软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(4):747−763 [doi: 10.13328/j.cnki.jos.005193]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

获取访存依赖:并发程序动态分析基础技术综述*

蒋炎岩[1,2], 许 畅[1,2], 马晓星[1,2], 吕 建[1,2]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](南京大学 计算机科学与技术系,江苏 南京 210023)

通讯作者: 马晓星, E-mail: xxm@nju.edu.cn

摘 要: 并发错误难触发、难调试、难检测.为应对这一挑战,已有动态程序分析技术通过观测或控制并发程序执行实现其质量保障.由于并发程序不确定性主要来自共享内存,实现其动态分析的基本问题即是获取线程访问共享内存的顺序,即获取访存依赖.提出访存依赖获取技术的综述框架,包含4个评价指标(即时性、准确性、高效性、简化性)、两种方法(在线追踪、离线合成)、两类应用(轨迹分析、并发控制).通过对已有技术的总结和分析框架中的空白,对未来可能的研究方向予以展望.

关键词: 并发;多处理器系统;动态分析;访存依赖

中图法分类号: TP311

中文引用格式: 蒋炎岩,许畅,马晓星,吕建.获取访存依赖:并发程序动态分析基础技术综述.软件学报,2017,28(4):747−763. http://www.jos.org.cn/1000-9825/5193.htm

英文引用格式: Jiang YY, Xu C, Ma XX, Lü J. Approaches to obtaining shared memory dependences for dynamic analysis of concurrent programs: A survey. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):747−763 (in Chinese). http://www.jos.org.cn/1000-9825/5193.htm

Approaches to Obtaining Shared Memory Dependences for Dynamic Analysis of Concurrent Programs: A Survey

JIANG Yan-Yan[1,2], XU Chang[1,2], MA Xiao-Xing[1,2], LÜ Jian[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract: Concurrent bugs are difficult to trigger, debug, and detect. Dynamic program analysis techniques have been proven useful in addressing such challenges. Due to non-deterministic nature of concurrent programs in which the major source of non-determinism is the shared memory, obtaining the order of shared memory accesses, i.e., shared memory dependences, is the basis of such dynamic analyses. This work proposes a survey framework to demonstrate the key issues in obtaining the shared memory dependences. The framework includes four performance metrics (immediacy, accuracy, efficiency and simplicity), two categories of approaches (online tracing and offline synthesis), and two categories of applications (trace analysis and concurrency control). Existing techniques as well as potential future work are studied in the paper.

Key words: concurrency; multi-processor system; dynamic analysis; shared memory dependence

单处理器的性能瓶颈致使共享内存多处理器系统迅速普及[1].此类系统通常对线程访存不作任何限制,运

* 基金项目: 国家重点基础研究发展计划(973)(2015CB352202); 国家自然科学基金(61472177, 61690204); 江苏省软件新技术与产业化协同创新中心资助

Foundation item: National Basic Research Program of China (973) (2015CB352202); National Natural Science Foundation of China (61472177, 61690204); the Collaborative Innovation Center for Novel Software Tehcnology and Industralization

收稿时间: 2016-06-19; 修改时间: 2016-09-08; 采用时间: 2016-11-26; jos 在线出版时间: 2017-01-24

CNKI 网络优先出版: 2017-02-20 13:51:11, http://www.cnki.net/kcms/detail/11.2560.TP.20170220.1351.010.html

行在其上的并发程序因此存在数据竞争、死锁、原子性违反、序列化错误等串行程序中不会发生的错误,并可能致使产生严重后果[2].然而,由于并发程序具有不确定性,并发错误的触发不仅需要特定的输入,还需要特定的线程调度,导致并发错误难触发、难复现、难检测.

为应对这一挑战,并发程序的动态分析技术对程序进行插装或对运行系统进行修改,然后观测并发程序的运行时行为,从而实现错误检测、错误定位、并发控制等机能.这类动态分析技术已在并发程序质量保障方面取得了卓著的成效,研究成果主要体现在两个方面:(1) 轨迹分析,如动态数据流分析[3]、并发错误检测[4,5]、轨迹重放调试[6]等;(2) 运行时并发控制,如事务内存[7−9]、数据竞争避免[10,11]和确定多线程[12,13]等.由于并发性存在于计算机系统栈的各个层次,根据研究对象、研究手段的不同,现有研究工作横跨体系结构、操作系统、程序设计语言和软件工程领域.

这类动态分析技术的基础是在运行时观测、记录并发程序的行为,通过分析、控制或消除程序执行中的不确定性,从而达到检测、复现或避免并发错误的目的.在共享内存多处理器系统中,并发程序执行结果的不确定性主要来自共享内存访问(由多核处理器[1]、系统调度[14]等原因引起):若两次执行同一程序但访存顺序不同,则程序的输出可能大相径庭;反之,若记录同一内存地址上的线程访问顺序,则可确保复现过往的执行结果.定义这一问题为获取访存依赖(共享内存访问依赖,shared memory dependences,即线程访问共享内存的顺序),其成为并发程序动态分析技术的基石.准确、高效地获得访存依赖是并发程序动态分析技术走向实用的重大挑战之一.

国内外已有一些并发程序动态分析技术的综述论文,然而现有的研究通常只局限于某个应用领域(一类动态分析技术的研究现状、挑战和未来工作),如执行重放[15,16]、并发错误检测和避免[17,18]等,未能系统、深入地阐明这些领域之间的内在联系(即本综述提出的访存依赖获取问题),亦未充分讨论由此带来的研究契机.

本文立足访存依赖获取这一并发程序动态分析的奠基性问题,对其 3 个要素——评价指标、实现方法和应用予以综述,涉及多个应用技术领域的相关研究,力图从更高层次阐明现有技术在各个方面做出的不同权衡及其原因,以全新的角度更好地认识现有技术的优势与不足,进而提出未来可行的研究方向.本文主要内容包括:

1. 对并发程序、并发执行、访存依赖进行形式化定义,明确研究对象和问题(见第 1 节).
2. 提出获取访存依赖的基础框架,包含 3 个要素及每个要素的分类框架.
   a) 4 个评价指标(见第 2.1 节):即时性、准确性、高效性和简化性.即时性要求能在访存发生前即获得依赖;准确性要求获得的访存依赖能尽可能真实地反映实际程序执行的访存顺序;高效性要求其获取开销尽可能地小;简化性要求获得的日志尽可能精简;
   b) 两类现有的访存依赖获取技术(见第 2.2 节):在线追踪和离线合成.在线获追踪在程序运行时即时捕获访存依赖;而离线合成则通过记录间接信息,使用推导或搜索算法合成过去执行中的访存依赖;
   c) 两类访存依赖的应用(见第 2.3 节):轨迹分析和并发控制.轨迹分析技术利用在线记录或离线合成的访存依赖实现测试、调试等功效;并发控制技术则依靠实时获得的访存依赖约束程序行为,从而实现错误避免.
3. 基于第 2.2 节的访存依赖获取技术框架,系统化地总结现有并发程序动态分析技术中使用的访存依赖获取方法(见第 3 节),试图理解各类技术之间的区别和联系.即时获取访存依赖可采用在线追踪法,此类技术通常具有准确性,但要付出较大的运行时开销,高效性和简化性不足.追踪法的设计关键是锁的指派和锁的实现,如允许放宽对即时性的要求,则可以使用合成法在运行时记录较少的信息,但要付出一定的合成代价,同时可能牺牲准确性.合成法的设计关键是获取何种信息以权衡合成代价和准确性、高效性和简化性.
4. 基于第2.3节的访存依赖应用框架,总结访存依赖的两类应用(见第4节):轨迹分析和并发控制.为实现并发程序的测试、调试和质量保障,轨迹分析技术依托访存依赖对过往的执行进行重放、分析或错误预测,亦是众多并发程序测试技术的基础;并发控制技术则在程序员的指导下或全自动地对程序运行时可能出现的错误予以规避.由于这类动态分析技术数量繁多且已有相关综述文献[18],仅选取具有代表性的一些工作,对这些技术

与访存依赖之间的关系予以阐释.

5. 基于提出的框架,探讨访存依赖相关研究的契机(见第 5 节).我们已在此框架的指导下取得了一定的研究进展,如缓存制导的执行重放[19]、基于偏向锁的依赖获取[20]和基于地址空间划分的依赖简化[21],并在此基础上提出 4 个未来可能的研究方向:融合现有技术、实现锁的自适应动态指派和实现、记录额外信息以实现高效的依赖合成以及对现实中的并发程序执行予以建模.

1 问题定义

1.1 并发程序的执行模型

首先,在一个简化的并发系统上给出基于字节码的程序及其执行的形式化定义,以便展开访存依赖的讨论.定义并发程序 $P=\langle T,X,I\rangle$为三元组,包含:

1. 线程集合 $T=\{t_1,t_2,\ldots,t_n\}$,t_i 是编号为 i 线程的标识符;
2. 变量集合 X,由共享变量集合 $X_s=\{x_1,x_2,\ldots,x_k\}$及每个线程 t 的局部变量集合 $X_t=\{x_1^t,x_2^t,\ldots,x_k^t\}$ 构成.X_s 中的共享变量任意线程均可访问,x_i^t 则仅限线程 t 访问;
3. 指令序列 $I=\{i_1,i_2,\ldots,i_m\}$,其中,指令包括:
   a) 根据局部变量 x_k^t 数值跳转:if (x_k^t) goto i_j;
   b) 读取共享内存变量 x_k^t : $x_k^t=R(x_{k'})$;
   c) 写入共享内存变量 x_k^t : $W(x_k,x_{k'}^t)$;
   d) 局部变量运算: $x_k^t=op(x_k^t,x_{k'}^t)$.

