address)||WF1.find(b.address)||
(RF0.find(b.address)
&&(request==GETX))||(RF1.find(b.address) &&
(request==GETX))
Racelog.append(b.address);
}
```

该算法中每个处理器核做如下动作。

1) 每当处理器核执行内存操作指令时，首先判断当前是处于未加锁并发竞争域还是加锁并发竞争域，如果是未加锁并发竞争域，则将内存操作指令的数据地址按照滑动窗口 1 和窗口 2 的前后顺序分别加入到窗口对应的签名对中，如果是加锁并发竞争域，则将内存操作指令的数据地址加入到滑动窗口 3 对应的签名对中。

2) 根据滑动窗口 1 和窗口 2 的空满状况交替清空并移动对应的签名对。

3) 如果遇到加解锁操作，则清空窗口 1 和窗口 2 对应的签名对。

4) 当收到来自其他处理器核的共享内存访问请求时，处理器核查找签名来检测是否有数据竞争发生，若检测到，则记录竞争地址到竞争日志。

如果要区分检测到的数据竞争属于双方均未加锁、仅发生方加锁、仅后发生方加锁、双方均未加锁这 4 类中的哪一类，还需要在 cache 一致性协议发送 gets 或 getx 请求消息时添加该地址是否在加解锁范围内的标识信息。如此，程序员可以更加方便地查找错误和修改程序。

7　性能评价

本文采用 GEMS[19]对基于滑动窗口的数据竞争检测算法进行了仿真，仿真配置如表 3 所示。测试负载选取典型的应用于多线程科学计算的 SPLASH2[20]。

表 3　　仿真配置

参数	说明
Cores	8 核，in-order，3 GHz
Private L1 Cache	分离的指令 cache 和数据 cache，大小为 64 KB、4 路组相关、采用写回策略、每个 cache 块为 64 byte，LRU 替换策略
Shared L2 Cache	2 MB，4 路组相关，LRU 替换策略
Memory	4 GB DRAM
Coherence	MESI
Consistency Model	sequential consistency（SC）

下面给出该数据竞争检测算法（RaceSW）在参数选取、硬件开销、检测性能和带宽开销方面的仿真结果。

7.1　参数选取

1) 滑动窗口

选取滑动窗口的大小决定了所能检测到的内存竞争距离的大小。为了选取合适窗口，对临界区内的内存操作数量进行了统计，结果如图 10 所示。所有测试负载中，内存操作个数小于 8 的临界区占的比例最大，比如 ocean、fft 的临界区均小于 8；小于 256 的临界区约占为 95%；大于 512 的临界区占的比例非常少。虽然，cholesky、fmm 中大于 1 024 的临界区所占比例相对较多，但实际数量均不超过 10 个。

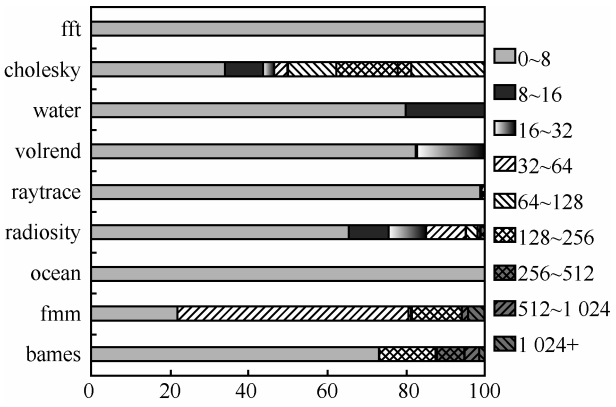

图 10　不同大小临界区所占比例统计

滑动窗口如果设置过大，不易于去除距离较远的数据竞争，而且会浪费硬件资源，如果过小则又会漏掉数据竞争。因此，本方案中，根据临界区大小的结果统计，将未加锁并发竞争域内引入的滑动窗口大小设置为 256，能够包含占绝大多数的小于 256 的临界区。如此，对于未加锁并发竞争域，采用滑动窗口技术，除初始情况外，均至少能保存每个线程内近期执行的 256 个内存操作，完全可以检测所有竞争距离在 0~256 范围内的数据竞争，可以检测部分竞争距离在 256~512 范围内的竞争，距离超过 512 不予检测。相应地，WF0/RF0、WF1/RF1 这 2 对签名均最多只能容纳 256 个内存操作的数据地址。对于加锁并发竞争域，滑动窗口大小不设限制，对应签名寄存器存放的地址数目也不做限制，从而可以检测所有打断临界区的操作。

2) 签名寄存器

选取的签名寄存器如果太小，则误报（false

