南京大學

研 究 生 毕 业 论 文
(申请博士学位)

论 文 题 目	并发程序共享内存访问依赖研究
作 者 姓 名	蒋炎岩
学科、专业名称	计算机软件与理论
研 究 方 向	并发程序的测试与分析
指 导 教 师	吕建 马晓星 许畅 教授

2017 年 11 月

学　　号：　DG1333011

论文答辩日期：

指 导 教 师：　　(签字)

Understanding Shared Memory Dependences in Concurrent Programs

by

Yanyan Jiang

Supervised by

Professor Jian Lu, Xiaoxing Ma, and Chang Xu

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in
Computer Software and Theory

Department of Computer Science and Technology
Nanjing University

November, 2017

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：并发程序共享内存访问依赖研究

计算机软件与理论 专业 2013 级博士生 姓名：蒋炎岩

指导教师 (姓名、职称)：吕建 马晓星 许畅 教授

摘 要

随着多核处理器的普及和编程语言的发展，并发程序在近年来得到了迅速的普及。由于并发程序中的线程访问共享资源的顺序并不确定，导致并发缺陷难触发、难复现、难检测，给并发程序的质量保障带来了重大挑战。

本文关注实现并发程序动态分析的基础技术。并发程序的动态分析技术具有代价相对较低、正确性 (soundness) 容易保证等优点，是目前最有效的并发程序质量保障手段之一，也是工业实践中应用最广泛的一类技术。为实现并发程序的动态分析，就必须在运行时观测并发程序的行为。不同于串行程序，并发程序执行包含线程间的共享资源依赖关系，这导致应用于串行程序的动态分析技术无法直接在并发程序中应用。这些依赖关系必须在运行时予以捕获或记录，才能实现对并发程序执行轨迹的分析和调控。

本文关注并发程序共享内存访问的数据依赖 (共享内存访问依赖，简称访存依赖) 相关的研究问题。由于共享内存是并发程序中最常见的共享资源也是实现其他类型共享资源的基础，因此实现访存依赖的高效观测和有效使用是实现并发程序动态分析的基本问题之一。也因并发程序执行中共享内存访问数量众多，使用朴素方法获取访存依赖将导致数千倍的运行时开销和海量的事件日志。如何设计、实现并发程序动态分析技术和与之适应的访存依赖的获取技术是十分具有挑战性的难题。

本文完成了对访存依赖相关的研究框架、获取技术和基于访存依赖的动态分析三方面的研究工作：

1. 针对目前对访存依赖研究缺乏系统性的现状，本文首先提出了访存依赖的理论框架与技术框架。理论框架帮助我们系统性地理解访存依赖获取这一问题，其中包括系统模型和访存依赖的形式化定义、访存依赖获取的问题空间。理论框架还提出了并发程序共享内存访问的局部性理论，刻画了现实程序访问共享内存的特征，为后续深入研究奠定了基础；技术框架帮助我们在统一的视角下研究来自体系结构、计算机系统、程序设计语言和软件工程四个研究领域的相关工作。技术框架包含三个要素：访存依赖获取

技术的四个评价指标、获取访存依赖的两类技术、访存依赖在并发程序动态分析中的两类应用。在技术框架下实现了对现有访存依赖相关技术所作出的权衡的全面综述，为进一步研究奠定了基础。

2. 在理论框架和技术框架的共同指导下，提出了利用线程和空间-线程局部性的高效访存依赖获取技术。基于乐观锁的高效访存依赖获取技术 RWTrace 在不引入额外同步/原子操作的前提下，能实现 $O(1)$ 时间对读操作的线程局部性检测，从而实现高效的访存依赖在线追踪，相比于朴素的互斥锁在访存密集型的程序上实现多达数百倍的性能提升；基于二分组 (bisectional coordination) 协议的高效访存依赖约减技术在运行时动态维护地址空间的区间划分，使得区间中的变量具有空间-线程局部性并被视作一个整体处理从而实现访存依赖约减，相比于高度优化的 RWTrace，在仅付出 0–54.7% 运行时开销的前提下实现多达 97% 的依赖减少。
3. 基于在访存依赖获取问题研究中的经验，提出基于访存依赖的动态分析技术：基于缓存的高效执行重放技术 CARE，其通过数值预测缓存实现近似的检测线程局部性检测，通过启发式算法离线合成访存依赖，实现高效、简洁的运行时记录；基于二分组协议的高效数据竞争检测和假共享检测算法，通过避免对整个变量组进行不必要的检测，能大幅减少检测开销和元数据维护的数量；基于依赖反转技术的应用程序崩溃一致性自动检测技术 C^3 和移动应用的并发测试技术 AATT，在知名开源软件中找到了前所未知的缺陷。

最后，基于二十多年来的研究都未能找到只进行 wait-free 修改实现准确访存依赖追踪的事实，本文提出没有免费 (只进行 wait-free 的修改) 午餐 (在多项式时间内获得满足顺序一致性的访存依赖) 的猜想。在复杂性探讨中，我们给出顺序一致性判定 (VSC) 问题的一个新证明，并据此证明了“没有免费午餐”猜想的特殊情形。

关键词：访存依赖；并发；并发程序；动态分析

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Understanding Shared Memory Dependences in Concurrent Programs

MAJOR IN: Computer Software and Theory

AUTHOR: Yanyan Jiang

SUPERVISOR: Professor Jian Lu, Xiaoxing Ma, and Chang Xu

Abstract

Concurrent programming has been popular since the widespread of multi-core processors and increased support in programming languages. However, the non-deterministic nature of concurrent programs brings new challenges: concurrency bugs are notoriously difficult to trigger, reproduce, and detect. Improving the quality of concurrent programs has become a major challenge.

This paper focuses on the dynamic analysis of concurrent programs in which a concurrent program's execution is observed and analyzed. Dynamic analysis is one of the most effective and efficient techniques towards quality concurrent programs because it strikes a good balance point between efficiency and completeness. Since dynamic analyses are based on execution traces and concurrent programs have data/resource dependences across threads, the characteristics of such dependences must be understood to realize effective and efficient dynamic analyses of concurrent programs.

This paper particularly focuses on a common problem faced by all dynamic analyses of concurrent programs: obtaining shared memory dependences, the order of shared memory accesses. Since the non-determinism of a concurrent program is mainly attributed to non-deterministic accesses to shared resources (in particular, the shared memory), the dependences (ordering) between these accesses must be faithfully logged and analyzed. However, due to the huge amount of shared memory accesses, how to effectively and efficiently obtain such dependences is a central problem that serves as the cornerstone of dynamic analyses of concurrent programs.