定义并发程序 $P=\langle T,X,I\rangle$在运行时的状态$\sigma=(V,PC,E)$,其中,

- $V:X\mapsto N$ 将每个变量映射到其当前数值;
- $PC:T\mapsto[m]$将每个线程标识符映射到其当前执行的指令;
- $E=\{e_1,e_2,\ldots\}$是访存事件日志的集合,包含执行历史中所有的访存事件.其中,$e_i=W(t_i,x_i,v_i)$代表线程 t_i 向变量 x_i 写入值 v_i,$e_i=R(t_i,x_i,v_i)$代表线程 t_i 从变量 x_i 读出值 v_i.

初始状态中,$V(x)=0$,$PC(t)=1$,$E=\varnothing$.

在任意时刻,P 的运行系统可以任意选择一个线程执行一条指令.若线程 t 被选中,则执行 $PC(t)$处的指令被执行,根据指令类型决定(V,PC,E)执行后的状态.

分支(真)	if (x_k^t) goto i_j; $V(x_k^t)\neq 0$	$(V,PC[t\mapsto j],E)$**
分支(假)	if (x_k^t) goto i_j; $V(x_k^t)=0$	$(V,PC[t\mapsto PC(t)+1],E)$
读共享内存	$x_k^t=R(x_{k'})$	$(V[x_k^t\mapsto V(x_{k'})],PC[t\mapsto PC(t)+1],E\cup\{R(t,x_{k'},V(x_{k'}))\})$
写共享内存	$W(x_k,x_{k'}^t)$	$(V[x_k\mapsto V(x_{k'}^t)],PC[t\mapsto PC(t)+1],E\cup\{W(t,x_k,V(x_{k'}^t))\})$
本地计算	$x_k^t=op(x_k^t,x_{k'}^t)$	$(V[x_k^t\mapsto op(x_k^t,x_{k'}^t)],PC[t\mapsto PC(t)+1],E)$

程序最后一条指令执行结束后的 E,即为本次执行的访存事件集合.在此模型中,每一线程都按程序依次执行指令,故存$\rightarrow_{po}$(程序顺序,program order)为 E 上的最小的偏序集满足对于任意线程 t,$\{e_i|t_i=t\}$在$\rightarrow_{po}$中被排序.此外,该执行模型假设内存模型满足顺序一致性[22],即,$e_i=R(t_i,x_i,v_i)$的读值 v_i 等于最近一次对 x_i 写操作时写入的值.一些访存依赖获取算法在修正了读/写操作的语义后,可扩展到松弛内存模型上[20,23],我们将在总结具体技术时予以讨论.

1.2 访存依赖获取:问题定义

定义访存事件 e_i,e_j 冲突当且仅当 $t_i\neq t_j\wedge x_i=x_j$ 且至少一个是 W 操作[24].访存依赖定义为访存事件的因果序对

** $M[a\mapsto b]$表示将映射 M 中的 a 映射值替换为 b.

的集合$\rightarrow_{smd}\subseteq E\times E$($e_i\rightarrow_{smd}e_j$表示 e_i 先于 e_j 发生)且其约束了 E 中所有冲突访存事件顺序:令$\rightarrow=tr(\rightarrow_{smd}\cup\rightarrow_{po})$为线程执行顺序$\rightarrow_{po}$和访存依赖$\rightarrow_{smd}$并集的传递闭包,满足任意冲突的两个事件 e_1 和 e_2,有 $e_1\rightarrow e_2\vee e_2\rightarrow e_1$.

根据定义,$\rightarrow_{smd}$消除了访存顺序产生的不确定性.按照$\rightarrow$的任一线性序执行 E 中的事件,将得到等价的执行(每个线程保证执行相同的路径且向系统外输出相同的值).根据线程执行模型,指令按顺序执行,因而存在一个完全反映真实执行的最小偏序$\rightarrow_d$,且所有冲突的访存事件在$\rightarrow_d$中排序.然而,由于现实中的并发程序通常会产生海量的共享内存访问,因此记录$\rightarrow_d$的开销通常是难以容忍的.根据应用场景不同,$\rightarrow_{smd}$不必与$\rightarrow_d$完全相同,甚至一些应用场景可以容忍$\rightarrow_{smd}$与$\rightarrow_d$之间存在冲突.图 1 展示了访存依赖及其不唯一性(变量 x,y 初值为 0,虚线为$\rightarrow_{po}$,实线为$\rightarrow_{smd}$).访存依赖获取定义为在程序运行中通过修改运行时系统(定制硬件、运行时系统或程序插装)的方式记录信息并以直接或间接的方式获得$\rightarrow_{smd}$的过程.访存依赖获取是并发程序动态分析技术的基石:在一定意义上符合$\rightarrow_d$的$\rightarrow_{smd}$可实现执行重放[25];在运行时检查$\rightarrow_{smd}$中的事件能检测或预测数据竞争[26]等运行时错误;在访存发生前获得$\rightarrow_{smd}$,并即时对线程执行顺序予以调控,则能实现事务内存[8]或确定多线程[12]等并发控制手段.尽管这些具体技术的研究领域各不相同,但其核心挑战均是如何高效、准确地获取访存依赖,从而实施轨迹分析或并发控制.本文首创性地将这些跨领域的研究工作通过访存依赖这一基本问题而关联起来,力图更好地理解它们的异同以及指导未来的研究方向.

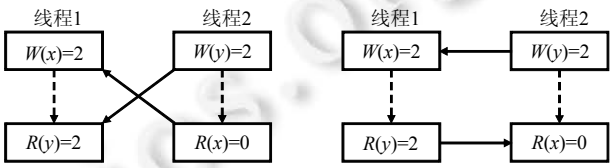

Fig.1 Shared memory dependence logs corresponding to equivalent execution traces

图 1 两组对应相同程序执行结果的访存依赖示例

2 综述框架

本文的框架基于访存依赖的 3 个要素——评价指标、获取技术和应用,具体而言包含 4 个评价指标:即时性、准确性、高效性和简化性;两类获取技术:追踪和合成;两类应用:轨迹分析和并发控制,其框架结构如图 2 所示.第 2.1 节对 4 个评价指标予以概述,第 2.2 节对两类获取技术予以概述,第 2.3 节对两类应用予以概述.后续章节将会展开对获取技术和应用的详细讨论.

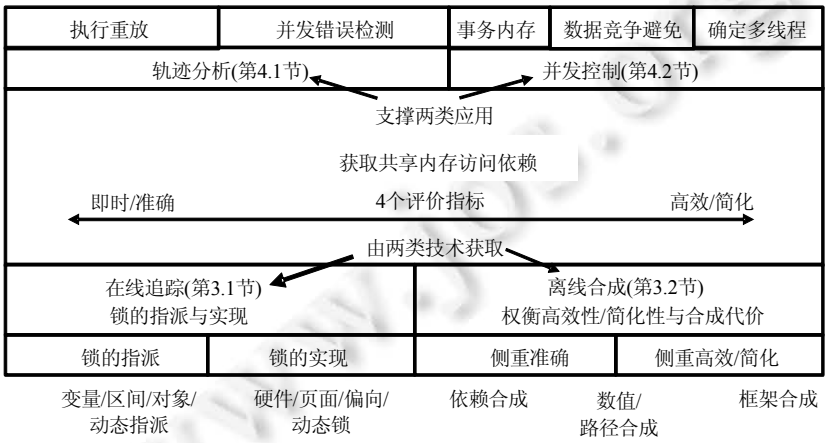

Fig.2 The survey framework

图 2 访存依赖综述框架

2.1 获取访存依赖的评价指标

即时性反映是否在访存事件 e_i 发生前即获得与之相关的访存依赖.即时性对实施运行时并发控制(如事务内存[8])是至关重要的,而轨迹分析类的应用(如测试、调试[6])则通常对即时性没有要求.

准确性反映获取的访存依赖是否正确地反映了执行中的实际访存顺序.获取$\rightarrow_d$的代价通常是较高的,但在有些应用(如基于因果关系的数据竞争检测[27])中又是不可避免的;对于调试、执行重放类的应用,只要求按$\rightarrow_{smd}$执行程序后每一线程的执行路径与$\rightarrow_d$相同(从而足以诊断轨迹中的错误).这一要求的一个充分条件是$\rightarrow_d \subseteq \rightarrow$[28],其不仅能保证执行路径相同,还能保证线程内部状态和输出相同;对性能极为敏感的应用[29],甚至允许$\rightarrow_{smd}$仅包含部分信息或与$\rightarrow_d$不一致.

高效性反映获取访存依赖时程序插装或软/硬件系统实施的修改对程序运行的性能降低的程度.在生产环境中,高效性至关重要,直接决定了一项技术能否在真实环境中部署.即便测试、调试环境能一定程度地容忍运行时开销,随着软件系统的增长、执行轨迹的增大,运行时开销也成为若干动态分析技术的瓶颈[30].

简化性反映访存依赖获取过程中所做的直接或间接记录的数量,而记录的数量直接影响到记录使用者的性能.若应用无需完全准确的访存依赖,$\rightarrow_d$和满足应用需求的最简记录数量可差数个数量级[31].研究者已对运行时[32]或运行后[33]实现访存依赖的精简做出了尝试,同时也证明了获取最简的访存依赖是NP-完全问题[33].