positive）会增多，如果过大，则浪费硬件资源。在本算法中分别用 WF0/RF0、WF1/RF1 来存储最多 256 个内存操作的数据地址；用 WF2/RF2 存放临界区中所有的内存操作对应地址，存放数据无上限要求，但大于 1 024 的临界区所占比例不到 2%。考虑较好的资源利用率，本文针对常用的 H3 散列签名寄存器的多个尺寸进行了测试，测试结果如图 11 所示，发现签名寄存器大于 128 bit 后，对检测到的数据竞争数量影响不太大，因此本文选用 128 bit 的读写签名寄存器。

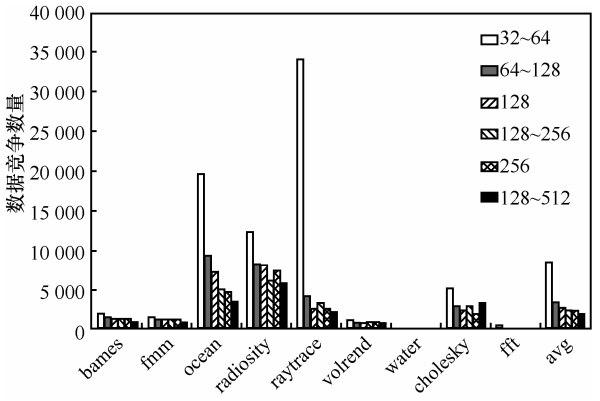

图 11 签名寄存器大小测试结果

7.2 硬件开销

该算法需要为每个处理器核添加一个 RaceSW 模块，对于 8 核的 CMP 系统，配置参数如表 3 所示，若不考虑运算器部分，该模块共添加 3 对 128 bit 的硬件签名寄存器，共 768 bit，外加 1 个 16 bit 的指令计数器（IC）和 5 个触发器，共添加 789 bit 的硬件资源，而文献[10]中为每个处理器核添加 4 kbit，RaceSW 硬件开销减小了约 80%，相比其他不使用签名的硬件检测算法[8,9]，RaceSW 在更大程度上降低了硬件开销。

7.3 检测到的数据竞争数量

图 12 分别给出了 RaceSW 在 4 核、8 核和 16 核 CMP 系统上检测到的数据竞争数量及其检测到的两大类型数据竞争所占的比例情况。可以看出，先发生方在未加锁并发域的数据竞争占绝大多数；不存在超长临界区的测试负载（如 water、volrend 、ocean、fft），没有检测到打断临界区的数据竞争，而存在超长临界区的测试负载都检测到了打断临界区的数据竞争，如 barnes、fmm 等。这同时也为编程人员针对临界区大小设置不合理提出了提示信息。因为 RaceSW 采用滑动窗口策略，仅检测竞争距离较近、更易引发并发错误的数据竞争，在降低硬件复杂度和硬件开销的前提下，相比文献[10]等，数据竞争检测效果有所下降。本文在相同 8 核环境下采用大尺寸签名寄存器代替签名队列针对文献[10]提出的 SigRace 检测方法进行了测试，并考虑到库文件带来的竞争，检测到的竞争数量比较结果如图 13 所示，RaceSW 可以检测到近期发生的约 21%的数据竞争。

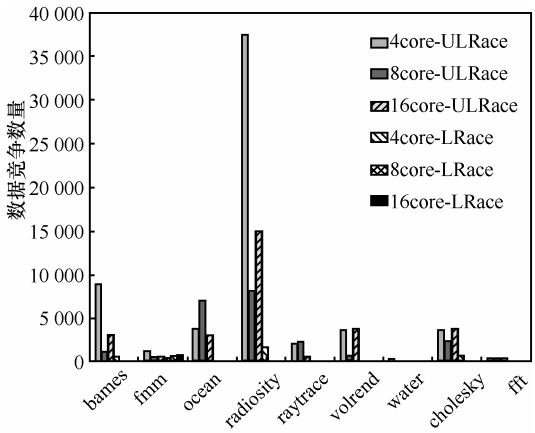

图 12 数据竞争数量统计

7.4 带宽开销

如果不提示竞争类型，则本方法不需要为 cache 一致性协议添加新的消息字段，带来的带宽开销为 0。如果要指出竞争的类型，则只需要在一致性消息 getx 和 gets 中添加 1 bit 的附加信息，用来指出后发生方来自未加锁并发竞争域还是加锁并发竞争域，此时，带宽开销不到 1%。

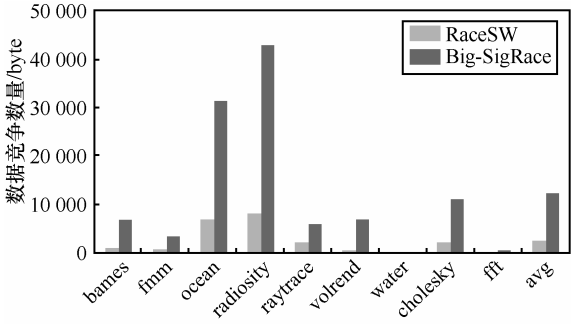

图 13 数据竞争数量比较

8 结束语

本文针对基于硬件的 happens-before 内存竞争检测方法硬件开销大的问题，提出了基于滑动窗口的动态数据竞争检测算法，该算法从远距离的内存

竞争引起并发错误的概率较小这一特点出发，将并发的线程片段细分为由线程近期执行的内存操作构成的未加锁并发竞争域和位于临界区内的加锁并发竞争域，并采用滑动窗口技术，用一对交替移动的可重写滑动窗口保存未加锁并发竞争域内的内存操作，用可变大小的滑动窗口保存加锁并发竞争域内的内存操作。硬件实现结构中，滑动窗口内的数据地址自动添加到小尺寸的签名寄存器中，当有来自其他处理器核的一致性共享请求消息时，通过查找签名检测到距离较近的、更易引发并发错误的内存竞争。基于 8 核的 CMP 系统下的仿真结果指出，该算法硬件开销小、带宽开销低。

参考文献：

[1] SAVAGE S, BURROWS M, NELSON G, et al. Eraser: a dynamic data race detector for multithreaded programs[J]. ACM Transactions on Computer Systems (TOCS), 1997, 15(4): 391-411.

