并行程序错误调试技术研究综述①

戴卓方, 张为华

(复旦大学 软件学院, 上海 201203)
(上海市数据科学重点实验室, 上海 201203)
(复旦大学并行处理研究所, 上海 201203)

摘 要: 随着多核设备的普及, 主流软件系统的构建模式已经由单线程串行转为多线程的并行方式. 然而, 由于并行程序的不确定性, 使得调试并行程序错误要比串行程序的错误困难很多. 因此, 如何高效地调试并行程序错误成为了亟待解决的问题. 对并行错误调试技术做了全面的研究与分析. 在此基础上, 进一步讨论了不同调试技术的优劣, 也对并行调试技术可能的发展方向进行展望.

关键词: 多核设备; 并行程序; 调试技术; 错误检测; 测试调度

Survey and Analysis of Debugging Concurrecy Bug

DAI Zhuo-Fang, ZHANG Wei-Hua

(Software School, Fudan University, Shanghai 201203, China)
(Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 201203, China)
(Parallel Processing Institute, Fudan University, Shanghai 201203, China)

Abstract: The popularity of multiple-core devices has made parallel programming a necessity to harness the abundant hardware resources. However, due to the non-determinism of parallel software, writing robust parallel software is notoriously hard. Therefore, how to debug concurrency bugs efficiently has become an issue that needs to be urgently deal with. In this paper, we have surveyed the parallel debugging technologies systematically. Further, based on the survey, we have made classifications and comparisons. At last, we have presented the prospects of the possible development direction of concurrency bug debugging approaches.

Key words: multicore; parallel software; concurrency bug debugging; bug detection; test schedule

随着计算机技术的发展, 我们已经进入多核时代. 大至几百乃至几千核的超级计算机或服务器, 小至个人计算机、平板电脑以及移动电话等, 都拥有了多核处理器. 因此, 为了利用多核设备带来的潜在计算能力, 并行程序得到了普遍的应用. 然而, 并行程序错误(concurrency bug)对软件的可靠性构成了严重的威胁. 由于并行程序错误导致的软件系统错误, 造成了许多严重的事故和巨大的财产损失. 例如, 1987 年的 Therac-25 射线超标事件, 导致了多名患者不幸身亡; 2003 年的美国东北地区大面积停电事故, 影响了超过 5000 万人; 又如 2012 年脸谱(Facebook)公司的 IPO 故障事故, 造成了数亿美元的经济损失. 可见, 并行程序错误引起的软件故障可能造成严重的经济、社会影响, 甚至可能危及人的生命. 因此, 如何在有效进行并行程序调试成为了软件领域的重大问题, 引起了学术界和工业界的广泛关注.

调试并行程序错误一般从两个方面着手: 错误检测与测试调度. 错误检测着眼于每一次测试, 致力于发现每一次运行时的潜在错误. 测试调度则旨在尽可能多地暴露出程序可能的交互情况, 尽可能保证测试的覆盖率. 这两部分相互合作, 关系紧密. 在调试过程中, 对于每一次测试, 运用错误检测工具去判断程

① 基金项目:国家自然科学基金(6137081);国家高技术研究发展计划(863)(2012AA010901)
收稿时间:2014-03-04;收到修改稿时间:2014-03-31

序是否出错. 同时, 调试人员运用测试工具及技术构建尽可能多而全面的测试用例或测试交互情况.

然而, 调试并行程序错误要比串行程序的错误困难很多. 其原因主要有三点：

① 开发者们往往习惯于串行程序的思维模式, 导致了编写并行程序时难免会产生一些并行程序错误;

② 并行错误可以主要分为数据竞争, 原子性违背, 顺序违背和死锁几种, 同时, 不同类型的并行错误样式不同, 因此需要不同的检测方法进行检测, 不存在一种较为通用测方法;

③ 并行程序的运行具有不确定性(non-determinism). 由于并行程序执行过程中存在数量众多的交互情况(interleaving), 加之并行程序执行的不确定性, 使得错误的现场难以重现, 导致了调试并行程序错误要比串行程序的错误困难很多. 因此, 并行程序错误的调试成为了十分具有挑战性的问题.

本文对当前并行程序错误调试的热点进行了分类和归纳. 从并行错误检测与测试程序调度两个方向阐述该方向在并行错误调试中的意义, 并选择其中典型的设计实例进行详细说明. 在此基础上, 本文讨论与分析了各类型的调试方法的优劣; 最后, 本文对并行调试技术未来可能的发展方向进行了展望.

本文后续组织如下：首先, 第 1 章从并行错误类型和调试方法两个方面进行背景综述. 然后, 第 2 章和第 3 章分别介绍并行程序调试的两方面：错误检测和测试程序调度. 最后, 第 4 章对各类型并行错误调试技术的特征与优劣进行归纳与分析, 并提出对并行程序调试未来发展的展望.

1 并行错误分类与调试背景

本节中, 我们将从并行错误类型和并行程序调试方法两个角度进行背景介绍, 并加以讨论分析.

1.1 并行错误分类

根据错误类型的不同, 现有主要的并行程序错误可以分为非死锁错误(non-deadlock bugs)和死锁错误(deadlock bugs), 它们各占并行程序错误的 70%和 30%[1]. 其中, 非死锁错误可以进一步分为原子性违背(atomicity violation)和顺序违背(order violation). 这两种类型的错误占了非死锁错误的 97%以上. 因此, 从错误样式分类, 原子性违背、顺序违背和死锁是最普遍的三类并行程序错误.