但是,这些评价指标难以同时满足:即时、准确的访存依赖追踪意味着需要运行时开销和更多的依赖数量.并发程序的动态分析技术通常需要根据应用场景的特点对这些评价指标加以权衡.理解这一权衡发生的场景和意义,即是本文的主要技术贡献.

2.2 获取访存依赖的技术

获取访存依赖的方法是对程序进程插装(编译时插装[28]或二进制改写[34])或者对运行系统进行修改(虚拟机[35]或定制硬件[36]),在程序执行时获得动态信息.根据是否在访存事件发生前即得到其访存依赖,其获取技术主要分为直接获取(追踪)和间接获取(合成)两类(如图 3 所示).概言之,在线追踪技术能即时得到准确的访存依赖,但通常对运行系统需要做出较大修改,高效性和简化性不足;离线合成技术由于间接记录的灵活性,能够在高效、简化和合成代价之间进行权衡,缺点是无法即时得到访存依赖.应用应根据其场景(对评价指标的需求)选择合适的访存依赖获取技术.

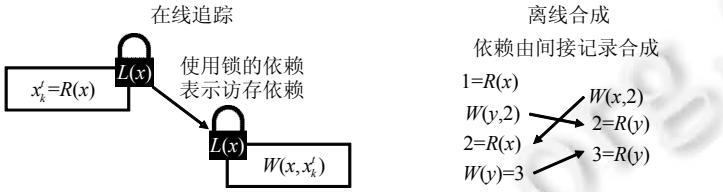

Fig.3 Two approaches to obtaining shared memory dependences: Online tracing and offline synthesis

图 3 访存依赖获取的两类技术:在线追踪、离线合成

2.2.1 直接获取:追踪访存依赖

访存依赖追踪技术直接在内存访问发生前即得到$\rightarrow_{smd}$.目前,这类工作均是通过对共享内存访问予以同步或互斥实现的:为共享变量 x 分配锁 $L(x)$,并用 $L(x)$保护 x 的访问.此时记录解锁和上锁之间的依赖关系,即能获得访存依赖.这类访存依赖追踪技术实现简单、准确性容易保证,是最早被研究的一类技术[6],且已得到广泛的应用.

然而,在程序运行时增加额外锁是与共享内存并发程序设计初衷背道而驰的.共享内存被定义为轻量级的通信方式,线程访问共享内存无需任何同步操作,从而最大程度地发挥了线程之间的并行性.另一方面,为了保证访存依赖的正确获取,依赖信息更新和访存需为原子操作,而对于实现互斥来说,锁或与之等价的同步手段是无法避免的[37].

为实现高效性和简化性,依赖追踪技术的两个关键问题是如何将变量指派到其对应的锁以及如何实现锁机制,从而降低其在运行时开销.锁的指派分为静态指派(在程序运行前确定指派方式,如按照硬件缓存线指派[38]、按对象指派[39]等)和动态指派(在运行时动态根据程序运行时的访存情况进行指派[21]).锁的实现可以借助定制硬件[38]、存储保护[40],也可使用软件实现,其中具有代表性的技术是利用访存局部性实现的偏向锁[20,35].

此外,静态线程逃逸分析[41]、数据竞争分析[42]等技术可以过滤不共享或无数据竞争的变量,从而降低上锁的开销;获取的依赖可以通过离线方式实现简化[31,33].由于此类技术与访存依赖获取完全正交,本文不再赘述.

2.2.2 间接获取:合成访存依赖

除直接追踪访存依赖以外,另一获取方法是在运行时记录间接信息,在程序结束(或运行到检查点)后,根据间接记录合成访存依赖.间接获取法适用于事后分析型的应用,如调试[43]和轨迹预测分析[44].由于无需在运行时立即得到依赖,依赖合成允许消耗更多计算资源甚至使用约束求解器[45],因此,间接记录也更加灵活,通常更加高效和简化.

因此,合成访存依赖的核心问题是:记录何种间接信息,以权衡准确性、高效性、简化性和合成的时间开销.如只记录少量框架性的信息,则合成$\rightarrow_{smd}$的代价极大或无法提供准确性保障,但因为其开销小,适用于生产应用场景[43,46].若在每个线程本地记录读写事件数值或执行路径(此记录无需额外线程间通信),则可以使用约束求解器合成与$\rightarrow_d$在每个线程执行效果等价的$\rightarrow_{smd}$,但依赖合成是 NP-完全问题.若允许进一步增加记录信息的数量(同时记录的高效性和简化性降低),则可在多项式时间推导出$\rightarrow_{smd}$并提供重现等价执行的准确性保证[47].

2.3 访存依赖的应用

基于执行重放的调试技术需要获取或合成访存依赖,从而得到与$\rightarrow_d$等价的执行轨迹.这一执行轨迹可以进一步用于错误预测等动态分析技术.我们统称这类应用为轨迹分析技术.相比开发者使用轨迹分析技术用于测试或调试并发程序,另一类并发控制技术则在生产环境中运行时,根据访存依赖调整程序的行为,以达到消除不确定性和并发错误避免的效果.

2.3.1 轨迹分析

轨迹分析技术在程序运行时或运行后对程序执行轨迹中的各类特征进行分析,从而实现测试、调试等功能,是目前并发程序质量保障的主流技术和研究热点[18].由于共享内存是并发程序不确定性的主要来源,获取访存依赖也是这类技术的核心组成部分.

为实现并发程序的执行重放调试,首先需要保证能够重现并发程序的执行结果,即,按照$\rightarrow_d$或与之等价的访存依赖顺序执行程序[6].在执行重放的基础上,我们还可以从执行轨迹中推断可能存在的并发错误(轨迹预测分析).例如,未被同步操作约束的两个冲突访存事件即构成一个数据竞争[26,48],使用准确的$\rightarrow_{smd}$即可在运行时或运行后预测此类错误.基于同一执行轨迹,研究者还提出了其他定义两个操作是否存在因果顺序约束的因果性模型[49,50],并将因果性模型实现为预测能力更强的并发程序错误检测技术[44].此外,执行轨迹中的访存依赖信息还能指导如原子性违反[4]等其他类型的错误发现,或为后续主动测试提供目标[5].

2.3.2 并发控制

并发控制技术改变程序运行时的访存顺序,从而达到事务处理、错误避免、消除不确定性等功效.并发控制实现的关键技术是在访存发生前即获得依赖信息(此时必须使用在线访存依赖追踪技术),从而对访存顺序进行调控(如检测到冲突事务时进行回滚).由于并发控制技术往往运行在生产环境中,要求在线获取准确的访存依赖,因此是最具挑战性的一类技术.

并发控制为程序员提供系统级的支持,以提高并发软件的质量、减少并发错误.软/硬件事务内存相关研究近年来已取得一定的进展[51,52],Intel 已开始尝试在商业处理器中集成事务支持,并在一些应用场景取得了显著的性能提升[53].为了进一步避免并发错误,运行时系统还可以主动地对程序行为在运行时予以修正.例如,Zhang等人将基本块构成的极大无环子图作为事务处理[54],从而避免有害数据竞争导致的并发错误.减少并发错误的终极目标是确定多线程[12,13]:采取确定的规则对系统内的共享资源进行调度,从而尽可能地降低不确定性带来的并发错误的可能.

3 获取访存依赖的技术

3.1 追踪访存依赖

为在运行时即时、准确地获得访存依赖,最直接的手段是保证共享内存访问的互斥性.若为所有共享变量 x 分配其对应的锁 $L(x)$,并用 $lock(L(x))$和 $unlock(L(x))$包裹所有对 x 的 R/W 操作.记录每个锁上锁、解锁之间的依赖关系所得的$\rightarrow_{smd}$ 保证能推导出$\rightarrow_{d}$ 中的所有依赖.注意,此处的锁不仅仅局限于互斥锁.广义来说,任何实现同步和互斥的机制都能作为锁实现在访存依赖中.追踪访存依赖的关键问题是锁的指派(即,如何进行变量到锁的映射)以及锁的实现(即如何实现互斥),如图 4 所示.锁的指派和实现决定了访存依赖获取技术的准确性、高效性和简化性.

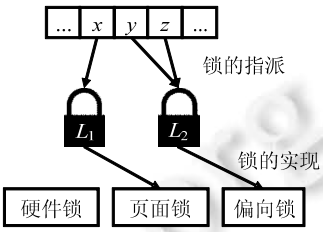

Fig.4 Key challenges of tracing shared memory dependences: Lock assignment and implementation

图 4 访存依赖追踪技术中的关键问题:指派变量对应的锁及高效地实现锁机制

3.1.1 锁的指派

锁的指派指定共享内存变量 x 对应的锁 $L(x)$.为每个变量分配其单独的锁(变量指派),能够获得准确的$\rightarrow_{d}$.然而,变量指派的开销通常是较大的.若在硬件层面实现,则将大幅增加电路实现的复杂性;若使用软件实现,则不仅增加了大量的锁操作,还可能降低缓存的使用率,从而大幅度降低了高效性.解决此问题的途径是进行锁的指派,即,将多个变量指派到同一个锁.锁指派不仅能减少锁的数量从而能够更好地利用现有硬件和访存局部性,还能将同时具有线程和空间局部性的访存事件作为一个单元事件,起到减少依赖数量的功效[21].