[2] DI P, SUI Y. Accelerating dynamic data race detection using static thread interference analysis[C]//The 7th International Workshop on Programming Models and Applications for Multicores and Manycores. ACM, 2016: 30-39.

[3] WU Z, LU K, WANG X, et al. Collaborative technique for concurrency bug detection[J]. International Journal of Parallel Programming, 2015, 43(2): 260-285.

[4] 陈睿, 杨孟飞, 郭向英. 基于变量访问序模式的中断数据竞争检测方法[J]. 软件学报, 2016, 27(3): 547-561.
CHEN R, YANG M F, GUO X Y. Interrupt data race detection based on shared variable access order pattern[J]. Journal of Software, 2016, 27(3): 547-561.

[5] 王文文, 武成岗. 动态容忍和检测非对称数据竞争[J]. 计算机研究与发展, 2014, 51(8): 1748-1763.
WANG W W, WU C G. Dynamically tolerating and detecting asymmetric race[J]. Journal of Computer Research and Development, 2014, 51(8): 1748-1763.

[6] LU K, WU Z, WANG X, et al. RaceChecker: efficient identification of harmful data races[C]//2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2015: 78-85.

[7] WESTER B, DEVECSERY D, CHEN P M, et al. Parallelizing data race detection[J]. ACM Sigplan Notices, 2013, 48(4): 27-38.

[8] PRVULOVIC M. CORD: cost-effective (and nearly overhead-free) order-recording and data race detection[C]//The Twelfth International Symposium on High-Performance Computer Architecture. IEEE, 2006: 232-243.

[9] ZHOU P, TEODORESCU R, ZHOU Y. HARD: hardware-assisted lockset-based race detection[C]//2007 IEEE 13th International Symposium on High Performance Computer Architecture. IEEE, 2007: 121-132.

[10] MUZAHID A, SUÁREZ D, QI S, et al. SigRace: signature-based data race detection[J]. ACM Sigarch Computer Architecture News, 2009, 37(3):337-348.

[11] QI S, OTSUKI N, NOGUEIRA L O, et al. Pacman: tolerating asymmetric data races with unintrusive hardware[C]//High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium. IEEE, 2012: 1-12.

[12] QI S, MUZAHID A A, AHN W, et al. Dynamically detecting and tolerating if-condition data races[C]//The 20th IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE Computer Society, 2014: 120-131.

[13] OROSA L. A hardware approach to detect, expose and tolerate high level data races[C]//The 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). IEEE, 2016: 159-167.

[14] DEVIETTI J, WOOD B P, STRAUSS K, et al. RADISH: always-on sound and complete race detection in software and hardware[C]//IEEE Computer Architecture (ISCA), 2012 39th Annual International Symposium. IEEE, 2012: 201-212.

[15] ARULRAJ J, CHANG P C, JIN G, et al. Production-run software failure diagnosis via hardware performance counters[J]. ACM Sigarch Computer Architecture News, 2013, 41(1): 101-112.

[16] SHENG T, VACHHARAJANI N, ERANIAN S, et al. RACEZ: a lightweight and non-invasive race detection tool for production applications[C]//The International Conference on Software Engineering. 2011: 401-410.

[17] Intel Corporation. Intel thread checker[EB/OL]. http://www.intel.com, 2008.

[18] LUSK E, BOYLE J, BUTLER R, et al. Portable programs for parallel processors[M]. Holt, Rinehart & Winston, 1988.

[19] MARTIN M M K, SORIN D J, BECKMANN B M, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset[J]. ACM SIGARCH Computer Architecture News, 2005, 33(4): 92-99.

[20] WOO S C, OHARA M, TORRIE E, et al. The SPLASH-2 programs: characterization and methodological considerations[J]. ACM SIGARCH Computer Architecture News, ACM, 1995, 23(2): 24-36.

作者简介：

朱素霞（1978-），女，山东寿光人，博士，哈尔滨理工大学副教授，主要研究方向为高性能体系结构、并行计算。

陈德运（1962-），男，黑龙江哈尔滨人，哈尔滨理工大学教授、博士生导师，主要研究方向为图像处理、探测和成像技术。

季振洲（1965-），男，黑龙江哈尔滨人，哈尔滨工业大学教授、博士生导师，主要研究方向为并行体系结构、并行计算和网络安全。

孙广路（1979-），男，黑龙江哈尔滨人，哈尔滨理工大学教授，主要研究方向为网络安全、模式识别和机器学习。