This paper studies three aspects of shared memory dependences: the research framework, obtaining shared memory dependences, and dynamic analyses based on shared memory dependences:

1. Realizing that there lacks a systematic understanding of shared memory dependences, we first proposed a theoretical framework of modeling shared memory dependences and a technical framework of studying related work. The theoretical framework contains the formalization of concurrent system, shared memory dependences, and their basic properties. Furthermore, it also includes the theory of locality in which characteristics of shared memory accesses in real-world concurrent program executions are studied. The technical framework, under which we surveyed existing work across several research domains, contains three key elements of shared memory dependences: the criteria of measuring the effectiveness and efficiency of a technique that obtains them; two categories of approaches to obtaining them; and dynamic analysis techniques based on them.
2. Based on these two frameworks, we developed two novel techniques that can effectively and efficiently capture shared memory dependences. Optimistic shared memory dependence tracing technique RWTrace exploits thread locality of concurrent programs. It can detect thread-local reads in $O(1)$ time without synchronization. By not keeping log for such events, it achieved hundreds of times of speed-up compared with a mutex-based implementation; Online shared memory dependence reduction via bisectional coordination exploits spatial-thread locality of concurrent programs. It maintains an interval partition of the address space such that variables in each interval exhibit spatial-thread locality, yielding up to 97% reduced shared memory dependences by paying only 0−54.7% of runtime overhead upon the highly optimized RWTrace.
3. We also studied how to use shared memory dependences in realizing software testing and debugging techniques. Deterministic replay technique CARE leverages the value prediction cache to guide an efficient record-replay of concurrent Java programs. Dynamic analyses (data race detection and false sharing detection) based on bisectional coordination can greatly reduce runtime overhead and metadata maintenance cost. Software testing techniques based on reversing dependences found previously unknown bugs in popular open-source software.

Finally, summarizing the past research, we found that nobody ever achieved a precise shared memory dependence tracing using only wait-free changes to the

program or concurrent system. Therefore, this paper also raises a negative conjecture: there is no free (wait-free modification) lunch (sequentially consistent trace). We provide a proof to a more restricted case of the VSC (verifying sequential consistency) problem, and use a similar technique to prove a special case of the no-free-lunch conjecture.

Keywords: Shared Memory Dependence, Concurrency, Concurrent Program, Dynamic Analysis

目　录

插图清单

附表清单

第一章　绪论

由于多核处理器的广泛部署和编程语言中并发特性的发展，并发程序在近年来得到了迅速的普及。并发程序中的多个线程访问共享资源的顺序不确定性使得并发程序具有巨大的调度空间，导致了由并发性引起的缺陷难触发、难复现、难检测，给并发程序的质量保障带来了重大挑战。

为应对这一挑战，并发程序的动态分析技术由于其基于准确的执行轨迹，从而代价较低且正确性 (soundness) 容易保证，是目前最有效的并发程序质量保障手段之一，也是工业实践中应用最广泛的一类技术。

为实现并发程序的动态分析，就必须实现在运行时观测并发程序的行为。并发程序相较串行程序的主要区别是其具有线程之间的数据依赖 (即访存依赖)，而获取访存依赖则是并发程序动态分析的基本问题。虽然目前有众多关于并发程序动态分析的研究，但对“访存依赖”这一基本概念的理解仍是不足的，表现为如下两点：

1. 缺乏对访存依赖的系统性认识。现有技术通常基于“需求驱动 → 经验性质 → 实验验证”的研究思路，使用直觉或经验性质，从某一侧面研究访存依赖相关的某一个具体问题 (如特定场景下的某类缺陷检测技术)，对宏观的问题空间理解不足，研究进展缓慢。
2. 缺乏统一的技术框架以系统化地研究现有技术的优势与不足。由于并发性存在于计算机系统栈的各个层次，因而来自体系结构、计算机系统、程序设计语言、软件工程等多个领域的研究者均在不同侧面研究过访存依赖相关的问题。缺乏统一的技术框架使得相关研究之间的内在关系未被揭示，阻碍了新技术的发现。

立足于这两个问题，本文开展了对访存依赖相关问题的研究：提出研究访存依赖的理论模型和技术框架，并在此基础上研究访存依赖的高效获取技术和基于访存依赖的动态分析。最后，我们还用“没有免费午餐”的猜想讨论访存依赖获取中的“难”和“易”，为未来的研究奠定了坚实的基础。

本章对全文关注的研究问题、研究挑战、研究思路、研究贡献予以了概括。首先介绍必要的背景知识：并发程序 (第 1.1.1 节)、并发程序的质量保障 (第 1.1.2 节) 和并发程序的动态分析 (第 1.1.3 节)，然后引入访存依赖的概念 (第 1.2.1 节) 并讨论其与并发程序动态分析的关系 (第 1.2.2 节)，提出访存依赖获取问题 (第 1.2.3 节)。之后，我们介绍目前的研究挑战 (第 1.3.1 节) 以及针对研究挑战制定的研究思路 (第 1.3.2 节)，并总结全文的研究贡献 (第 1.3.3 节)。

1.1 研究背景

1.1.1 并发程序

并发程序通过在程序中引入多个串行的指令执行流，以通过共享资源的方式通信协作来完成计算任务 [18]。本文讨论的范畴限定为运行在单处理器或多处理器的共享内存计算机系统[1]上的并发程序。如图 1.1 所示，区别于串行程序 (左)，并发程序 (右) 的数据依赖可能来自其他线程[2]。

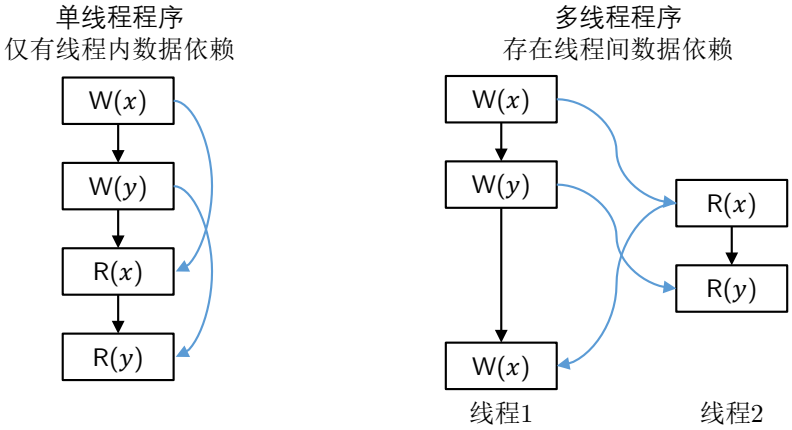

图 1.1 串行程序与并发程序区别示意图。

在程序中引入并发性主要目的如下 [14]：

1. 实现多任务处理：在单处理器或多处理器系统上通过分时和调度使得计算机系统呈现同时执行多个任务的能力。这一能力大幅降低了需要同时服务多个目标对象的服务器程序的编程难度，同时也是有效利用多处理器系统并行计算资源的必要基础；
2. 提高单处理器计算资源的利用率：在处理器需要等待外部资源 (如 I/O 或套接字数据) 时能调度同一计算任务中的其他部分而不必将计算资源浪费在等待外部资源上；
3. 有效利用多处理器系统中的并行计算资源：并发程序能在处理器数量增加时将计算任务分派到多个处理器上同时执行，从而实现相对于串行程序 (同一时刻只能运行在一个处理器上) 的计算加速。

[1] 常见的单结点计算设备 (如嵌入式设备、移动智能设备、个人电脑、工作站和服务器) 均是此类计算机系统。由网络连接、以消息传递进行通信的分布式计算机系统 (包括具有分布式共享内存的系统) 不在本文讨论之列。

[2] 如文中如不作特别说明，执行轨迹中的每一方框代表一个共享内存访问事件，其中 $\mathsf{R}(x)$ 表示对变量 x 的读事件，$\mathsf{W}(x)$ 表示对变量 x 的写事件，事件按照从上到下的顺序发生。