静态指派的一类代表方式是,将连续访问的变量指派为同一锁.在硬件实现的锁中,最常见的指派方式是将同一硬件缓存线指派为同一锁(缓存指派)[38,55−59].在此基础上,可以借助(或扩展)缓存一致性协议实现同步和依赖检测,且依赖信息可以在缓存一致性协议中捎带.软件实现的锁从实现简便的角度可在静态时将连续的内存地址指派为同一锁(区间指派,Dunlap 等人提出按虚拟内存页进行指派[40];Jiang 等人提出按缓存线进行指派[20]),或根据静态时的访问名称进行指派[28,47],但也存在其他指派方式.在面向对象程序中,同一对象内变量的并发访问模式通常是类似的.文献[39]最早提出了按对象粒度进行依赖追踪,即,将同一对象内的内存指派为同一锁.对象指派在面向对象程序设计语言中被广泛采纳[35,60].

锁也可以在运行时刻动态指派.Ceze 等人基于硬件的实现[61,62],将连续的内存访问打包,使用 Bloom filter 确定其读/写变量的集合,并借助电路实现的 Bloom filter 操作进行锁的互斥性检测.在软件实现中,由于数据竞争检测问题需要得到准确的$\rightarrow_{d}$,从而要求必须使用变量指派[27,63],带来较大的运行时开销.在此领域中,诞生了一些动态指派的工作以提高数据竞争检测的高效性.若使用对象指派检测对象级的数据竞争,具有高效性但存在误报风险[39],因此,Yu 等人提出使用对象指派检测潜在数据竞争,并据此把有潜在数据竞争的对象切换为变量指派,从而实现高效性和准确性之间的权衡[29].观察到准确依赖追踪的开销主要来自数组访问,Wilcox 等人提出对数组访问的模式进行提取,实现将若干常见的数组访问模式中的锁进行合并,从而提高数据竞争检测的效率[30].在这些工作的基础上,结合程序的线程和空间局部性,Jiang 等人提出了更具一般性的动态锁指派方法,维护地址空间的区间划分(指派同一区间中的变量为同一锁),使得区间内的变量同时具有线程和空间两种局部性,并在获取访存依赖后对划分实施动态调整[21].

锁指派的另一个研究方向是用尽可能少的锁保护若干条而非一条指令.由于必须在指令执行前即持有正

确的锁,因此这类工作是由静态分析实现的.Cherem 等人对这类指派问题以及锁的正确性进行了形式化,归纳总结了静态锁类型、锁组合的正确语义,并用此框架实现了基于抽象对象锁和读-写锁组合的高效事务支持[64].Lee 等人进一步结合了静态和动态数据竞争分析的信息,针对不同抽象变量的访问特点,使用函数锁、基本块锁或循环锁,使之能够高效地获取访存依赖[65].

3.1.2 锁的实现

缓存一致性协议为硬件访存依赖技术提供了自然的上锁机制,因此,该类技术上锁带来的时间开销通常不是其可用性的关键问题[16].但是,由于系统缓存一致性事件频繁发生,短时间内即产生大量依赖,因此,研究工作的核心关注于如何在不大幅改变系统实现的前提下减少依赖的数量.Xu 等人提出,在不违背$\rightarrow_d$的前提下对记录的偏序进行调整,继而删除可被其他依赖推导出的冗余依赖[38,66].Ceze 等人提出记录指令块依赖以实现记录精简[61,62].Chen 等人提出:在上锁的同时增加和记录额外的全局时钟信息,能快速筛除在全局时钟意义下已排序的事件,从而实现大幅度的记录简化[67].

锁还可以借助存储访问控制机制实现.Dunlap 等人提出:使用分页保护机制实现写者在内存页级的互斥,从而在有访存依赖发生时触发缺页异常进而实施记录[40].这类似于按逻辑页进行变量指派,并使用读-写锁予以保护.这一技术也应用在并发控制技术中[68−70].

在不借助定制硬件的前提下,追踪访存依赖需要插装程序或修改运行系统,使用锁保持访存和依赖获取的原子性.追踪访存依赖需保证访存事件和其记录发生在互斥的临界区(通常只有少量机器指令),因而互斥的开销是影响高效性的首要因素,而高效性又直接关系到访存依赖获取技术的实用性.Patil 等人提出直接将基于硬件的访存依赖获取技术[38]使用二进制改写在软件层次实现[71],但在很多应用中都表现出具有巨大的运行时开销.

偏向锁技术[72]假设不同线程访问同一锁的频率不同,且“偏向”频繁持有锁的线程以提高其性能.在访存依赖追踪技术中,类似的偏向设计也成为决定其是否高效的重要因素.例如,Chen 等人提出偏向读操作,在写共享内存时使用互斥锁并在临界区中用原子操作更新版本号,而在读操作时使用读版本号的方式降低开销[73].Bond 等人利用线程局部性设计乐观锁[35],其基本假设是同一变量的连续访问通常发生在同一线程,因此在线程获得对象锁完成访存后,乐观地认为下次访问仍然发生在同一线程,从而不立即释放该锁,而是在虚拟机安全点检查到有等待该锁的线程时加以释放.这一技术可大幅度提高线程局部性强之应用访存依赖追踪的高效性,但在依赖产生频繁的应用中会带来较大的性能问题,因此,该技术的扩展[74]实现了在乐观锁和朴素锁之间的自适应切换(根据访问模式切换锁的实现).Jiang 等人提出了同时偏向读操作和线程本地操作的锁实现,在写时使用朴素锁,读时则采用投机读取和读后验证版本号的方式,在验证到非线程本地读操作时回退到朴素锁[20].

由于锁机制导致额外的同步操作,通常依赖追踪算法都会改变程序的内存模型.大部分工作在对运行系统修改后保证了程序执行满足顺序一致性[28,35,61,62].另一方面,一些技术则刻意放松了内存模型(以软件模拟的局部写缓存为代表[10])以实现高效的依赖获取.基于硬件定制的技术能够捕获底硬件底层乱序执行、写缓存等行为,从而记录松弛内存模型的行为[58,59,66,67,75].基于乐观锁的技术[20]在线程局部读时将不引入任何同步操作,并且能够支持在 x86-TSO 上的正确依赖记录(记录的$\rightarrow_{smd}$的任一线性排序均不满足顺序一致性,但每个读事件对应的写者被正确记录).

3.2 合成访存依赖

相比追踪技术在运行时即得到访存依赖,合成技术在运行时实施间接记录,在执行结束后对记录进行分析,从而恢复执行中的访存依赖.这类技术放弃了访存依赖追踪的即时性,因而记录通常更高效、更简化.同时也意味着,合成访存依赖的时间复杂度较高且记录的信息可能不足以合成准确的依赖.原则上说,只要有极少量的信息(如程序崩溃的现场信息[43]),就能通过搜索的方式合成出用于重新并发错误的执行.记录更多信息会降低合成算法的时间复杂性,同时也降低记录的高效性和简化性.因此,访存依赖合成技术的关键问题是记录何种信息以达到记录的高效性/简化性与合成的准确性/高效性的权衡(如图 5 所示).

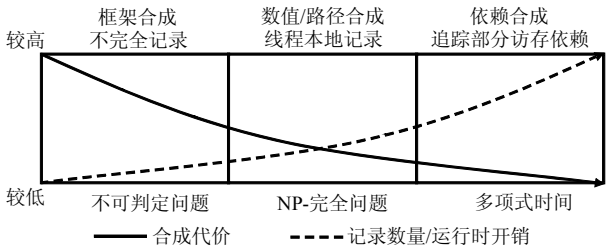

Fig.5 Key challenges of synthesizing shared memory dependences: The trade-off between efficiency, reduction, and cost

图 5 访存依赖合成技术中的关键问题:权衡高效性/简化性与合成代价

3.2.1 框架合成

框架合成工作放弃准确性,只记录运行时的框架信息,从而实现极大的记录高效性和简化性,但合成的准确性和高效性则较低.Zamfir 等人提出:在仅有程序崩溃现场(核心转储)的情况下对程序的路径、调度予以启发式搜索,从而生成一个满足崩溃现场的执行[43].由于在程序执行过程中完全没有实施记录,恢复$\rightarrow_d$是不可能的,但搜索算法在理论上保证能返回一个与$\rightarrow_d$执行产生同样错误的执行轨迹(如复现原执行中最后一条引发错误的指令),从而达到调试的目的.通过记录对错误重现有重大意义的信息(如 Park 等人提出的记录程序中的同步顺序[46],Altekar 等人提出的记录程序的输出[76])并约束生成的访存依赖满足所记录的信息,用这些信息制导搜索,则能有效地缩小搜索空间,降低依赖合成的代价.

3.2.2 数值/路径合成

数值/路径合成的思想是:在线程本地实施记录,然后根据本地记录生成全局合法的$\rightarrow_{smd}$.虽然$\rightarrow_{smd}$可能不完全准确地反映$\rightarrow_d$,但合成得到的$\rightarrow_{smd}$则能够保证每个线程执行与$\rightarrow_d$相同的路径,从而保证并发程序行为(输出、错误)的重现.数值合成的优势是:记录发生在线程本地,而无需锁、原子操作等同步操作.

数值合成在线程本地记录每一访存事件 e_i 对应的值 v_i.Netzer 等人证明:若所有写操作的 v_i 均不同,则存在一个 $O(n\log n)$的算法推导出$\rightarrow_d$(n 为总事件数量)[77];但若允许写入相同值,则满足条件的$\rightarrow_{smd}$不唯一,且合成的时间复杂性是 NP-完全的[78].