同时, 从错误根源进行分析, 数据竞争(data race)可谓是并行错误中最臭名昭著的一员. 我们并没有把数据竞争归入错误样式的分类中, 原因在于并非所有数据竞争都是程序错误, 程序员甚至可能利用一些良性的数据竞争来进行同步, 如手动的 while 循环轮询(ad-hoc while flag checking). 但是, 不当的数据竞争却极易引发并行程序错误; 且 3%的非死锁并行程序错误的一部分也仅属于数据竞争, 无法归类于原子性违背或者顺序违背. 因此, 为了研究的完备性, 在该分类中, 我们将依次介绍这几种主要的并行程序错误的检测：数据竞争, 原子性违背, 顺序违背, 死锁和其他一些并行错误.

① 数据竞争：数据竞争是最臭名昭著的并行程序错误之一, 其定义为当两个访存操作同时访问同一变量, 且至少有一个访存为写操作. 这里的“同时”定义为这两个访存操作间没有合理的同步操作(synchronizations)进行保护, 如锁操作(lock)等. 如图 1 所示, 线程 1 和线程 2 都对 x 进行了写操作, 但由于没有正确的同步保护, 引发了数据竞争, 导致了此时的 x 的值并不确定.

Thread1	Thread2
x=1;	x=2;
printf(“%d”,x);	printf(“%d”,x);

图 1 数据竞争代码示例

② 原子性违背：原子性违背是近年来提出且十分受关注的并行程序错误. 它发生于一个线程的必须串行化(desired serializability)的代码区域与另一个线程出现交叉(interleaving), 导致该代码区域的串行化假定被破坏的情况. 具体示例如图 2 所示, 线程 2 代码(c)和(d)在锁的保护下读取字符串与其长度的值, 读取间被线程 1(a)和(b)打断, 导致了线程 2 的局部变量字符串值与长度值不符, 在后续操作将引发错误.

③ 顺序违背：顺序违背也是近年来广受关注的并行程序错误之一. 当一个线程的操作 A 本应当在另一个线程的操作 B 之前执行, 但实际却发生在 B 之后, 这就是一个顺序违背. 例如, 操作 A“读文件”本应该发生在操作 B“打开文件”之后, 但实际却提前到 B 之前发生, 从而导致错误. 如图 3 所示, 若是线程 2 的读

操作发生在线程 1 的打开文件操作前, 则会导致顺序违背.

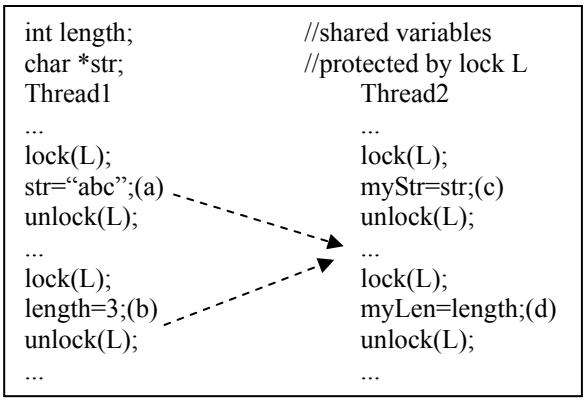

图 2 原子性违背代码示例

Thread1	Thread2
...	...
lock(L);	lock(L);
fopen(...);	read(...);
unlock(L);	unlock(L);
...	...

图 3 顺序违背代码示例

④ 死锁: 死锁发生于两个或者更多的线程循环等待其他线程释放它们已经获取的共享资源(如锁等), 但却没有任何一方可以打破这种等待状态. 如图 4 所示, 线程 1 和线程 2 分别拿到 L1 和 L2, 但由于必须拿齐 L1 和 L2 才能往下执行, 因此形成死锁.

Thread1	Thread2
...	...
lock(L1);	lock(L2);
lock(L2);	lock(L1);
...	...
unlock(L2);	unlock(L1);
unlock(L1);	unlock(L2);
...	...

图 4 死锁代码示例

⑤ 其他并行错误: 近年来, 随着宽松内存模型的推广, 一些仅在宽松内存模型下, 由于乱序提交所导致的并行错误也开始受到研究人员的关注. 串行一致性违背(Sequential Consistency Violations)便是这类错误的代表, 它发生于指令流下游的访存指令于上游访存指令之前提交, 从而导致的错误.

1.2 并行错误的调试

随着多核设备的普及, 为了充分利用设备的计算能力, 并行程序得到了广泛的应用. 然而, 程序中种类繁多的并行程序错误直接威胁了软件的健壮性, 造成了许多严重的事故和巨大的财产损失.

因此, 如何编写健壮的并行程序变得至关重要. 并行错误的调试主要分为两个方面: 错误检测与测试调度. 错误检测着眼于每一次测试, 致力于发现每一次运行时的潜在错误. 测试调度则旨在尽可能多地暴露出程序可能的交互情况, 尽可能保证测试的覆盖率. 这两部分相互合作, 关系紧密. 在调试过程中, 对于每一次测试, 运用错误检测工具去判断程序是否出错. 同时, 调试人员运用测试工具及技术构建尽可能多而全面的测试用例或测试交互情况.

然而, 相比串行程序, 并行程序的调试要困难很多. 主要原因在于:

① 并行程序的思维模式与串行程序的思维模式不同. 然而, 开发者们往往习惯于串行程序的思维模式, 导致了编写并行程序时难免会产生一些并行程序错误;

② 并行错误种类繁多, 不同类型的并行错误样式不同. 因此, 不存在一种较为通用的检测算法检测所有并行错误;

③ 并行程序具有不确定性, 使得错误的现场难以重现. 程序员往往无法通过反复跟踪某次错误用例来定位错误, 导致了调试并行程序错误要比串行程序的错误困难很多. 因此, 并行程序错误的调试成为了十分具有挑战性的问题.