并发程序在近年来迅速普及，其原因有二：(1) 在硬件方面，随着摩尔定律在单处理器上的终结，单个处理器性能提升遇到瓶颈，处理器进入“横向发展”时代，共享内存多处理器系统迅速普及 [65]，必须使用并发程序才能有效利用多处理器的并行计算资源；(2) 在软件方面，现代程序设计语言对并发性的支持越来越好。不同于早期程序设计语言 (如 C 语言) 对并发性的支持很少而必须通过外部库 (如 pthreads) 实现，现代程序设计语言 (如 Java, Go 等) 则对并发性有了原生支持，更进一步促进了并发程序的普及。

并发程序由多个*线程*组成，其中每个线程都可以看作是一个串行程序。并发程序中的线程可以访问*共享资源* (资源包括内存变量、操作系统资源如描述符和句柄等) 以实现线程间通信，协作完成计算任务。并发程序运行时的软件平台和硬件系统称为*并发系统*。并发系统在运行时维护各个线程的状态、调度线程执行并确定共享资源访问的规约。例如，基于 POSIX 线程库 [6] 编写的 C 语言并发程序，其中的线程执行其被 `pthread_create` 函数创建时所指定的入口函数代码，并且可以访问共享内存或通过系统调用访问操作系统中的共享资源。同一个基于 POSIX 线程库编写的 C 语言并发程序在不同并发系统上 (处理器和操作系统) 的运行时行为可能具有一定差异 (如不同处理器具有不同的内存模型)；再如，Java 语言在语言级上支持线程，并发程序中的每个线程均对应一个 `java.lang.Thread` 类型对象，且该对象包含此线程的元数据和代码。Java 虚拟机中的所有线程拥有一个共享的堆区，其运行时行为则符合 JSR-133 标准 [7] 的规定。

1.1.2 并发程序的质量保障

并发程序的执行具有*不确定性* (*non-determinism*)，其原因是并发系统通常对线程访问共享资源的顺序不做任何限制。由于各个线程在多次执行中所处的环境不同[1]，因而其中指令执行的相对速度不同，多次运行同一并发程序可能获得截然不同的指令执行顺序、程序运行时状态和输出结果。由于并发程序执行的不确定性，在包含 t 个线程、每个线程访问 n 次共享资源的并发程序中，不同的调度数量达到指数级的 $O(t^n)$。由于现实程序中的 n 通常非常大 (取决于程序执行的长度，甚至远大于输入的数量)，并发程序通常具有巨大的调度空间 (interleaving space)。

并发程序的不确定性导致了并发缺陷*难触发、难复现、难检测*。首先，有些并发缺陷必须由特定的线程调度才能触发。其次，并发程序有指数级的输入空间 (输入长度为 k 的程序有 $2^{O(k)}$ 种合法的输入)。若并发程序中的缺陷必须

[1]线程的运行受到系统设置、处理器内部状态 (如缓存)、外部输入、中断、系统中其他线程等的影响。

在指定的输入和特定的线程调度下才能触发，在有限的测试资源约束下找到包含缺陷的输入和调度就更加困难。而即便在运行时触发了并发缺陷 (如导致程序崩溃)，程序员仅根据错误的现场信息 (如崩溃时的寄存器、栈区和堆区的快照) 有时也难以在测试环境中重现错误和诊断出错的原因。

并发程序不确定性致使并发缺陷具有难触发、难复现、难检测的特点，而使得并发缺陷成为软件系统中的“定时炸弹”，随时威胁软件系统的可靠性。以下列举一些由并发缺陷而引发重大损失的案例：

1. 1985 至 1987 年 Therac-25 放射性治疗仪的软件缺陷导致了 6 起患者接受过量辐射的人员伤亡事件，是迄今为止最严重的放射性治疗仪引起的医疗事故。事故的原因是 Therac-25 软件在某种特定的并发事件调度下触发了非预期的竞争条件 (race condition)，违反了程序员假设的原子性，造成了治疗仪反馈给操作员的状态与治疗仪实际状态之间的不一致，最终使患者接受过量的辐射 [88]。
2. 2003 年 8 月北美洲东北部的大停电波及了加拿大和美国的逾五千万人口，造成了数百亿美元的经济损失。引发事故的原因之一是 XA/21 电网管理系统中的一个竞争条件使得电网的自动警报系统无法正常工作，最终令操作员对情况误判而未能及时调配电网负载，致使本可控制在局部地区的小规模停电演变为大范围的电网过载瘫痪 [39]。

在并发程序广泛普及的今天，并发程序的数量和规模日益增长、用户不断增加，如何在测试阶段暴露缺陷、在调试阶段诊断缺陷、在运行阶段避免缺陷从而提高并发程序的质量是十分具有挑战性的难题。

近二十年来，来自计算机体系结构、计算机系统、程序设计语言、软件工程等领域的研究者向这一问题发起了挑战，并提出了多种并发程序质量保障技术，取得了卓有成效的研究成果。目前，并发程序的质量保障技术根据其作用的对象大体可以分为三类：

1. 语言级技术在并发程序编写之前即对其行为进行约束，在语言层面 (或库函数层面) 对共享资源的访问进行限制，以从源头避免一些程序员在程序中引入并发缺陷的可能性。例如 Scala 语言 [9] 提倡使用 Actor Model [11] 在多个动作者之间发送同步或异步的消息、使用 Future 处理异步事件。这一模型倡导“仅使用消息传递进行线程间通信”的并发编程模式，在具有足够强表达能力的同时彻底消除一些由资源竞争引起的并发缺陷。
2. 代码级技术在并发程序编写完成后对其代码进行静态分析或验证，在程序被运行前即可发现其中潜在的缺陷或缺陷模式并报告给程序员复审。对于规模较小但运行在安全性攸关场景下的并发程序，可以通过模型检验技术

[58] 枚举其所有可能的执行，并证明其满足某些安全性质 (例如不会发生死锁或活锁) 或符合程序规约；对于规模更大的复杂程序，我们也可以对其中的一些常见并发缺陷模式进行静态检测，如检测程序中可能存在数据竞争的共享内存访问 [125]。

3. 执行级技术在有测试用例时或在生产环境中实际运行程序，并通过修改并发程序或运行时系统实现对程序运行时状态的监控，在并发缺陷发生前予以预测或规避、发生时予以检测、发生后予以修复。例如，通过修改运行时系统约束并发程序的调度空间，可以避免运行时与测试环境中差异较大的调度而触发缺陷 [40]。在程序运行时记录的信息还可用于重现并发缺陷的触发过程 [51]、预测并发程序中的缺陷 [55] 或生成更多的测试用例 [102]。

对比在三方面展开的并发程序质量保障技术，语言级技术能从根本上避免一部分并发缺陷的发生，但同时也可能引入新类型的并发缺陷，且无法适用于已用现有编程语言和库函数编写的代码，并在某些场合 (如需要高并发性能的数据库或服务器应用) 下使用受限的并发模式致使无法达到既定的目标。代码级技术原则上能应用于任何程序，直接根据程序代码产生缺陷报告，但由于程序设计语言的 Turing 完备性、输入空间的隐含规约等问题，模型检验和静态代码分析在大规模现实程序上的实用性仍面临重大挑战，为实用化而作出的近似处理 (如不准确的别名分析、无法建模隐式控制流等) 也导致了误报、漏报等问题。执行级技术具有最佳的可用性，只要程序能够执行，其执行轨迹就能用于分析，但其分析结果仅限于一次执行，无法推广到并发程序的其他输入和调度。