为了应对合成大型记录的复杂性,Lee 等人借助硬件实现的全局同步点,将依赖合成问题限制在一段执行窗口中,从而缩小问题的规模[79,80].在窗口内为每个访存事件 e_i 建立变量 O_i 代表其序号,则 $e_i\rightarrow_{po}e_j$ 的约束可用 $O_i<O_j$ 表示,访存事件 e_i 和 e_j 的因果关系可用$(v_i=v_j)\wedge(O_i<O_j)\wedge\forall k(x_k\neq x_i\vee O_k<O_i\vee O_k>O_j)$表示,最后使用 SMT 求解器求解使所有读事件均有其对应写者的约束可满足性问题,即可合成$\rightarrow_{smd}$.

除了考虑执行轨迹中的访存操作外,Huang 等人提出:在线程执行到 if (x_k^l) goto i_j 指令时,记录条件判断的真/假值,从而得到高度可压缩的记录[23].这一记录可以用于唯一确定每一线程的执行路径(从而推导出每一线程的访存事件).通过在文献[79]提出的约束可满足性问题的基础上增加符号化的变量数值和路径条件约束,求解约束问题即可合成访存依赖.这类合成技术需要求解复杂的约束可满足性问题.通过记录一定的额外信息,则能起到简化约束的效果.例如,Yuan 等人提出:在访存时记录处理器的本地时钟,并根据这一信息推断出具有物理时间先后关系的访存事件,从而删除冗余的约束,提高约束求解的效率[81].

由于数值/路径合成的记录完全存放于线程本地,其对运行系统的修改通常对内存模型没有影响.因而在对访存语义进行适当建模后,可实现松弛内存模型意义上的重放.例如,Lee 等人扩展了文献[80]的建模方式以适用于 TSO 内存模型[82];文献[23]提出的技术能够合成在 PSO 内存模型下的调度.

3.2.3 依赖合成

依赖合成意味着我们可以在运行时记录访存依赖的一个子集,再通过合成的方式求解未记录的依赖.例如,按照两个访存事件的先后关系,访存依赖的类型可分为 R-W,W-W 和 W-R 依赖.在偏向锁的设计中,W-R 和 W-W 依赖通常可以用较小的开销得到,而记录 R-W 依赖的代价则可能超过其他二者之和[20,35].在依赖合成中,W-R 和

W-W 依赖可以推导出读-写依赖[19],因此,Zhou 等人提出了使用 W-W 依赖和读值合成 W-R,R-W 依赖的技术,能够在多项式时间内实现$\rightarrow_{smd}$的合成[47].Jiang 等人使用了 W-W 依赖和部分 R-W 依赖实现合成[19].Liu 等人进一步证明:即使在运行时只记录 W-R 依赖,也可保证合成$\rightarrow_{smd}$复现每个线程的行为.其核心思想是:丢弃那些未产生依赖的写操作[83],然后使用类似于文献[79]的技术构造约束进行求解.

4 访存依赖的应用

4.1 轨迹分析

访存依赖反映了程序运行时访问共享内存的顺序,亦包涵了程序执行中不确定性事件的因果关系.轨迹分析技术即从执行轨迹中提取信息,实现并发程序的测试、调试等质量保障方面的技术.一般而言,轨迹分析技术需要准确的访存依赖,以正确地恢复执行轨迹、检出并发错误.

4.1.1 执行重放

轨迹分析最基础的应用是重现过往并发程序的执行.通常,在开发、测试环境下实施依赖获取可复现执行并予以单步调试:要求按照访存依赖顺序$\rightarrow_{smd}$再次执行同一程序,在外部输入相同(通常可以实施记录)的假设下,能够保证重现$\rightarrow_{d}$所对应的每一线程的执行路径和输出,从而保证能够观测到与之前执行相同的结果.这类应用对准确性的要求较高,而对高效性和简化性只要满足应用需求即可.这类场景通常使用依赖追踪的技术或依赖合成技术,以保证重现执行效果[28,40,47].最后,由于对运行系统的修改不可避免地会改变程序的语义(如增加额外的同步操作),从而可以导致部分错误在获取依赖的过程中被隐藏.这一问题的终极解决方案是,对硬件加以对程序语义无任何干扰的修改(如借助缓存一致性协议[38]).目前,Honarmand 等人已经提出了硬件级松弛内存模型实现下执行重放的基础实现模型[75].

执行重放的另一应用场景是生产环境中大型系统错误的复现.在这一场景中,获取准确的访存依赖带来的开销通常是难以接受的,此时,基于合成技术[43,46,76]则可以通过较少的运行时开销帮助开发者复现生产环境中的并发错误.

4.1.2 轨迹预测分析

有时,记录的执行轨迹可能没有产生错误,但若对其稍作修改,则可能暴露其中的并发错误.轨迹预测分析即为这类技术,分析由访存依赖确定的执行轨迹,从而预测潜在的并发错误.

数据竞争检测是访存依赖的一类直接应用.定义为$e_i \rightarrow_{smd} e_j$且$e_i$和$e_j$之间没有发生任何同步操作是多种并发错误的诱因[2],且程序员应当尽力避免.比较两个事件在同步意义下的因果关系,即可实现数据竞争预测分析.由于数据竞争检测需要完全准确的访存依赖($\rightarrow_{d}$),因此其记录难以实现高效和简化.长久以来,这类预测分析都带来较大的运行开销.Flanagan 等人提出了一种时钟维护方法,大大降低了其开销[27];Wilcox 等人在此基础上提高了数组访问的高效性[30];Yu 等人则使用动态锁指派,在检测到对象级数据竞争后才切换到变量指派,检测准确的数据竞争,从而提高了高效性[29].这些技术的发展也代表了访存依赖获取技术的进步.

上述的数据竞争定义在 happens-before[84]模型上,但这一模型只是e_i和e_j之间实际因果性的一个近似[50].基于因果性的定义,研究者提出了在没有程序语义情况下仅观察访存事件所能推断出的最大因果模型[49]及基于此的预测分析算法[44].Huang 等人则提出,将一般性的错误模式(资源竞争、原子性违反、并发迭代器、空指针引用等)编码为轨迹上的性质,从而使用约束求解器找出轨迹中的并发错误[85].

这些基于访存依赖的技术不仅能够检测数据竞争,还能检测原子性违反[86]、假共享[87]等并发错误,在并发程序质量保障方面显现出卓著的成效,但其具体技术超过了本文的范围,这里不再赘述.

4.1.3 轨迹的其他应用

在一次执行轨迹分析的基础上,轨迹分析的结果也能为更复杂的并发程序的测试和分析技术提供指导.Sen 等人分析轨迹中可能的数据竞争点,再使用 fuzz testing 技术予以确认[5];无状态模型检验技术[88,89]通过反转执行中的依赖实现状态空间的遍历;Cai 等人提出,通过启发式地反转多个依赖实现高效的数据竞争检测[90];Lee 等人提出,利用轨迹信息决定静态变量应当指派的锁[65].

4.2 并发控制

除测试、调试类的支持外,访存依赖还可用于在运行时调整并发程序行为,从而实现并发错误的预防和避免.此类技术可由程序员指定(事务内存),也可直接实现在运行时系统(数据竞争避免、确定多线程).由于并发控制技术通常用于生产系统,其主要挑战之一即是高效地在运行时获取准确的访存依赖.

4.2.1 事务内存

事务内存允许程序中使用 atomic 关键字包裹一段程序区域,并由运行时系统保证原子区域的可序列化[52].事务内存既可在硬件实现,也可使用软件实现,而其实现的两个关键问题是冲突检测和冲突化解.事务冲突定义为并发事务中存在两个访存事件访问同一地址且至少有一个是写,因此,冲突检测等价于运行时访存依赖的追踪.本综述中,访存依赖追踪技术的两个关键技术也可用于理解事务内存技术.Dice 等人总结了事务内存的设计空间[9],其中包含锁的指派(文献中讨论了静态的对象指派、缓存线指派、区间指派等)和锁的设计(主要讨论了基于版本的锁)等关键技术,这和访存依赖追踪技术十分类似.Dice 等人还提出使用变量指派和用位压缩存储字节数组实现的偏向锁,大幅度地提高了读操作的性能[91];Dalessandro 等人提出,将所有变量指派为同一锁,从而能够实现最简的冲突检测和解决[92];Zhang 等人提出基于偏向锁[35]实现事务内存,使用了面向对象语言中常用的对象指派和偏向锁实现冲突检测[54].

4.2.2 数据竞争避免

数据竞争带来并发错误的主要原因是竞争的写操作破坏了其他线程的原子性,因此,避免数据竞争导致并发错误的一类方法是修改运行时系统的语义,将一个连续区域中的指令作为事务处理保证其原子性,从而降低了并发错误发生的可能性.为避免数据竞争及所带来的其他后果,Berger 等人提出使用分页保护技术实现页面指派[68],建立了松弛内存模型系统,使竞争写操作发生在线程本地,从而不会破坏其他线程的原子性.Sengupta 等人提出通过扩展事务内存技术,在静态时对程序进行插装,将控制流图中不包含同步操作、方法调用和回边的极大区域包裹为事务,从而避免数据竞争情况的发生[11].

另一种避免数据竞争有害后果的方式是修改运行时系统,在数据竞争发生时抛出异常.Lucia 等人提出:通过定制硬件修改内存模型隔离数据竞争,并在同步操作时予以竞争检测,在数据竞争发生时陷入异常[57].这类数据竞争检测技术需要准确的访存依赖,因此,纯软件实现的开销对于生产系统来说是难以接受的.鉴于数据竞争检测的开销主要来自 R-W 依赖的追踪,Biswas 等人提出在线程本地维护读事件的列表,将这类竞争的检测延迟到同步区域结束,从而实现高效的区域冲突检测(仅能检出区域间的 R-W 数据竞争)[93].