接下来, 我们将从错误检测与测试调度两个方面来分析并行程序调试.

2 错误检测

错误检测, 即基于程序的状态, 通过相应的检测算法, 来判断程序中是否存在潜在的错误. 错误检测可以从检测方法和错误类型两个角度进行分类.

并行程序的错误检测方法主要可以分为两类: 静态检测与动态检测. 前者通过程序的源代码语义分析, 基于程序验证(program verification)[2]的思想与编译技术的方法, 采用符号执行(Symbolic execution)[3]等方式分析源码中问题所在. 后者则程序执行过程中收集相关信息(instrumentation), 并通过特定的算法分析来暴

露错误.

接下来, 我们将先从检测方法和不同错误类型的检测算法两个角度对错误检测进行归纳分析. 最后, 我们讨论分析现有的优化加速技术.

2.1 错误检测方法

错误检测方法主要可以分为两类：静态检测与动态检测. 前者通过程序的源代码语义分析进行离线(offline)的静态检测; 后者则依赖在程序执行过程中收集的信息进行线上(online)的检测.

① 静态检测方法：静态检测基于静态源代码分析, 通过程序验证或符号执行的方式分析源码语义或者程序控制流. 然而, 传统的静态检测往往受限于缺少程序运行时的一些状态和信息, 而不得不承受一些漏报(false-negative)和伪报(false-positive). 因此, 为了尽可能地获取运行时相关信息, 用于检测并行程序的静态工具, 往往会采用流敏感(flow-sensitive)的过程间分析, 来收集必要的信息. 其中典型如 RaceX[4], 它扫描并记录代码中加锁操作, 多线程的上下文, 和危险的共享访问等. 在此基础上, 采用符号执行等传统静态方法进行错误检测. 然而, 由于静态信息无法获取实际运行指令流等信息, 更无法体现程序动态运行时线程的交互情况, 限制了该类工具的检测精确性.

② 动态检测方法：动态检测通过在运行时插桩等方法获取并记录程序运行时状态(往往是访问信息和同步信息). 基于收集的信息结合相应的算法进行检测. 较之静态检测, 这样的做法能得到更准确的程序运行状态, 因此检测的精确性更高, 应用的也更加广泛.

2.2 并行错误分类及检测算法

如今, 主要的并行程序错误包括：数据竞争, 原子性违背, 顺序违背, 死锁和其他一些并行错误. 接下来, 我们分类别依次进行归纳分析.

2.2.1 数据竞争

数据竞争的检测算法分为两种类型：基于时间戳(timestamp)的检测算法和基于锁集合(lockset)的检测算法.

基于时间戳的检测算法：这类算法按照程序运行时的同步操作, 将程序划分为不同的执行段(epoch), 同时赋予每一个执行段对应的时间戳, 不同执行段内的缓存操作通过时间戳的比较来判断是否被正确的同步保护. Happens-before[5]算法是这类算法中的典型. 在运行阶段, Happens-before 算法为每个线程维护一个向量时间戳, 每条访存会被赋上当前线程的时间戳. 当线程发生同步操作(比如锁)时, 该线程的时间戳会向后推移. 检测阶段, 若可能导致数据竞争的两个访存的时间戳没有明确的前后关系, 则两个访存存在数据竞争.

基于锁集合的检测算法：这类算法根据程序运行时拿放锁的操作, 为程序中每一个线程维护一个锁集合. 通过比较不同变量的锁集合是否相交, 来判断访问是否被同一个锁保护, 即访问是否安全. 该方面的典型算法为 Lock-set[6]. 在运行阶段, Lock-set 算法为每个线程维护一个锁集合, 线程每次拿锁或者放锁, 都会更新这个集合. 同时, 算法为每个变量维护一个锁的集合(初始化为全集). 检测阶段, 当线程访问某个变量时, 算法将对变量锁集合与访问线程锁集合做交集, 若交集为空, 则视为有数据竞争.

我们在这里对这两类算法做一个简单总结和比较：这两个算法各有优劣, 基于时间戳的算法类型无法检测出本次交互情况未包含但可能潜在的数据竞争, 但可以保证不会有伪报. 基于锁集合的算法类型可以检测出潜在的数据竞争, 但却可能存在伪报. 近年来, Fasttrack[7]通过尽可能使用标量代替 Happens-before 的向量时间戳, 以此来提高算法效率.

数据竞争工具一般分为软件工具和硬件工具. 软件工具在程序运行时动态收集信息, 然后运用上述算法进行检测[7-9]. 硬件工具通过可以利用缓存一致性来判断是否存在共享变量, 并通过添加专用检测硬件进行检测[10,12].

软件方面, 基于上述检测算法构建的针对 C/C++ 程序的检测工具会造成高于 30 倍的性能开销[8,9]; 最好的针对 JAVA 程序的检测工具会造成 8.5 倍的性能开销[7]. 此外, LiteRace[10]通过选择检测部分程序而非全部来降低性能开销. KUDA[11], 通过利用 GPU 的大量计算资源并行地做 Happens-before 或者 Lock-set 的算法检测, 在算法检测方面取得了 3-14 倍的性能提升. Frost[12]则通过代码段级别的并行代替线程级别的并行, 利用额外的计算资源来降低检测开销.