1.1.3 并发程序的动态分析

本文关注执行级的并发程序质量保障技术。在执行级对程序运行时状态予以监测和调控的技术又称为*动态分析技术* (dynamic analysis) [54]。动态分析最早被用于串行程序的测试、调试和分析，之后则在并发程序中被广泛研究。并发程序的动态分析技术通过对程序进行插装或对运行系统进行修改，而后观测/调控并发程序的运行时行为，从而实现各类并发程序的质量保障技术——缺陷检测、缺陷诊断、并发控制等。这类动态分析技术已在并发程序质量保障方面成效卓著，研究成果主要体现在轨迹分析和运行时并发控制两个方面。轨迹分析包括动态数据流分析 [59]、并发缺陷检测 [95, 116]、轨迹重放调试 [84] 等技术；运行时并发控制包括事务内存 [47, 63, 120]、数据竞争避免 [46, 117] 和确定多线程 [40, 45] 等技术。

相比于语言级的技术 (如使用特定的并发模型)，动态分析不需要其使用者 (并发程序的开发者或测试人员) 付出额外的学习成本和项目重构代价，只要有

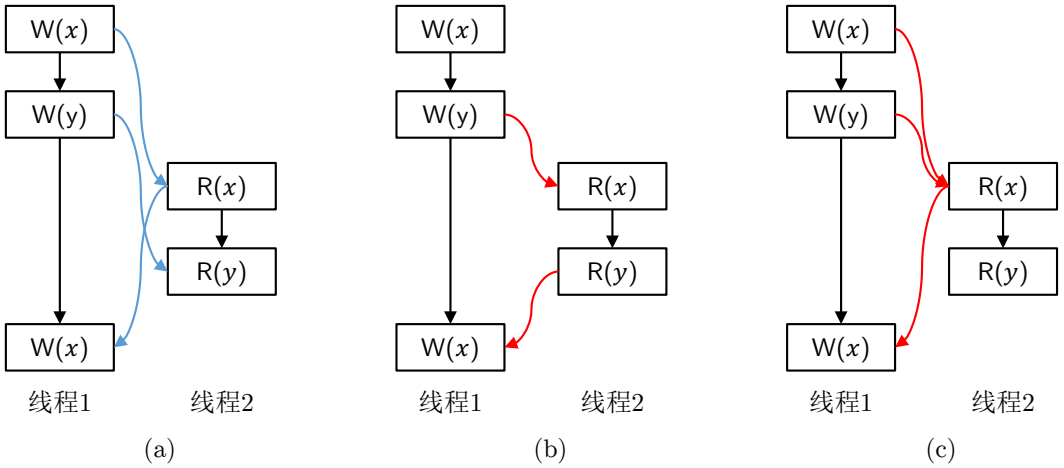

图 1.2 *访存依赖示意图。图 (a) 为数据依赖，图 (b), (c) 为访存依赖。*

并发程序的对应测试用例即可开展动态分析，动态分析工具通常是全自动的。相比于代码级的技术 (如静态代码分析器)，动态分析基于一次确定的程序执行，因此无需考虑代码级的分支、循环、调度的多种不确定性可能，通常效率较高，能支持大规模的实际程序，且分析的结果通常是准确 (sound) 的，产生误报的情形较少。动态分析是目前最有效的并发程序质量保障手段之一，也是工业实践中应用最广泛的一类技术，因此本文关注于并发程序动态分析的相关技术。

1.2 访存依赖与并发程序的动态分析

1.2.1 访存依赖

区别于串行程序在给定输入的情况下具有完全确定的控制流和数据流，并发程序中的线程通过共享资源实现线程间通信，合作完成计算任务，因此线程访问资源的先后次序会对程序行为有重大影响。这使得并发程序拥有巨大的调度空间而难测试、难调试、质量难以保障。

观测并发程序执行中线程访问共享资源的*顺序*是理解并发程序执行的必要条件，也是并发程序动态分析的基础 (将在下节中讨论)。在本文讨论范围的单/多处理器共享内存计算机系统中，线程间通信主要由*共享内存*实现。此类系统中，共享内存是访问延迟最小 (纳秒级)、代价最低 (仅需一条或几条机器指令)、数量最大 (多达数十 GB) 的共享资源，且其他类型的资源 (如锁、消息机制、并发数据结构等) 均可基于共享内存实现。因此对并发程序共享内存访问顺序进行观测能够代表对并发程序共享资源的观测。

本文的研究对象是*访存依赖* (*shared memory dependence*)，即*并发程序访问共享内存的顺序*。非形式地说，访存依赖指程序执行轨迹中可能产生数据依赖

的共享内存访问事件之间的顺序关系 [28]。即访存依赖是定义在所有共享内存访问事件 E 上的一个二元关系，由其确定任何一对有可能产生线程间数据依赖的共享内存访问事件[1]在执行中实际发生的先后关系。图 1.2a 中蓝色曲线箭头所示为读/写操作之间的数据依赖。图 1.2b 和图 1.2c 中的红色曲线箭头分别描述了两个访存依赖关系，均能代表图 1.2a 中的数据依赖。访存依赖及其相关概念将在第 2 章中进行形式化定义。

访存依赖包含了具有潜在数据依赖事件之间的顺序关系。因此根据访存依赖可以对并发程序的执行轨迹进行串行化。在外部输入确定的前提下，任意多次按照访存依赖的规定顺序执行并发程序，都将得到完全相同的执行结果 (包括输出、程序状态等)。早期的研究已观测到访存依赖的这一性质，并将其用于实现并发程序的执行重放 [112]。获取访存依赖是实现并发程序动态分析的基石。

1.2.2 访存依赖与动态分析

Lu 等人调研了开源软件中的并发缺陷模式 [94]，除较为容易检测的死锁缺陷 (deadlock bug) 外，剩余缺陷的 97% 可归结为原子性违反 (atomicity violation, 占 67%) 和顺序违反 (order violation, 占 33%)。因此这两类缺陷的检测、复现、诊断、避免即成为并发程序动态分析的主要目标。这两类问题均是由于对共享资源 (本文中研究共享内存) 的访问顺序不当导致的，因此获得共享资源的访问顺序 (即访存依赖) 是并发程序动态分析中的重要问题。

并发程序的动态分析根据其产生作用的时间点可以分为两类：运行后 (post-mortem) 的*轨迹* (*trace*) 分析技术对程序执行轨迹中的各类特征进行分析，从而实现并发缺陷的检测、复现和诊断等功能；运行时的*并发控制*技术在运行时检测程序的执行，并在必要时改变线程访问内存的顺序，从而实现事务语义、缺陷避免、消除不确定性等质量保障技术。

对于这两类动态分析技术，获取访存依赖都是其基础问题和先决条件：

1. 对于轨迹分析技术，重点是得到执行轨迹。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，即按照与执行轨迹一致的访存依赖顺序执行程序 [84]。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缺陷 (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 [49, 115]，使用与数据依赖完全一致的访存依赖即可对执行轨迹中的数据竞争予以检测。基于同一执行轨迹，研究者还提出了其他定义共享内存访问操作之间因果性的因果模型

[1]仅当以下三个条件满足时，一对事件可能产生线程间数据依赖：(1) 访问同一内存地址；(2) 发生在不同线程；(3) 至少有一个是写操作。此定义与数据竞争 (data race) 的定义类似。