4.2.3 确定多线程

鉴于并发错误的主要诱因是执行的不确定性,确定多线程试图从根本上消除不确定性,使并发程序在相同的输入上表现出相同的行为,从而降低并发程序测试及调试的难度,并在运行时避免并发错误的发生.在线追踪访存依赖亦是确定多线程技术的基石.Devietti 等人提出在线程之间以确定性的方式传递令牌,通过缓存一致性协议捕获访存依赖,并在其发生时等待令牌,此即实现了系统的确定化[12].Bergan 等人扩展了文献[68]的技术,将写操作缓存在本地,从而减少了访存依赖的数量,在实现确定多线程的同时缩短了等待令牌的时间[94].

另一类工作[10,69,70]则利用存储保护机制,以页面粒度实施锁的指派,从而实现确定化.

Cui 等人提出:事先记录程序执行轨迹,使程序即使在输入不同的情况下也尽最大可能复用已有轨迹中的调度,从而降低可能导致并发错误调度(如数据竞争和原子性违反)的可能性,避免并发错误的出现[95,96].在此基础上,对同步操作使用令牌传递机制,则可同时实现确定多线程和并发错误避免[13].

5 研究契机

本文提出了访存依赖获取的几个关键问题:依赖追踪中的锁指派、锁分配;依赖合成中的信息使用.通过填补综述框架中的空白,我们已取得了一定的研究进展,设计了基于值-依赖混合的合成方法 CARE[19]、利用偏向锁 RWTrace[20]和基于二分区间的动态锁指派方法 bisectional coordination[21].鉴于这一框架仍有很多空白之处,现将一些有价值的访存依赖获取相关的研究问题予以展望.

5.1.1 已有技术之间的融合

目前,各个领域的研究仍然相对封闭.本综述的目的之一就是对这些领域之间共同的技术予以总结,从而可以促进领域之间技术的融合.已有一些通过借鉴另一领域的技术进行实现的例子:文献[35]是软件实现的乐观MESI 协议;文献[23]提出的路径合成技术借鉴了硬件数值合成技术[79]中的约束可满足性问题构造;Zhang 等人提出了使用硬件事务内存实现数据竞争检测的加速[97].现有一些技术(如文献[62])在硬件上得以高效实现,但如何使用软件实现仍是挑战之一;反之,软件中研究的技术也可能集成到硬件实现中,其中一个例子是将动态锁指派技术[21]实现在硬件层次,将若干连续的缓存线进行绑定,作为原子处理.这一融合对硬件修改的复杂性较低,不仅能够降低访存依赖获取的开销,还可能利用运行时信息(如缓存线之间的空间和线程局部性)实现缓存预取,从而提高多核处理器的执行效率.

5.1.2 锁的动态自适应指派和实现

在软件访存依赖获取技术中,一类新兴的研究是在运行时自适应地调整锁指派,以提高简化性[21];或调整锁实现,以提高高效性[74].此类技术也是未来研究的可行方向.由于程序运行时具有各种局部性,这类新技术可能对访存依赖获取的高效性和简化性带来巨大的提升.然而,这两类研究目前都处于初级阶段:实施动态指派虽然能够提高简化性,但动态指派自身亦带来一定的时间开销,尤其是在动态指派机制复杂性增加后,这类开销更有可能掩盖其带来的益处,如何权衡这些要素,是这类技术未来的研究重点.除了动态锁指派和动态锁实现本身技术的研究以外,其适用的应用领域亦是未来研究的契机之一.特定的应用由于均摊到每一访存事件从而能够接受更高的代价(如软件事务内存[8]),此时,动态锁指派或动态锁实现技术带来的收益(如降低锁的数量和事务回滚的频率)可能远远大于其自身的开销,动态锁技术即可促进应用领域的进步.

推而广之,未来研究还可以实现追踪/合成技术的动态切换.Wu 等人观察到函数有轨迹记录和输出记录两种方式,从而使用网络最小割建模两种记录方式的代价并予以最优选择,从而实现高效记录[98].同理,注意到:路径合成技术[23]在访存事件多、执行路径简单(如生产者-消费者模型)的应用中能大幅提高记录的高效性和简化性;反之,若执行在访存少、路径复杂(如科学计算)的应用中则会记录大量不包含不确定性的分支,其性能甚至不如基于锁的追踪技术.如能在运行时刻根据程序自身特性,自适应地在多个技术之间融合,则能取长补短,实现更高效的访存依赖获取.

5.1.3 记录额外信息以降低合成复杂性

另一个研究方向是如何在运行时用较少的代价得到一些额外信息,从而大幅度降低访存依赖合成的复杂性.现已有一些工作利用了这类信息:Chen 等人利用了时钟信息减少依赖的数量[67];Liu 等人使用了程序中的同步操作来降低约束求解的难度[83].然而,目前还没有一个系统化的方式理解何种信息可以何种方式获取、有怎样的代价、对访存依赖获取有怎样的影响.对这一问题的深入理解,势必将推进访存依赖合成技术实用化的进程.

5.1.4 建模并发程序的执行

目前,一些研究工作[19,83]虽在实际中证实了有效性,但其合成访存依赖的时间复杂性仍未得到证明,我们目前猜想,它们的时间复杂性都是 NP-完全的.另一方面,这类技术(包括一些已经被证明是 NP-完全问题的技术[79])在实践中的表现都是非常优秀的.因此我们有理由相信,现实中并发程序的执行满足一定的特性,从而依赖合成在实际中的复杂性是较低的.框架合成技术[46,76]也隐含地利用了这一假设,且已经证实:在实际应用中,此类技术的开销是可以接受的.

如能准确地建模真实世界中并发程序的执行,则能更好地理解这类技术的优势与不足.基于这一模型,已有技术的优缺点均可在理论模型上予以分析;提出新技术的研究者在进行实例研究的同时,还能从理论上评估其表现(如在何种情况下表现为多项式时间复杂性,或在模型上对比与其他技术的开销).

6 总 结

本文讨论了访存依赖获取技术的核心要素:4 个评价指标(即时、准确、高效、简化)、两大类方法(在线合

成、离线获取)和两大类应用(轨迹分析、并发控制).本文提出的框架成功地提取了来自多个领域的并发程序动态分析技术中的共性问题,并能很好地理解这些技术因应用特征不同而在评价指标之间做出的权衡.此外,当前的框架中出现了一些空白,也为未来工作的开展奠定了基础.希望本文能进一步促进并发程序动态分析领域的研究发展,实现自动化的并发程序质量保障.

References:

[1] Hofstee HP. Future microprocessors and off-chip SOP interconnect. IEEE Trans. on Advanced Packaging, 2004,27(2):301−303. [doi: 10.1109/TADVP.2004.830355]

[2] Lu S, Park S, Seo E, Zhou Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: Eggers SJ, Larus JR, eds. Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2008. 329−339.

[3] Goodstein ML, Vlachos E, Chen S, Gibbons PB, Kozuch MA, Mowry TC. Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2010. 257−270. [doi: 10.1145/1736020.1736050]

[4] Lu S, Tucek J, Qin F, Zhou Y. AVIO: Detecting atomicity violations via access interleaving invariants. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2006. 37−48. [doi: 10.1145/1168857.1168864]

[5] Sen K. Race directed random testing of concurrent programs. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2008. 11−21. [doi: 10.1145/1375581.1375584]

[6] Leblanc TJ, Mellor-Crummey JM. Debugging parallel programs with instant replay. IEEE Trans. on Computers, 1987,C-36(4):471−482. [doi: 10.1109/TC.1987.1676929]

[7] Herlihy M, Moss JE. Transactional memory: Architectural support for lock-free data structures. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 1993. 289−300. [doi: 10.1145/165123.165164]

[8] Shavit N, Touitou D. Software transactional memory. In: Anderson JH, ed. Proc. of the Symp. on Principles of Distributed Computing (PODC). ACM, 1995. 204−213.

[9] Dice D, Shavit N. Understanding tradeoffs in software transactional memory. In: Proc. of the Int'l Symp. on Code Generation and Optimization (CGO). 2007. 21−33. [doi: 10.1109/CGO.2007.38]

[10] Devietti J, Nelson J, Bergan T, Ceze L, Grossman D. RCDC: A relaxed consistency deterministic computer. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2012. 67−78. [doi: 10.1145/1950365.1950376]

[11] Sengupta A, Biswas S, Zhang M, Bond MD, Kulkarni M. Hybrid static-dynamic analysis for statically bounded region serializability. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2015. 561−575. [doi: 10.1145/2694344.2694379]

[12] Devietti J, Lucia B, Ceze L, Oskin M. DMP: Deterministic shared memory multiprocessing. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2009. 85−96. [doi: 10.1145/1508244.1508255]

[13] Cui H, Simsa J, Lin Y, Li H, Blum B, Xu X, Yang J, Gibson GA, Bryant RE. Parrot: A practical runtime for deterministic, stable, and reliable threads. In: Proc. of the Symp. on Operating Systems Principles (SOSP). 2013. 388−405. [doi: 10.1145/2517349.2522735]

[14] Pusukuri KK, Gupta R, Bhuyan AN. Thread tranquilizer: Dynamically reducing performance variation. ACM Trans. on Architecture and Code Optimization, 2012,8(4):46−66. [doi: 10.1145/2086696.2086725]

[15] Gao L, Wang R, Qian DP. Deterministic replay for parallel programs in multi-core processors. Ruan Jian Xue Bao/Journal of Software, 2013,24(6):1390−1402 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4392.htm [doi: 10.3724/SP.J.1001.2013.04392]

[16] Chen Y, Zhang S, Guo Q, Li L, Wu R, Chen T. Deterministic replay: A survey. ACM Computing Surveys (CSUR), 2015,48(2):17−None. [doi: 10.1145/2790077]

[17] Hong S, Kim M. A survey of race bug detection techniques for multithreaded programs. Journal of Software: Testing, Verification and Reliability, 2015,25(3):191−217. [doi: 10.1002/stvr.1564]

[18] Su XH, Yu Z, Wang TT, Ma PJ. A survey on exposing, deteting and avoiding concurrency bugs. Chinese Journal of Computers, 2015,38(11):2215−2233 (in Chinese with English abstract).