硬件方面, 由于缓存一致性可以有效检测出共享变量, 一般都基于缓存一致性协议(Cache Coherence)进行构建. 该方面的典型设计如基于 Happens-before 算法的 CORD[13]和基于 Lock-set 算法的 HARD[14], 二

者都取得了良好的性能, 几乎不带来性能开销. RADISH[15]在二者基础上, 提出重用一部分缓存进行元数据的存放, 同时开发了相应的软件版本. 此外, 也有缓存分离的硬件设计, 其中典型为 SigRace[16]. SigRace 基于 Bloom Filter 产生的访存序列签名的粒度进行检测, 若这段访存存在问题, SigRace 则将程序回滚至该段签名之前, 而后将比较精度细化到每一条访存.

2.2.2 原子性违背

一般来说, 原子性违背的检测算法需要对访存进行串行序列处理(serializability). 而后, 通过检测一个串行访存序列是否被其他线程的同类访存所打断, 来判断是否存在原子性违背. 该方面的代表算法为 ColorSafe[17]. ColorSafe 首先对变量着色, 一起分配(malloc)空间或者程序员标注(annotation)的变量会被着上同一种颜色, 即归为一组. 当一个线程访问同样颜色的数据时, 则这多次访问被视为必须串行序列处理的代码区域. 若另一个线程对同样颜色数据的访问与该串行序列处理区域出现交叉(interleaving), 则视为串行序列处理序列被破坏, 算法将报告一个原子性违背.

原子性违背的检测一般利用缓存耦合的硬件支持, 对所有的访存进行动态收集, 然后运用上述算法进行解决[17,18]. 其中, AtomAid[18]将程序先划分为相应的“原子块”, 而后检测“原子块”被破坏的缓存行为, 它出了软硬件两种解决的版本. 而 ColorSafe[17]则基于上述 ColorSafe 算法,通过“着色”判定“原子块”, 再利用缓存行与缓存一致性协议复用, 提出了高效的硬件解决方案.

2.2.3 顺序违背

一般的顺序违背检测算法是基于时间戳的数据竞争检测算法进行修改, 在判断两个操作的先后关系后, 再检测这两个操作的先后关系是否合理. 典型算法如结合 Happens-before 算法与规则检测, 检测用例如下: 通过 Happens-before 算法检测出, 操作 A“读文件”发生在操作 B“打开文件”之前, 而后检测发现该关系不合理(必须先“打开文件”而后才能“读文件”), 因此报告该顺序违背.

顺序违背检测工具一般都是基于软件实现, 原因在于顺序违背不需要收集每一条缓存, 仅仅收集相关的操作(如文件操作等等)和生成时间戳所需的同步操作. 这些操作的数量远远少于缓存的数量, 因此, 便没有动态收集信息和检测导致的性能瓶颈. 2ndStriker[19]和 ConMen[20]是该方面的代表工具, 涉及了文件操作、指针操作、变量初始化检查等等顺序违背的检测.

2.2.4 死锁

死锁的检测包括静态的模型验证方法. 可以通过并发有穷状态机进行检测, 其中的典型代表为 JPF[2]; 也可以通过动态的监控方法, 使用超时机制, 加之动态记录资源调度情况, 来判断资源分配是否构成“锁环”, 如监控方法的代表工具 Pulse[21].

2.2.5 其他

在宽松内存模型下, 串行一致性违背也受到了广泛的关注. 其中, Vulcan[22]通过额外硬件检测不同线程访存顺序是否成环来支持该类型错误的检测.

2.3 错误检测优化方法

检测算法提供了错误检测的方法论, 如何利用这些算法去构建高效的检测工具显得至关重要. 一般来说, 构建的方法可以分为两类: 一是在现有体系结构上提供软件版本的支持; 二是提出新式的体系结构, 提供硬件平台上的支持.

2.3.1 软件工具

这类工具的好处在于软件版本的支持形式可以即刻供开发人员使用, 其中典型如数据竞争的检测工具 ThreadSanitizer[8]和 FastTrack[7], 原子性违背检测工具 AtomAid[18]以及顺序违背检测工具 ConMen[20]. 但是, 这类工具的瓶颈在于动态收集信息与算法检测带来的巨大开销. 例如, 使用 ThreadSanitizer 进行数据竞争检测会带来 30 倍以上的性能开销. 解决该开销的方法主要有三种思路: ① 取样, 仅仅选取程序的一部分进行检测; ② 并行加速, 利用并行技术加速检测过程; ③ 代码段并行, 通过代码段并行方式代替线程并行方式, 同时利用额外的硬件资源进行错误检测. 接下来, 我们将详细解释各种技术, 并给出其中的典型例子.

① 取样: 取样(Sampling)技术, 顾名思义, 即对整个程序进行“取样”, 挑选出一些具有代表性的“样本”以供检测. 该技术可以用来解决动态收集信息性能上的瓶颈. 通过该技术, 检测工具可以仅仅检测整个程序的一些关键部分, 因此降低整体的性能开销. 然而, 由于舍弃了大部分的访存信息, 该技术存在检测精确性损失的风险. 该类技术的典型应用为 LiteRace[10].

LiteRace 通过取样技术仅收集检测程序中 2%的访存. 因此, 它的额外性能开销仅为 28%. 然而, 由于舍弃了大部分的访存信息, 它不得不承受 30%的精度损失.

② 并行加速：该方法利用多核甚至众核的计算资源, 将检测算法并行化至多核上, 解决算法检测性能上的瓶颈. 然而, 该技术仍无法逾越动态收集信息产生的性能屏障. 该方面的典型应用为 KUDA[11]. KUDA 利用 GPU 并行加速检测算法, 取得了检测阶段 3-14 倍的性能提升. 然而, KUDA 无法避开动态收集信息带来的 7 倍左右的性能开销.