(causal model) [118, 122]，并将因果模型实现为预测能力更强的并发程序缺陷检测技术 [71]。此外，执行轨迹中的访存依赖信息还能指导如原子性违反 [95] 等其他类型缺陷的发现，或为后续主动测试提供目标 [116]。

2. 对于并发控制技术，需要在程序运行时观测程序的状态并对其行为进行调整。并发控制实现的关键技术是在共享内存访问发生前即获得访存依赖信息，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并要求在线获取准确的访存依赖，因此其设计通常也是最具挑战的。并发控制技术通过调整并发程序的执行轨迹以实现开发者预期的功能或避免并发缺陷，以提高并发程序的质量。软件/硬件事务内存相关研究近年来已取得一定的进展 [50, 61]，Intel 已开始尝试在商业处理器中集成事务内存支持并在一些应用场景取得了显著的性能提升 [133]；为了进一步避免并发缺陷，运行时系统还可以主动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 [137]，从而避免有害数据竞争引发的并发缺陷；减少并发缺陷的终极目标是确定多线程 [40, 45]：采取确定的规则对系统内的共享资源进行调度，从而尽可能降低不确定性带来并发缺陷的可能。

综上所述，*获取访存依赖是对并发程序进行动态分析的基础*。由于现代多处理器系统访问共享内存的代价很低，并发程序的共享内存访问通常数量大、频率高。故如何在获取访存依赖的同时不对程序执行的速度、内存使用、语义等产生过多负面影响，是实现高效动态分析技术的一大挑战。

1.2.3 访存依赖的获取问题

朴素的访存依赖获取技术只需修改运行时系统或对共享内存访问进行插装，在共享内存访问前后插入同步操作将所有对共享内存的访问串行化，并在同时记录全局的时间戳，即可得到所有共享内存访问的全序，满足动态分析的需求。然而，朴素的访存依赖获取可能带来巨大的开销 [22]，而其串行化内存访问与并发程序利用多处理器并行计算资源的初衷亦背道而驰。

为了实现更高效的并发程序动态分析来提高并发程序的质量，针对其具体应用场景设计高效的访存依赖获取技术是一个具有挑战性的难题。访存依赖的获取技术在近年来受到越来越多研究者的关注。鉴于并发性存在于计算机系统栈的各个层次，根据研究对象和研究手段的不同，现有的研究工作横跨体系结构、计算机系统、程序设计语言和软件工程等领域。

访存依赖的获取是通过程序插装 (编译时插装 [69]、二进制改写 [22])、修改运行时系统 (虚拟机 [28] 或定制硬件 [17])，在程序执行的过程中记录直接或间

接信息后，在线或离线推导得到。根据动态分析技术的需求，高效的访存依赖获取技术应当具有以下特点：

1. 在访存事件发生前就能获得依赖关系，从而能实现对程序行为的及时控制以避免并发缺陷的触发；
2. 对运行时系统造成的时间开销和语义影响尽可能小，从而能降低在开发环境下的开销并且能够应用于生产系统；
3. 能在满足动态分析需求的前提下记录数量尽可能的少，从而减少动态分析技术需要分析的访存依赖数量、提高其效率。

显然，“获取及时”、“开销小”、“数量少” 的目标是难以同时实现的。加之访存依赖的获取技术与其支持的动态分析技术相关，此即定义了一个问题空间。目前，根据访存依赖获取技术是否能在访存事件发生前即得到依赖，主要分为直接获取 (在线追踪) 和间接获取 (离线合成) 两类。

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作出较大修改，高效性和简化性不足。访存依赖追踪技术直接在内存访问发生前即得到。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 保护对变量 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，是最早研究的一类技术 [84]，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，从而最大程度发挥线程之间的并行性。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 [15]。

为实现高效性和简化性，访存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，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 [129]、按对象指派 [124] 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 [80])。锁的实现可以借助定制硬件 [129]、存储保护 [52]，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 [28, 79]。

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。应用需根据其场景 (对评价指标的需求) 选择合适的访存依赖获取技术。除直接追踪外，访存依赖的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，如缺陷定位 [136] 和轨迹预测分

析 [71]。由于不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 [87]，因此间接记录也更加灵活，也通常更加高效、简化。

因此，合成访存依赖的核心问题是通过选取记录不同的间接信息以实现对准确性、高效性、简化性和合成时间开销的权衡。如只记录少量框架性的信息，则合成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 [109, 136]。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在每个线程执行效果等价的依赖，但依赖合成是目前技术难以求解的 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)，推导出的访存依赖能够提供保证重现等价执行的准确性 [144]。

1.3 本文工作

1.3.1 研究挑战

访存依赖是实现并发程序动态分析的基础。目前尚无一项单一的访存依赖获取技术能满足所有动态分析技术的需求。针对动态分析对即时性、数量、开销等的要求，访存依赖获取技术在各方面的指标之间进行权衡。然而，“访存依赖”及其相关问题仍然主要是作为其应用的附属技术研究，缺乏系统化的研究和理解，目前的研究主要有两点局限：

首先，缺乏对访存依赖这一事物的理论认识及量化研究。因此，现有研究工作的基本思路通常是“需求驱动 $\rightarrow$ 经验性质 $\rightarrow$ 实验验证”。为了实现高效的动态分析，现有研究通常从一些实际观察出发，使用并发程序的一些经验性质 (如线程局部性 [144]、读/写不对称性 [28]、同步局部性 [34])，依靠启发式优化 [90] 或约束求解 [87] 实现，然后再以现实中的基准程序或并发缺陷予以验证。这一流程确实促进了并发程序质量保障技术的发展，但与此同时，由于缺少对访存依赖自身性质的认识，基于“试错”的研究效率有待提高且研究进展缓慢，现有的技术仍然难以满足很多动态分析对性能等指标的要求。

其次，缺乏统一的技术框架用以研究现有技术的优势与不足，无法系统化地指导新的访存依赖相关技术的发展。由于并发性存在于计算机系统栈的各个层次，体系结构、计算机系统、程序设计语言和软件工程等领域均展开了对并发程序动态分析的相关研究。由于未曾意识到获取访存依赖是这些技术的基础，虽然这些研究具有内在关联，研究技术也相互重叠，但技术体系尚未打通，阻碍了技术的进一步发展。例如在体系结构领域使用全局时钟为访存事件定序的技术 [34] 在软件工程领域也得到了应用 [135]。

由于存在这两点局限，目前很多并发程序的动态分析技术开销巨大，仍然只能停留在实验室或测试阶段：执行重放、数据竞争检测等技术在共享内存访问密集型的应用中可能带来数百或数千倍的时间开销，而使其实用化受到较大阻碍。受限于目前研究方法落后的原因，相关研究进展较为缓慢。另一方面，尽管现有动态分析技术在缺陷检测方面取得了卓著的成效，大型软件、新兴软件 (如移动应用) 中依然包含很多缺陷，如何将访存依赖 (及其推广) 用于缺陷检测实现软件质量的提升，也是具有挑战性的研究问题。

1.3.2 研究思路

本文认为，访存依赖获取研究的最大挑战即在于针对目前研究的两点局限，全面地理解访存依赖的定义和性质，并系统化地分析现有技术，以找出其中的研究契机。只有突破这两项挑战，才能指导系统化的访存依赖获取技术研究，并将它们应用到并发程序的动态分析中去。