[19] Jiang Y, Gu T, Xu C, Ma X, Lu J. CARE: Cache guided deterministic replay for concurrent Java programs. In: Jalote P, Briand LC, van der Hoek A, eds. Proc. of the Int'l Conf. on Software Engineering (ICSE). ACM, 2014. 457−467.

[20] Jiang Y, Li D, Xu C, Ma X, Lu J. Optimistic shared memory dependence tracing. In: Proc. of the Int'l Conf. on Automated Software Enginnering (ASE). 2015. 524−534. [doi: 10.1109/ASE.2015.11]

[21] Jiang Y, Xu C, Li D, Ma X, Lu J. Online shared memory dependence reduction via bisectional coordination. In: Proc. of the Symp. on the Foundations of Software Engineering (FSE). 2016. 822−832. [doi: 10.1145/2950290.2950326]

[22] Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. on Computers, 1979,C-28(9):690−691. [doi: 10.1109/TC.1979.1675439]

[23] Huang J, Zhang C, Dolby J. CLAP: Recording local executions to reproduce concurrency failures. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2013. 141−152. [doi: 10.1145/2491956.2462167]

[24] Adve SV, Boehm H. Memory models: A case for rethinking parallel languages and hardware. Communications of the ACM, 2010, 53(8):90−101. [doi: 10.1145/1787234.1787255]

[25] Ronsse M, De Bosschere K. RecPlay: A fully integrated practical record/replay system. ACM Trans. on Computer Systems, 1999, 17(2):133−152. [doi: 10.1145/312203.312214]

[26] Dinning A, Schonberg E. An empirical comparison of monitoring algorithms for access anomaly detection. In: Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP). 1990. 1−10. [doi: 10.1145/99163.99165]

[27] Flanagan C, Freund SN. FastTrack: Efficient and precise dynamic race detection. In: Hind M, Diwan A, eds. Proc. of the Conf. on Programming Language Design and Implementation (PLDI). ACM, 2009. 121−133.

[28] Huang J, Liu P, Zhang C. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In: Roman G-C, Sullivan KJ, eds. Proc. of the Symp. on the Foundations of Software Engineering (FSE). ACM, 2010. 385−386.

[29] Yu Y, Rodeheffer T, Chen W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In: Proc. of the Symp. on Operating Systems Principles (SOSP). 2005. 221−234. [doi: 10.1145/1095809.1095832]

[30] Wilcox JR, Finch P, Flanagan C, Freund SN. Array shadow state compression for precise dynamic race detection. In: Proc. of the Int'l Conf. on Automated Software Enginnering (ASE). 2015. 155−165. [doi: 10.1109/ASE.2015.19]

[31] Huang J, Zhang C. An efficient static trace simplification technique for debugging concurrent programs. In: Proc. of the Int'l Conf. on Static Analysis (SAS). 2011. 163−179. [doi: 10.1007/978-3-642-23702-7_15]

[32] Netzer RHB. Optimal tracing and replay for debugging shared-memory parallel programs. In: Proc. of the Workshop on Parallel and Distributed Debugging (PADD). 1993. 1−11. [doi: 10.1145/174266.174268]

[33] Jalbert N, Sen K. A trace simplification technique for effective debugging of concurrent programs. In: Proc. of the Symp. on the Foundations of Software Engineering (FSE). 2010. 57−66. [doi: 10.1145/1882291.1882302]

[34] Bhansali S, Chen W, de Jong S, Edwards A, Murray R, Drinić M, Mihočka D, Chau J. Framework for instruction-level tracing and analysis of program executions. In: Proc. of the Int'l Conf. on Virtual Execution Environments (VEE). 2006. 154−163. [doi: 10.1145/1134760.1220164]

[35] Bond MD, Kulkarni M, Cao M, Zhang M, Fathi Salmi M, Biswas S, Sengupta A, Huang J. OCTET: Capturing and controlling cross-thread dependences efficiently. In: Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 2013. 693−712. [doi: 10.1145/2509136.2509519]

[36] Bacon DF, Goldstein SC. Hardware-Assisted replay of multiprocessor programs. In: Proc. of the Workshop on Parallel and Distributed Debugging (PADD). 1991. 194−206. [doi: 10.1145/122759.122777]

[37] Attiya H, Guerraoui R, Hendler D, Kuznetsov P, Michael MM, Vechev M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In: Proc. of the Symp. on Principles of Programming Languages (POPL). 2011. 487−498. [doi: 10.1145/1926385.1926442]

[38] Xu M, Bodik R, Hill MD. A flight data recorder for enabling full-system multiprocessor deterministic replay. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2003. 122−135. [doi: 10.1145/859618.859633]

[39] von Praun C, Gross TR. Object race detection. In: Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 2001. 70−82. [doi: 10.1145/504282.504288]

[40] Dunlap GW, Lucchetti DG, Fetterman MA, Chen PM. Execution replay of multiprocessor virtual machines. In: Proc. of the Int'l Conf. on Virtual Execution Environments (VEE). 2008. 121−130. [doi: 10.1145/1346256.1346273]

[41] Choi J, Gupta M, Serrano M, Sreedhar VC, Midkiff S. Escape analysis for Java. In: Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 1999. 1−19. [doi: 10.1145/320384.320386]

[42] Voung JW, Jhala R, Lerner S. RELAY: Static race detection on millions of lines of code. In: Crnkovic I, Bertolino A, eds. Proc. of the Joint Meeting of the European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2007. 205−214.

[43] Zamfir C, Candea G. Execution synthesis: A technique for automated software debugging. In: Proc. of the European Conf. on Computer Systems (EuroSys). 2010. 321−334. [doi: 10.1145/1755913.1755946]

[44] Huang J, Meredith PO, Rosu G. Maximal sound predictive race detection with control flow abstraction. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2014. 337−348. [doi: 10.1145/2594291.2594315]

[45] Nieuwenhuis R, Oliveras A, Tinelli C. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL. Journal of the ACM, 2006,53(6):937−977. [doi: 10.1145/1217856.1217859]

[46] Park S, Zhou Y, Xiong W, Yin Z, Kaushik R, Lee KH, Lu S. PRES: Probabilistic replay with execution sketching on multiprocessors. In: Proc. of the Symp. on Operating Systems Principles (SOSP). 2009. 177−192. [doi: 10.1145/1629575.1629593]

[47] Zhou J, Xiao X, Zhang C. Stride: Search-Based deterministic replay in polynomial time via bounded linkage. In: Proc. of the Int'l Conf. on Software Engineering (ICSE). 2012. 892−902. [doi: 10.1109/ICSE.2012.6227130]

[48] Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems, 1997,15(4):391−411. [doi: 10.1145/265924.265927]

[49] Serbanuta TF, Chen F, Rosu G. Maximal causal models for sequentially consistent systems. In: Proc. of the Int'l Conf. on Runtime Verification (RV). 2012. 136−150. [doi: 10.1007/978-3-642-35632-2_16]

[50] Smaragdakis Y, Evans J, Sadowski C, Yi J, Flanagan C. Sound predictive race detection in polynomial time. In: Proc. of the Symp. on Principles of Programming Languages (POPL). 2012. 387−400. [doi: 10.1145/2103656.2103702]

[51] Dolby J, Hammer C, Marino D, Tip F, Vaziri M, Vitek J. A data-centric approach to synchronization. ACM Trans. on Programming Languages and Systems, 2012,34(1):4:1−4:48. [doi: 10.1145/2160910.2160913]

[52] Harris T, Cristal A, Unsal OS, Ayguade E, Gagliardi F, Smith B, Valero M. Transactional memory: An overview. IEEE Micro, 2007,27(3):8−29. [doi: 10.1109/MM.2007.63]

[53] Yoo RM, Hughes CJ, Lai K, Rajwar R. Performance evaluation of Intel(R) transactional synchronization extensions for high-performance computing. In: Proc. of the Int'l Conf. for High Performance Computing (SC). 2013. 1−11. [doi: 10.1145/2503210.2503232]

[54] Zhang M, Huang J, Cao M, Bond MD. Low-Overhead software transactional memory with progress guarantees and strong semantics. In: Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP). 2015. 97−108. [doi: 10.1145/2688500.2688510]

[55] Min SL, Choi J. An efficient cache-based access anomaly detection scheme. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 1991. 235−244. [doi: 10.1145/106972.106996]

[56] Ananian CS, Asanovic K, Kuszmaul BC, Leiserson CE, Lie S. Unbounded transactional memory. IEEE Micro, 2006,26(1):59−69. [doi: 10.1109/MM.2006.26]

[57] Lucia B, Ceze L, Strauss K, Qadeer S, Boehm H. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2010. 210−221. [doi: 10.1145/1815961.1815987]

[58] Hower DR, Hill MD. Rerun: Exploiting episodes for lightweight memory race recording. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2008. 265−276. [doi: 10.1109/ISCA.2008.26]

[59] Montesinos P, Ceze L, Torrellas J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2008. 289−300. [doi: 10.1109/ISCA.2008.36]

[60] Yang Z, Yang M, Zang B, Xu L, Chen H. ORDER: Object centric deterministic replay for Java. In: Nieh J, Waldspurger CA, eds. Proc. of the USENIX Annual Technical Conf. (ATC). USENIX Association, 2011. 1−14.