③ 代码段并行：该方式将程序划分成代码段(epoch), 通过代码段级别的并行代替线程级别的并行[12]. 利用代码段并行加速检测算法, 可以取得比线程级别并行更好的可伸缩性(scalability)[23]. 另一方面, 通过多份副本同时运行并比较结果异同来判断是否错误[24], 这种检测方法称为基于代码段结果(epoch outcome-based)的检测技术, 其优势在于低性能开销. 但是, 代码段并行却需要耗费三倍的计算资源. Frost[12]为该方面典型应用, 在额外消耗 3 倍硬件资源的前提下, Frost 能将性能开销控制在 12%之内.

2.3.2 硬件结构

由于软件检测工具几乎都不得不面对难以逾越的动态收集信息开销. 因此, 通过提供新型的硬件检测架构, 可以有效地加速检测的过程. 这其中的典型架构如 SigRace[16], RADISH[15]和 ColorSafe[17]. 较之软件工具, 硬件架构的优势在于可以通过添加额外硬件来分担错误检测的计算负载, 从而做到高效的错误检测. 然而, 由于依赖特定且有限的硬件支持, 该方案不仅在实际性和灵活性有所欠缺, 更在可伸缩性(scalability)和可扩展性方面存在较大挑战. 接下来, 我们按照硬件架构收集访存信息的方式, 将众多具体实现分为两类：

① 缓存耦合：由于缓存一致性可以判断共享变量, 而共享变量是并行错误的必要条件, 因此, 可以利用缓存来辅助并行错误的检测. 该方面的典型应用如 HARD[14]和 RADISH[15]. 基于缓存(Cache)结构的硬件设计通过在缓存行(Cache Line)中加入所需的元数据(Mata Data), 并引入相应的检测单元, 利用缓存一致性协议(Cache Coherence)进行数据通信. 由于复用了缓存协议, 该方面设计几乎不带来性能开销. 然而, 基于缓存的架构会在包含数据的缓存行被替换的时候遭受精度损失, 同时, 其可伸缩性(scalability)也受缓存一致性协议的限制.

② 缓存分离：由于基于缓存的架构都不可避免需要面临缓存行被替换的问题, 通过缓存分离的架构, 将元数据存储于外部专有结构成为了另一种思路. 该方面的典型设计为 SigRace[16]. 该类型的架构通过在处理器收集访存信息, 并发送到片上专有的处理硬件进行后续的处理. 该方面设计的难点在于如何有效地存储足够多的历史元数据并减轻比较负担.

3 错误测试

传统的软件测试旨在通过静态构建不同的输入数据来检测软件的健壮性. 并行程序的测试则更加复杂. 由于并行程序有着众多的交互情况, 而并行错误的触发通常依赖一些小概率事件的复杂的执行序列, 因此, 这些错误往往隐藏在一些罕见的交互情况之中[25]. 传统的静态构建测试缺乏程序运行时的动态信息, 为了更好地暴露出罕见的交互情况, 测试工具必须了解当前程序状态, 并进行必要的线程调度[25]. 因此, 相比于静态测试技术, 动态测试在并行程序方面应用更为广泛. 基于这点, 本文将着重介绍动态测试技术于并行程序测试方面的应用.

不同于传统的静态测试工具, 动态测试工具会在程序运行过程中进行线程的调度, 旨在测试出更多不同的交互情况, 其中的典型方法或工具如压力测试, Ctrigger[25], PCT[26]和 Chess[27]. 接下来, 本节将按照调度的策略进行介绍.

3.1 压力测试

压力测试不对线程调度进行干预, 仅仅通过大量的重复测试样本来触发可能存在的并行程序错误. 由于部署简单, 且可以保持程序原本的并行度, 压力测试在业界被大量应用.

3.2 启发式引导

启发式引导测试实时监控程序运行行为, 并根据结果进行线程调度. 首先, 测试工具先预先执行程序, 记录其中共享内存访问和同步信息; 然后, 测试工具基于预设规则提取“可疑”行为. 提取规则基于“越是罕见的线程调度方式越可能存在缺陷, 并且难以被发现”这一观察. 举例来说, 线程 A 连续前后两次对地址 x 进行了写访问操作(Wr x@A), 与此同时, 线程 B 中存在一次对 x 的读访问操作(Rd x@B)(该读访问操作

可能发生在两次写访问操作之间). 该例子中, “Wr x@A -- Rd x@B -- Wr x@A”最难出现, 因此最为“可疑”; 最后, 测试工具以最“可疑”的方式调度程序. 其中的典型用例如 Ctrigger[25].

3.3 随机调度

随机调度, 即通过在调度中通过对系统的调度机制的修改(利用插入随机延迟, 随机上下文切换, 或者改变线程优先级等方式), 从而加剧操作系统调度的随机性. 随机调度[26]的子策略种类较多, 一般来说, 可以分为以下几种:

① 随机休眠: 随机休眠是随机于程序中选取若干个休眠点, 其典型用例如 Ben-Asher Y 等人提出的调度工具[28]. 测试过程中, 当遇到休眠点时, 调度机制就进行一次不定时长的系统休眠, 以此加剧线程调度的随机性.

② 随机抢占: 类似于随机休眠, 该策略在程序中预先选取一系列抢占点(一般选取同步操作等关键抢占时机). 当程序运行到抢占点时, 系统随机选取其他线程进行切换. PCT[26]工具中提供了该策略的实现.