针对缺乏访存依赖获取问题理论认识的研究现状，首先应明确问题的定义和范围，即形式化地定义访存依赖及其获取问题。在此基础上，我们注意到现实中的程序访问共享内存的模式既非随机亦非固定，因而在形式化的框架下研究访存依赖所具有的性质，即访存依赖的理论建模，对设计高效的访存依赖获取技术有重要指导意义。最后，我们还希望能从理论研究中认识到现有工作的固有局限性，即在何种条件下，获取访存依赖具有怎样的难度。

而针对缺乏统一框架研究现有技术的优势与不足的研究现状，本文在理论部分的指导下提出能容纳现有访存依赖获取技术的框架，并在框架中寻找实现高效访存依赖获取技术的研究契机。获取访存依赖技术本身的挑战是在即时性 (在访存事件发生前就能获得依赖关系)、准确性 (能获得每个变量准确的数据依赖)、高效性 (不引起额外开销和调度变化) 和简化性 (数量少) 之间作出权衡。技术部分的研究主要关注于寻找这些技术指标中的新平衡点。

应用上述理论模型和技术框架，我们根据并发程序访问共享内存的特征，吸取技术框架中已有技术的长处、弥补现有技术的缺点，特别关注局部性理论在访存依赖获取技术中的应用，提出了能取得数量级提升的高效访存依赖获取技术，从而实现高效的并发程序动态分析。此外，我们还注意到“反转”访存依赖将导致新的、不同的执行而触发并发程序中潜在的缺陷。本文还根据反转数据依赖关系这一思想，提出一系列软件质量保障技术，寻找开源软件中的前所未知缺陷，以提高其质量。

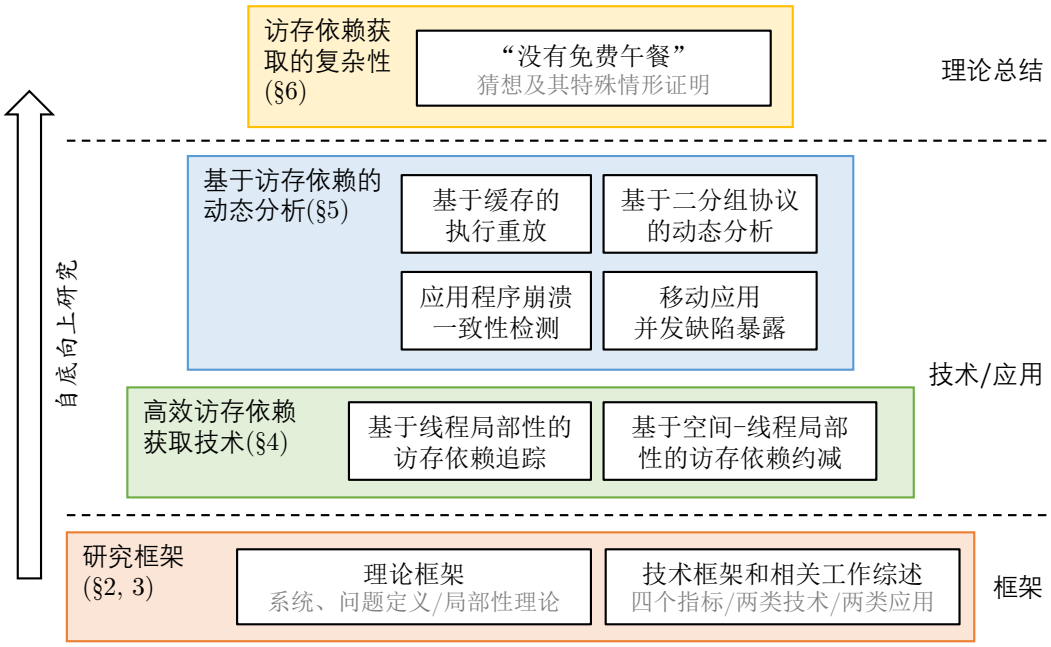

图 1.3 本文研究贡献总结。

1.3.3 研究贡献

基于上述研究思路开展研究，本文的主要工作总结如图 1.3 所示，现概述研究贡献如下：

首先，针对访存依赖相关问题缺乏理论、技术框架的现状，我们对来自体系结构、计算机系统、程序设计语言、软件工程四个领域的相关工作进行了归纳总结，在此基础上提出访存依赖的理论框架与技术框架：

1. 理论框架对访存依赖、访存依赖获取问题进行了形式化定义和建模。针对“共享内存访问具有局部性”这一传统认识，对线程、空间-线程、同步局部性进行了形式化定义，提出了现实程序具有的局部性模型，并针对现实程序进行了局部性的量化研究；
2. 技术框架则对访存依赖相关的研究工作进行了理解和探讨。技术框架包含三个要素：访存依赖获取技术的评价指标 (即时性、准确性、高效性、简化性)、两类访存依赖获取技术 (在线追踪、离线合成)、两类基于访存依赖的并发程序动态分析 (轨迹分析、并发控制)。这一技术框架成功统一了来自多个领域的研究工作，深化了我们对访存依赖获取技术研究进展、研究挑战和研究机遇的理解。

其次，在理论框架、技术框架的共同指导下，我们发现利用并发程序具有的各类局部性是改善现有访存依赖获取技术运行时开销大、日志数量多的有效手段。据此，我们提出两项访存依赖获取的新技术：

1. 基于访存依赖具有的线程局部性，提出基于乐观锁的访存依赖追踪技术 RWTrace。其能在不引入额外同步操作的前提下在 $O(1)$ 的时间内有效检测读操作是否具有线程局部性，以实现高效的访存依赖获取。相比于朴素的互斥锁，在访存密集型的程序上能实现多达数百倍的性能提升；
2. 基于访存依赖具有的空间-线程局部性，提出二分组 (bisectional coordination) 协议实现在线访存依赖约减。其能在运行时维护地址空间的区间划分，使得区间中的共享内存地址具有空间-线程局部性，并可整体化处理，从而实现访存依赖的约减。相比于高度优化的 RWTrace，在仅付出 0–54.7% 运行时开销的前提下，实现多达 97% 的访存依赖数量约减。

再次，基于上述两个框架和设计高效访存依赖获取算法的经验，我们进一步研究了访存依赖在动态分析中的应用，提出了以下可针对实际程序使用的具体动态分析技术：

1. 基于访存依赖的并发程序动态分析技术。其中包括两份研究工作：基于缓存的执行重放技术 CARE 和基于二分组协议的缺陷检测算法。CARE 使用数值预测缓存检测读事件的线程局部性以减少记录数量，并使用离线合成技术实现高效的 Java 程序执行重放调试，相比于使用互斥锁的技术减少多达数十倍的运行时开销和数百倍的访存依赖数量；基于二分组协议的高效数据竞争检测和假共享检测算法则利用二分组协议高效维护具有空间-线程局部性地址空间划分的特性，实现检测调用数量和元数据维护开销的大幅减少；
2. 基于依赖反转 (在多次执行中将依赖顺序进行调换) 技术实现的各类软件测试和分析技术。其中包括两份研究工作：应用程序的崩溃一致性检测技术 C^3 和移动应用并发缺陷暴露技术 AATT。C^3 实现了软件中由于文件系统操作顺序反转引起的原子性违反缺陷的全自动检测。其检出了知名开源软件 (如 GNU coreutils, GNU zip 等) 中的前所未知的崩溃一致性缺陷；AATT 通过同时探索移动应用的输入空间和调度空间，实现了高效的并发缺陷的暴露。其自动检出了流行开源移动应用中前所未知的并发缺陷。