[61] Ceze L, Tuck J, Torrellas J, Cascaval C. Bulk disambiguation of speculative threads in multiprocessors. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2006. 227−238. [doi: 10.1109/ISCA.2006.13]

[62] Ceze L, Tuck J, Montesinos P, Torrellas J. BulkSC: Bulk enforcement of sequential consistency. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2007. 278−289. [doi: 10.1145/1250662.1250697]

[63] Elmas T, Qadeer S, Tasiran S. Goldilocks: A race and transaction-aware Java runtime. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2007. 245−255. [doi: 10.1145/1250734.1250762]

[64] Cherem S, Chilimbi T, Gulwani S. Inferring locks for atomic sections. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2008. 304−315. [doi: 10.1145/1375581.1375619]

[65] Lee D, Chen PM, Flinn J, Narayanasamy S. Chimera: Hybrid program analysis for determinism. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2012. 463−474. [doi: 10.1145/2254064.2254119]

[66] Xu M, Hill MD, Bodik R. A regulated transitive reduction (RTR) for longer memory race recording. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2006. 49−60. [doi: 10.1145/1168857.1168865]

[67] Chen Y, Hu W, Chen T, Wu R. LReplay: A pending period based deterministic replay scheme. In: Proc. of the Int'l Symp. on Computer Architecture (ISCA). 2010. 187−197. [doi: 10.1145/1815961.1815985]

[68] Berger ED, Yang T, Liu T, Novark G. Grace: Safe multithreaded programming for C/C++. In: Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 2009. 81−96. [doi: 10.1145/1640089.1640096]

[69] Liu T, Curtsinger C, Berger ED. Dthreads: Efficient deterministic multithreading. In: Proc. of the Symp. on Operating Systems Principles (SOSP). 2011. 327−336. [doi: 10.1145/2043556.2043587]

[70] Bergan T, Hunt N, Ceze L, Gribble SD. Deterministic process groups in dOS. In: Arpaci-Dusseau RH, Chen B, eds. Proc. of the Conf. on Operating Systems Design and Implementation (OSDI). USENIX Association, 2010. 177−191.

[71] Patil H, Pereira C, Stallcup M, Lueck G, Cownie J. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In: Proc. of the Int'l Symp. on Code Generation and Optimization (CGO). 2010. 2−11. [doi: 10.1145/1772954.1772958]

[72] Russell K, Detlefs D. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In: Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 2006. 263−272. [doi: 10.1145/1167473.1167496]

[73] Chen Y, Chen H. Scalable deterministic replay in a parallel full-system emulator. In: Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP). 2013. 207−218. [doi: 10.1145/2442516.2442537]

[74] Cao M, Zhang M, Sengupta A, Bond MD. Drinking from both glasses: Combining pessimistic and optimistic tracking of cross-thread dependences. In: Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP). 2016. 20:1−20:13. [doi: 10.1145/2851141.2851143]

[75] Honarmand N, Torrellas J. RelaxReplay: Record and replay for relaxed-consistency multiprocessors. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2014. 223−238. [doi: 10.1145/2541940.2541979]

[76] Altekar G, Stoica I. ODR: Output-Deterministic replay for multicore debugging. In: Proc. of the Symp. on Operating Systems Principles (SOSP). 2009. 193−206. [doi: 10.1145/1629575.1629594]

[77] Netzer RH, Miller BP. On the complexity of event ordering for shared-memory parallel program executions. In: Wahhabi BW, ed. Proc. of the Int'l Conf. on Parallel Processing (ICPP). Pennsylvania State University Press, 1990. 93−97.

[78] Gibbons PB, Korach E. Testing shared memories. SIAM Journal on Computing, 1997,26(4):1208−1244. [doi: 10.1137/S0097539794279614]

[79] Lee D, Said M, Narayanasamy S, Yang Z, Pereira C. Offline symbolic analysis for multi-processor execution replay. In: Albonesi DH, Martonosi M, August DI, Martínez JF, eds. Proc. of the Int'l Symp. on Microarchitecture (MICRO). ACM, 2009. 564−575.

[80] Narayanasamy S, Pereira C, Calder B. Recording shared memory dependencies using strata. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2006. 229−240. [doi: 10.1145/1168857.1168886]

[81] Yuan X, Wu C, Wang Z, Li J, Yew P, Huang J, Feng X, Lan Y, Chen Y, Guan Y. ReCBuLC: Reproducing concurrency bugs using local clocks. In: Proc. of the Int'l Conf. on Software Engineering (ICSE). 2015. 824−834. [doi: 10.1109/ICSE.2015.94]

[82] Lee D, Said M, Narayanasamy S, Yang Z. Offline symbolic analysis to infer total store order. In: Proc. of the 2011 Int'l Symp. on High Performance Computer Architecture (HPCA). 2011. 357−358. [doi: 10.1109/HPCA.2011.5749743]

[83] Liu P, Zhang X, Tripp O, Zheng Y. Light: Replay via tightly bounded recording. In: Proc. of the Conf. on Programming Language Design and Implementation (PLDI). 2015. 55−64. [doi: 10.1145/2737924.2738001]

[84] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978,21(7):558−565. [doi: 10.1145/359545.359563]

[85] Huang J, Luo Q, Rosu G. GPredict: Generic predictive concurrency analysis. In: Proc. of the Int'l Conf. on Software Engineering (ICSE). 2015. 847−857. [doi: 10.1109/ICSE.2015.96]

[86] Huang J, Zhang C. Persuasive prediction of concurrency access anomalies. In: Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA). 2011. 144−154. [doi: 10.1145/2001420.2001438]

[87] Liu T, Berger ED. SHERIFF: Precise detection and automatic mitigation of false sharing. In: Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). 2011. 3−18. [doi: 10.1145/2048066.2048070]

[88] Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I. Finding and reproducing Heisenbugs in concurrent programs. In: Draves R, van Renesse R, eds. Proc. of the Conf. on Operating Systems Design and Implementation (OSDI). USENIX Association, 2008. 267−280.

[89] Park S, Lu S, Zhou Y. CTrigger: Exposing atomicity violation bugs from their hiding places. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2009. 25−36. [doi: 10.1145/1508244.1508249]

[90] Cai Y, Cao L. Effective and precise dynamic detection of hidden races for Java programs. In: Proc. of the Joint Meeting of the European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering (ESEC/FSE). 2015. 450−461. [doi: 10.1145/2786805.2786839]

[91] Dice D, Shavit N. TLRW: Return of the read-write lock. In: Proc. of the Symp. on Parallelism in Algorithms and Architectures (SPAA). 2010. 284−293. [doi: 10.1145/1810479.1810531]

[92] Dalessandro L, Spear MF, Scott ML. NOrec: Streamlining STM by abolishing ownership records. In: Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP). 2010. 67−78. [doi: 10.1145/1693453.1693464]

[93] Biswas S, Zhang M, Bond MD, Lucia B. Valor: Efficient, software-only region conflict exceptions. In: Aldrich J, Eugster P, eds. Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA). USENIX Association, 2015. 241−259.

[94] Bergan T, Anderson O, Devietti J, Ceze L, Grossman D. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2010. 53−64. [doi: 10.1145/1736020.1736029]

[95] Cui H, Wu J, Tsai C, Yang J. Stable deterministic multithreading through schedule memoization. In: Arpaci-Dusseau RH, Chen B, eds. Proc. of the Conf. on Operating Systems Design and Implementation (OSDI). USENIX Association, 2010. 207−221.

[96] Cui H, Wu J, Gallagher J, Guo H, Yang J. Efficient deterministic multithreading through schedule relaxation. In: Proc. of the Symp. on Operating Systems Principles (SOSP). 2011. 337−351. [doi: 10.1145/2043556.2043588]

[97] Zhang T, Lee D, Jung C. TxRace: Efficient data race detection using commodity hardware transactional memory. In: Proc. of the Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2016. 159−173. [doi: 10.1145/2872362.2872384]

[98] Wu M, Long F, Wang X, Xu Z, Lin H, Liu X, Guo Z, Guo H, Zhou L, Zhang Z. Language-based replay via data flow cut. In: Proc. of the Symp. on the Foundations of Software Engineering (FSE). 2010. 197−206. [doi: 10.1145/1882291.1882322]

附中文参考文献:

[15] 高岚,王锐,钱德沛.多核处理器并行程序的确定性重放研究.软件学报,2013,24(6):1390−1402. http://www.jos.org.cn/1000-9825/4392.htm [doi: 10.3724/SP.J.1001.2013.04392]

[18] 苏小红,禹振,王甜甜,马培军.并发缺陷暴露、检测与规避研究综述.计算机学报,2015,38(11):2215−2233.

蒋炎岩(1988－),男,江苏南京人,学士,主要研究领域为软件测试与分析.

马晓星(1975－),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为自适应软件系统,软件工程.

许畅(1977－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为大数据软件工程,软件测试与分析,自适应与嵌入式系统.

吕建(1960－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为大数据软件工程,软件测试与分析,自适应与嵌入式系统.