③ 策略抢占: 该调度基于随机抢占, 不同于随机进行线程选择, 该策略对各个线程按照其触发错误的可能性划分了优先级, 并倾向于切换给优先级高的线程, 以此加大测试出错误的可能性. 同时, 该策略减少调度点数目, 并在调度点改变各个线程的优先级. PCT[26]便是该策略的典型应用.

3.4 系统调度

该调度策略基于模型检验技术, 其目的在于不仅仅提供一种高概率的调度策略, 而是要断言测试程序是否存在错误, 其典型应用如 Chess[27].

系统调度会穷举一切可能的程序状态空间, 并一一进行测试, 以此来保证测试的正确性.

当然, 一般来说, 在实际的测试环境中, 穷尽所有程序状态空间是不现实的. 因此, 实际的系统调度策略会着眼于程序中关键的抢占时机, 仅仅在这些时机进行系统调度, 穷尽一切可能性.

4 分析与展望

本节中, 我们将对各类型并行错误调试技术的特征与优劣进行归纳与分析, 并对未来可能的发展方向进行了展望.

4.1 错误检测

4.1.1 现状分析

表 1 对本文提及的检测技术进行了总结与比较.

表 1 各种检测技术比较

类别			运行时开销	精度损失	额外资源
静态	/	/	无(线下检测)	是	无
动态	软件工具	取样	低(<1.5x)	是(30%)	无
		并行加速	高(10x~200x)	否	无
		代码段并行	低(< 2x)	否	三倍计算资源
	硬件支持	缓存紧耦合	极低(< 1.05x)	是(20%)	硬件修改
		缓存分离/松耦合	低(<1.25x)	否	硬件修改

① 静态检测: 静态检测工具的好处在于可以离线进行, 因此对于大规模代码则更有优势, 可以完成一些动态检测无法执行的测试. 但是, 由于源码等静态信息无法获取实际运行指令流等信息, 加之并行程序数量众多的线程交错导致了其运行的不确定性, 导致了其运行时状态更加难以预估. , 因此静态检测的精确性不足.

② 动态检测: 得益于可以获知程序运行状态, 动态检测往往较静态检测所依赖的源代码信息全面而准确. 动态检测的方法主要包括软件工具和硬件支持两个部分.

(1) 软件工具

软件工具的优势在于可以即刻为开发人员使用, 但最大问题在于难以逾越动态收集信息的瓶颈. 现有软件工具往往是检测覆盖率或者计算资源和收集瓶颈间的权衡:

① 取样方法: 仅仅检测所选取的程序关键 “样本”, 从而降低整体的性能开销. 然而, 由于舍弃了大部分的访存信息, 该技术存在检测精确性损失的风险.

② 并行加速: 将检测算法并行化至多核上, 解决算法检测性能上的瓶颈. 然而, 该方法仍无法逾越动态收集信息产生的性能屏障.

③ 代码段并行: 利用额外的计算资源同时运行多份程序副本, 基于各副本结果来检测错误, 虽然不需要进行动态信息收集, 但却需要耗费三倍的计算资源.

(2) 硬件支持

硬件支持的优势在于可以较好地解决信息收集的瓶颈, 但主要有两大缺点: ① 硬件支持都必须面临所添加硬件资源的有限性, 导致其所能存储的历史记录有限, 从而影响其检测窗口的长度. ② 由于所添加的

硬件资源都是单一目的的, 因此都只能针对某种特定的并行错误, 影响了其通用性. 具体而言：

① 缓存紧耦合：这类方案优点在于几乎不带来性能开销, 但在缓存行被替换的时候会丢失信息, 造成精度损失, 且可伸缩性受到缓存一致性协议的限制.

② 缓存分离/松耦合：这类结构虽然成功避开了缓存行替换时的精度损失问题, 但却不得不承受回滚代价或者替换缓存行维护代价, 因此性能较前者差.

现状总结：由于静态检测受限于精确性, 近年来的研究多集中于动态检测. 其中, 软件方面, 有两个瓶颈问题：一是收集访存的开销, 二是执行错误检测算法的开销. 对于前者, 取样方法提供了速度与精确性间的权衡, 代码段并行提供了速度与额外硬件资源间的权衡; 对于后者, 通常采用并行手段进行加速. 硬件方面, 同样有两个瓶颈问题：一是片上硬件资源的有限性, 二是硬件架构的灵活性. 对于前者, 重用缓存或者利用软件辅助可以在一定程度上解决问题; 对于后者, 目前的硬件架构还没有该方面的研究.

趋势总结：动态检测是近年来研究的热点. 其中, 软件检测方面, 研究集中在如何逾越动态收集信息的瓶颈, 加速检测算法方面较少. 硬件检测方面, 研究热点由五年前的缓存紧耦合结构转移到近几年缓存分离/松耦合结构的设计. 其原因在于, 在如今并行计算线程数量日益攀升的趋势下, 松耦合架构更有利于可伸缩性.

4.1.2 挑战与展望

基于以上分析, 可以看出目前的检测手段存在如下挑战：① 居高不下的访存收集开销极大地限制了软件工具的速度; ② 如何获取检测所需计算能力与如何维护历史元数据是硬件系统所面临的最大挑战.

针对这两方面的挑战, 我们提出了两个值得探索的方向：

① 如今, 众核通用计算设备(如 GPU)的普及, 极大增强了硬件的计算能力, 有效利用这些设备的计算能力和存储空间于错误检测, 将解决硬件架构计算和存储资源的问题. 例如, 可以通过少许的硬件支持收集动态信息, 解决收集瓶颈, 而后利用 GPU 等众核设备进行检测加速, 并在各个模块之间利用流水线技术进行并行化, 来解决性能问题.