最后，纵观二十多年来访存依赖相关问题的研究成果和本文的研究经验，在仅对并发程序进行 wait-free 修改的前提下，获取准确的访存依赖是这一研究领域的终极理想之一。我们对这一目标提出否定的猜想：“天下没有免费的午餐 (No Free Lunch, NFL)”，即在合理的前提假设下，通过对共享内存访问操作进

行 wait-free 修改，则总存在一个程序，使得推导其满足顺序一致性的访存依赖时间复杂度是 NP-完全的。

我们对这一猜想的证明作出了一些尝试，证明了顺序一致性验证 (VSC) 问题的一个更受限情况，用类似的技术进一步证明了 NFL 猜想的一个特殊情形，并根据这些理论结论对现有的技术进行总结和展望。

1.4 论文组织结构

本文依据研究贡献的逻辑顺序组织如下：

第 2 章给出全文的理论框架，包括形式化的系统模型和问题定义，以及并发程序共享内存访问所具有的局部性特征理论。基于此，第 3 章提出“四个指标、两类技术、两类应用”的技术框架，对访存依赖相关的研究工作进行综述并讨论研究契机。

第 4 章提出我们在理论框架和技术框架指导下设计、实现的访存依赖获取技术：基于乐观锁的访存依赖追踪 (第 4.1 节)、基于二分组协议的访存依赖数量约减 (第 4.2 节)，及其实现技术和实验评估 (第 4.3 节)。

第 5 章描述基于访存依赖的动态分析技术，包括基于缓存和依赖推导的执行重放 (第 5.1.1 节)、基于二分组协议的动态分析 (第 5.1.2 节)、应用程序的崩溃一致性自动检测 (第 5.2.1 节) 和移动应用的并发缺陷暴露 (第 5.2.2 节)。

第 6 章提出“没有免费午餐”的猜想及其形式化定义，并给出其特殊情况的证明，探讨这一理论结果对访存依赖获取问题未来工作的启示。

最后，第 7 章对全文工作予以总结并提出未来可能的研究方向。

第二章　理论框架

对并发程序进行动态分析 (观测或调整并发程序的执行轨迹以实现缺陷的检测、诊断或避免) 是保障并发程序质量的重要手段。然而，由于并发程序不同于串行程序，其存在线程间的数据依赖，无法从单个线程的执行路径中推导出满足全局一致性的执行轨迹，因此获得这些线程间的数据依赖关系 (访存依赖) 是进行并发程序动态分析的先决条件。

因并发程序执行中通常包含海量的共享内存访问，若使用朴素的算法将共享内存访问串行化来实现访存依赖的获取，则会带来巨大的运行时时间开销和海量的依赖日志，无法满足对现实程序实现并发程序动态分析的需求。

为了更好地理解访存依赖，摆脱现有研究工作“需求驱动 $\rightarrow$ 经验性质 $\rightarrow$ 实验验证”思路的局限，本章提出了访存依赖获取问题的理论框架，对访存依赖进行形式化定义、提出访存依赖的问题空间、理解访存依赖所具有的性质。理论框架包含以下两部分：

1. 第 2.1 节明确本文的研究对象。具体而言，我们给出一个字节码语言 Smile 语言及其运行的并发系统、执行轨迹、访存依赖等概念的形式化定义，并在这一简化的系统模型上定义访存依赖获取的问题空间。
2. 第 2.2 节提出本文研究对象所具有的性质。具体而言，我们提出并发程序共享内存访问具有的局部性理论，提出了一个用来形式化和量化“局部性”这一非形式化概念的框架，并用这一框架实现了线程局部性、空间-线程局部性和同步局部性的形式化定义，在现实程序中量化验证了这些局部性的存在。局部性是本文实现高效访存依赖获取的重要技术基础。

2.1　问题定义

给予研究问题一个明确的定义是对其进行系统化研究的先决条件，故本节给出访存依赖的形式化定义：“访存依赖获取”是通过对运行时系统的修改或程序插装，得到访存依赖的过程。我们首先在第 2.1.1 节定义了并发系统的模型，包括 Smile 字节码语言、Smile 程序执行的语义及其内存模型。基于这一模型，我们在第 2.1.2 节给出访存依赖的定义和性质，并在第 2.1.3 节讨论访存依赖获取的两个朴素算法，而由此引发对这一问题空间的初步讨论。

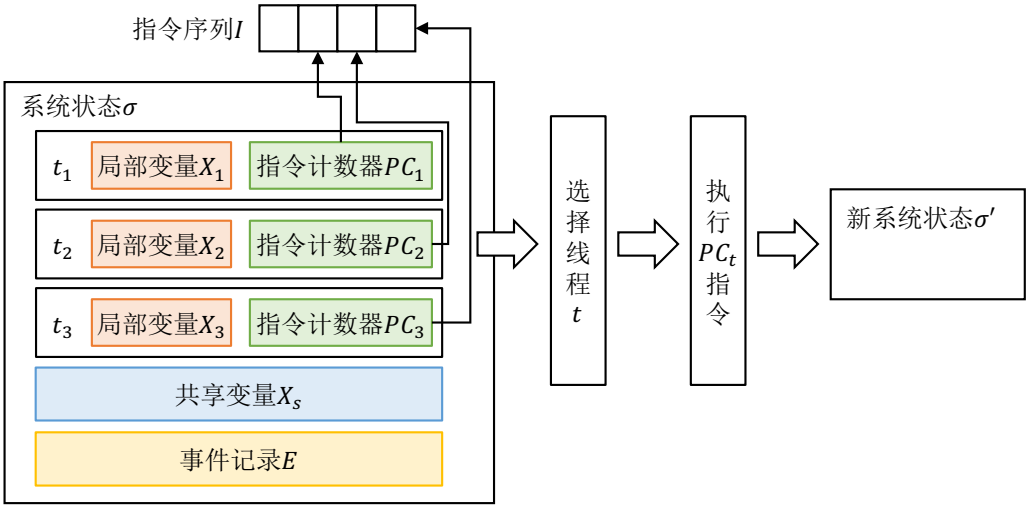

图 2.1 讨论访存依赖相关理论所用的并发系统模型。

2.1.1 系统模型

2.1.1.1 Smile 语言

首先在一个简化的并发系统 (图 2.1) 上给出基于字节码的程序语言 Smile。本文后续讨论均基于此语言。定义 Smile 程序 P 为三元组 $P = \langle T, X, I \rangle$，其中：

1. T 为线程集合，$T = \{t_1, t_2, \ldots, t_{|T|}\}$，$t_i$ 代表编号为 i 线程的标识符。
2. $X = X_s \cup X_t$ 为变量集合，由共享变量集合 $X_s = \{x_1, x_2, \ldots, x_{|X_s|}\}$ 及每个线程的局部变量集合 $X_t = \{x_1^{t_1}, x_2^{t_2}, \ldots, x_{|X_t|}^{t_{|X_t|}}\}$ 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 $I = \{i_1, i_2, \ldots, i_{|I|}\}$，其中指令包括
   (a) 根据局部变量数值 x_i^t 跳转：if (x_k^t) goto i_j；
   (b) 读取共享内存变量：$x_k^t = R(x_{k'})$；
   (c) 写入共享内存变量：$W(x_k, x_{k'}^t)$；
   (d) 线程本地的局部变量运算：$x_k^t = op(x_k^t, x_{k'}^t)$。