② 通过动静态检测方式的结合, 利用静态检测优化动态检测的负载量问题, 利用动态检测解决静态检测的精确度问题. 例如, 使用静态的编译器优化与适当的程序员标注减轻插桩带来的开销, 然后在此基础上进行动态检测.

4.2 测试

4.2.1 现状分析

表 2 各种测试调度策略比较

类别		错误触发率	速度	需要专用工具
压力测试		低	快(并行)	否
启发式引导		高	慢(串行)	是
随机测试	随机休眠	低	快(并行)	否(修改程序)
	随机抢占	高	慢(串行)	是
	策略抢占	高	慢(串行)	是
系统调度		全覆盖	极慢(串行且需穷尽程序状态空间)	是

表 2 从错误触发率和运行速度两个方面, 对本文提及的测试技术进行了总结与比较. 可以得出以下结论：

① 压力测试：虽然错误触发率不如其他的测试技术, 但优势在于简单直接, 部署方便, 运行速度快, 因此应用广泛.

② 启发式引导：基于随机抢占/策略抢占的随机调度和系统调度错误触发率高, 甚至可以达到 100%(系统调度). 但是, 它们需要专用的调度工具获知程序当前运行状态, 以此做出相应决策. 因此测试程序无法保持原有的并行粒度, 往往采用单线程调度, 运行速度较慢.

③ 随机测试：基于随机休眠的随机调度介于上述二者之前, 不需要专用的调度工具, 仅仅需要对源码进行一定量的插桩. 可以作为以上二者的折中.

④ 系统调度：系统调度优势在于能够断言测试程序是否存在错误, 但其劣势在于需要穷举一切可能的程序状态空间, 因此需要专门的调度工具进行一一调度, 速度瓶颈十分严重.

现状总结：虽然较之其他测试手段, 压力测试被应用得最为广泛, 但由于它的实现简单直接, 该方面的研究已经比较成熟. 因此, 近年来的研究多集中于两个方面：一是如何进一步提高随机测试或者启发式引导测试的错误触发率; 二是如何降低测试的开销. 提高错误触发率方面, 一般使用在关键区域(critical section)(如同步区域)插桩调度的方法. 若需要更加细致的调度, 则使用运行时监测工具, 根据程序当前状态进行调度. 降低测试开销方面, 通常采用并行手段

优化.

趋势总结：近年来, 为了能够进一步提高错误的触发率, 并行错误测试方面的研究热点由简单的随机测试转移到了较为复杂的启发式引导测试或策略抢占测试. 另外, 由于复杂的测试手段存在速度方面的瓶颈, 如何优化测试速度也越来越受研究人员的关心.

4.2.2 挑战与展望

基于以上分析, 可以看出：虽然现有的测试技术已经能在一定程度上暴露和发现程序潜在错误, 但在测试完备性, 可扩展性和测试速度方面, 还存在许多挑战：

① 系统调度测试技术的执行代价随着程序空间的增长而增长, 因此, 目前对复杂的程序进行系统调度测试的性能开销往往是实际无法承受的. 如何有效裁剪程序运行空间, 既优化系统调度测试的测试速度, 又保证测试的完备性, 依然需要进一步探索;

② 目前专用调试工具都受限于单线程调试, 当程序线程数目变多时, 现有测试技术的单线程测试却无法利用如今多核硬件资源, 导致其可扩展性方面的极大缺陷, 性能方面的瓶颈将愈发严重. 因此, 测试的速度与可扩展性问题也是亟待解决的;

③ 随机抢占, 策略抢占或者启发式测试中, 往往需要在关键区域执行调度算法, 如何提高调度算法性能, 做到实时无延迟的调度, 也是挑战之一;

④ 目前的并行程序测试仅仅暴露出错误的表面表现形式, 但无法指出错误的本质根源. 只有在调度测试的同时辅助于错误检测, 才能够最根本地提升并行错误调试的效率.

针对这几方面的挑战, 我们提出了以下值得探索的方向：

① 动静结合的方法：如今, 符号执行方法提出了状态合并的理念, 即减少符号执行引擎执行时的状态数. 因此, 如何利用程序员标注关键区域、编译器优化或者符号执行等静态方法预先对系统空间进行裁减, 从而减少程序的状态, 再加以动态方法进行测试, 是值得探索的方向;

② 不同粒度的测试区域方法：测试中的有效调度主要集中在一些关键区域的同步原语的顺序. 因此, 在非关键区域, 采取并行多线程调度, 加速调试过程; 在关键区域, 再应用调试算法细粒度调度. 这种粗细粒度结合的调度策略, 可以加速整个调度的过程.

③ 并行错误测试的软件加速技术：测试调度时的软件速度问题可能随着调度情况的复杂愈发严重, 但目前解决速度问题不突出, 也无法适应将来众核平台环境. 该方面, 可以利用众核通用计算设备(如 GPU)的计算能力和存储空间进行并行化, 从而解决问题.

④ 针对并行错误统一的测试方法：程度调度可以暴露出更多的交互情况, 若是能在调度的同时加以错误检测工具的检测, 就能够更加统一且有效地解决并行错误调试的问题. 因此, 如何利用现在平台丰富的计算资源, 在程序调度的同时加以错误检测, 将这两个方面有机地结合起来, 是值得探索的方向.

5 结语

随着多核设备的普及, 为了充分利用计算资源, 多线程的并行编程时代已经来临. 然而, 由于并行程序的不确定性, 使得调试并行程序错误要比串行程序的错误困难很多, 引起了学术界和工业界的广泛关注. 本文对并行程序错误调试技术进行了研究与分析, 包括错误检测和测试调度两个方面, 分别进行了总结归纳. 其中, 错误检测的主要瓶颈在于软件实现中的访存收集开销和硬件实现中的额外计算存储资源开销. 利用编译器和程序员标注优化缓存开销, 与利用众核设备解决硬件资源开销是该方面的两个解决方向. 测试调度方面, 问题主要存在于测试完备性与测试速度之间的权衡. 如何通过静态工具对系统空间的裁剪, 以减少有效程序状态空间, 是值得尝试的方向. 最后, 面对日趋复杂的并行程序, 如何充分利用高效的算法与计算资源进行错误调试, 将是一个巨大的挑战, 也是值得期待的努力目标.

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. Operating Systems Review, 2008, 42(2): 329–339.

2 Visser W, Havelund K, Brat G, Park S, Lerda F. Model checking programs. Automated Software Engineering, 2003, 10(2): 203–232.

3 King JC. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385–394.

4 Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03).

ACM. New York, NY, USA. 2003. 237–252.
5 Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558–565.
6 Savage S, Burrows M, Nelson G, et al. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems (TOCS), 1997, 15(4): 391–411.
7 Flanagan C, Freund SN. FastTrack: efficient and precise dynamic race detection. ACM Sigplan Notices. ACM, 2009, 44(6): 121–133.
8 Serebryany K, Iskhodzhanov T. ThreadSanitizer: data race detection in practice. Proc. of the Workshop on Binary Instrumentation and Applications. ACM, 2009. 62–71.
9 Sack P, Bliss B E, Ma Z, et al. Accurate and efficient filtering for the intel thread checker race detector. Proc. of the 1st Workshop on Architectural and System Support for Improving Software Dependability. ACM, 2006. 34–41.
10 Marino D, Musuvathi M, Narayanasamy S. LiteRace: effective sampling for lightweight data-race detection. ACM Sigplan Notices. ACM, 2009, 44(6): 134–143.
11 Bekar UC, Elmas T, Okur S, et al. KUDA: GPU accelerated split race checker. Workshop on Determinism and Correctness in Parallel Programming (WoDet). London, England, UK. 2012.
12 Veeraraghavan K, Lee D, Wester B, et al. DoublePlay: Parallelizing sequential logging and replay. ACM Trans. on Computer Systems (TOCS), 2012, 30(1): 3.
13 Prvulovic M. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. High-Performance Computer Architecture, 2006. The Twelfth International Symposium on. IEEE, 2006. 232–243.
14 Zhou P, Teodorescu R, Zhou Y. HARD: Hardware-assisted lockset-based race detection. High Performance Computer Architecture. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2007. 121–132.
15 Devietti J, Wood BP, Strauss K, et al. RADISH: always-on sound and complete Ra Detection in Software and H ardware. ACM SIGARCH Computer Architecture News. IEEE Computer Society, 2012,40(3):201–212.
16 Muzahid A, Suárez D, Qi S, et al. SigRace: signature-based data race detection. ACM SIGARCH Computer Architecture News. ACM, 2009, 37(3): 337–348.
17 Lucia B, Ceze L, Strauss K. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. ACM SIGARCH Computer Architecture News. ACM, 2010, 38(3): 222–233.
18 Lu S, Tucek J, Qin F, et al. AVIO: Detecting atomicity violations via access interleaving invariants. ACM SIGOPS Operating Systems Review. ACM, 2006, 40(5): 37–48.
19 Gao Q, Zhang W, Chen Z, et al. 2ndStrike: Toward manifesting hidden concurrency typestate bugs. ACM SIGPLAN Notices, 2011, 46(3): 239–250.
20 Zhang W, Sun C, Lu S. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. ACM Sigplan Notices, 2010, 45(3): 179–192.
21 Li T, Ellis CS, Lebeck AR, et al. Pulse: A dynamic deadlock detection mechanism using speculative execution. USENIX Annual Technical Conference, General Track. 2005. 31–44.
22 Muzahid A, Qi S, Torrellas J. Vulcan: Hardware support for detecting sequential consistency violations dynamically. Microarchitecture (MICRO), 2012 45th Annual IEEE. ACM International Symposium on. IEEE, 2012. 363–375.
23 Wester B, Devecsery D, Chen P M, et al. Parallelizing data race detection. Proc. of the eighteenth international conference on Architectural support for programming languages and operating systems. ACM, 2013. 27–38.
24 Veeraraghavan K, Chen PM, Flinn J, et al. Detecting and surviving data races using complementary schedules. Proc. of the 23rd ACM Symposium on Operating Systems Principles. ACM, 2011. 369–384.
25 Park S, Lu S, Zhou Y. CTrigger: Exposing atomicity violation bugs from their hiding places. ACM Sigplan Notices, 2009, 44(3): 25–36.
26 Burckhardt S, Kothari P, Musuvathi M, et al. A randomized scheduler with probabilistic guarantees of finding bugs. ACM Sigplan Notices. ACM, 2010, 45(3): 167–178.
27 Musuvathi M, Qadeer S, Ball T, et al. Finding and reproducing heisenbugs in concurrent programs. Proc. of the 8th USENIX conference on Operating systems design and implementation. USENIX Association. 2008. 267–280.
28 Ben-Asher Y, Eytani Y, Farchi E, et al. Producing scheduling that causes concurrent programs to fail. Proc. of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging. ACM, 2006. 37–40.