Smile 语言包含了共享内存的读/写操作和线程本地的 Turing-Complete 计算，既能反映并发程序访问共享内存的复杂性，亦较为简洁，易于研究其性质。

2.1.1.2 Smile 程序语义

定义并发系统在运行时的状态 $\sigma = \langle V, PC, E \rangle$，其中：

1. $V : X \mapsto \mathbb{Z}$ 将每个变量映射到其当前数值；
2. $PC : T \mapsto \mathbb{N}$ 将每个线程标识符映射到其下条指令的编号；

表 2.1 Smile 程序 P 在状态 $\sigma = \langle V, PC, E\rangle$ 时选中线程 t 执行的语义。

名称	$PC(t)$ 处指令	条件	新状态 σ'
分支 (真)	if (x_k^t) goto i_j	$V(x_k^t) \neq 0$	$\langle V, PC[t \mapsto j], E\rangle$
分支 (假)	if (x_k^t) goto i_j	$V(x_k^t) = 0$	$\langle V, PC[t \mapsto PC(t)+1], E\rangle$
读共享内存	$x_k^t = R(x_{k'})$	$\top$	$\langle V[x_k^t \mapsto V(x_{k'})],$ $PC[t \mapsto PC(t)+1],$ $E \cup \{R(t, x_{k'}, V(x_{k'})\}\rangle$
写共享内存	$W(x_k, x_{k'}^t)$	$\top$	$\langle V[x_k \mapsto V(x_{k'}^t)],$ $PC[t \mapsto PC(t)+1],$ $E \cup \{W(t, x_k, V(x_{k'}^t)\}\rangle$
局部运算	$x_k^t = op(x_k^t, x_{k'}^t)$	$\top$	$\langle V[x_k^t \mapsto op(x_k^t, x_{k'}^t)],$ $PC[t \mapsto PC(t)+1], E\rangle$

3. $E = \{e_1, e_2, \ldots\}$ 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 $e = W(t, x, v)$ 代表线程 t 向变量 x 写入值 v，$e = R(t, x, v)$ 代表线程 t 从变量 x 读出值 v。我们 $e.rw \in \{r, w\}$ 表示事件 e 的类型 (r 代表读事件、w 代表写事件)，用 $e.t$, $e.x$ 和 $e.v$ 分别表示事件 e 发生的线程、访问的变量和读/写的数值[1]。

并发系统的初始状态 $\sigma_0 = \langle V_0, PC_0, E_0\rangle$，其中 $\forall x \in X.\, V_0(x) = 0$，$\forall t \in T.\, PC_0(t) = 1$, $E_0 = \varnothing$。在任意时刻，假设并发系统处于状态 $\sigma = \langle V, PC, E\rangle$，其可以任意选择一个线程 $t \in T$ 并执行一条指令。表 2.1 定义了所有指令的语义，在状态为 σ 时，若线程 t 被选中，则取出 $PC(t)$ 处的指令 $i_{PC(t)}$ 执行，根据表中语义得到新状态 σ'。如此往复，当 t 执行指令后 $PC(t)$ 不再为合法指令 ($PC(t) < 1 \vee PC(t) > |I|$) 时程序结束。因此，一次程序执行可以看作是形如

$$\tau = \left[\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} \sigma_n\right]$$

的执行轨迹，其中 $\sigma_i = \langle V_i, PC_i, E_i\rangle$，$t_i$ 为在第 i 步时选择的线程。$E = E_n$ 即为本次执行的访存事件集合。在此模型中，每一线程都按顺序依次执行指令，故存在指令执行的全序，亦存在 E 中事件的全序 $\prec_{tot}$。定义*程序顺序* (program order, $\prec_{po}$) 为 E 上的最小偏序集使得对于任意线程 t，所有发生在线程 t 中的事件 ($W(t, x, y) \in E$ 或 $R(t, x, y) \in E$) 在 $\prec_{po}$ 中被排序。易见偏序关系 $\prec_{po}$ 是存在且是唯一的，满足 $e_i \prec_{po} e_j$ 当且仅当 $(i < j) \wedge (e_i.t = e_j.t)$。

[1]每次共享内存读/写指令均生成一个新的事件，它与之前所有的事件都不同。因此 E 中可能存在两个不同的事件 e_1, e_2，它们的类型相同，且 $e_1.t = e_2.t \wedge e_1.x = e_2.x \wedge e_1.v = e_2.v$。

在此模型上，容易定义其他类型的指令。例如，定义锁指令 $\texttt{lock}(\ell)$ 和 $\texttt{unlock}(\ell)$ (锁变量 ℓ 可以使用共享变量实现，相应的上锁/解锁指令可以使用过滤锁算法实现 [64]，或借助并发系统提供的其他指令如原子交换、链接读/条件写等实现)，使其能保护一对上锁-解锁操作之间的原子性，即并发系统的执行将保证对于线程 t 执行的一对配对的上锁/解锁操作 $e_l = \texttt{lock}(\ell)$ 和 $e_u = \texttt{unlock}(\ell)$，不存在事件 e 满足 $(e_l \prec_{tot} e \prec_{tot} e_u) \wedge (e.t \neq t) \wedge (e = \texttt{lock}(\ell))$。

2.1.1.3 内存模型

上述系统模型满足*顺序一致性*内存模型 [83]。顺序一致性的定义是存在所有指令的全序，使得在此全序下，对变量 x 的读操作所得数值等于其之前最近一次写操作写入的数值，即对于任意读事件 e_r，都存在一个写事件 e_w 满足：

$$(e_r.x = e_w.x) \wedge (e_r.v = e_w.v) \wedge \\ \Big[\forall e \in E.\,((e.x = e_r.x) \wedge (e.rw = w)) \rightarrow ((e_r \prec_{tot} e) \vee (e \prec_{tot} e_w))\Big]。$$

易见 E 中事件的实际发生顺序 $\prec_{tot}$ 是满足顺序一致性的全序。

实际的并发系统 (如多处理器计算机) 出于性能考虑，通常会对内存模型的一致性进行放松，允许访存指令的乱序执行。常见的松弛内存模型有 TSO (Total Store Order) [119], PSO (Partial Store Order) [75], Java 内存模型 [7] 等。松弛内存模型允许多个线程在某些时刻拥有对全局共享内存不一致的局部视图，从而允许编译器和乱序执行处理器实现更高效的指令调度。松弛内存模型是顺序一致内存模型的弱化实现，即任何满足顺序一致性的执行 τ 也在松弛内存模型下保持一致性 (否则系统行为将违反开发者对 “程序指令串行执行” 的基本假设)，因此本文的讨论主要关注满足顺序一致性的内存模型，且绝大部分情形均能推广到松弛内存模型上。本文将对松弛内存模型系统需要额外修正的情形单独予以讨论[1]。

2.1.2 访存依赖的定义

2.1.2.1 访存依赖的定义

对于访存事件 $e_i, e_j \in E$，定义 e_i 和 e_j 冲突 (用 $e_i \asymp e_j$ 表示)，当且仅当以下条件同时满足：

[1]例如，访存依赖追踪算法 RWTrace [79] 使用了 wait-free 的快速依赖检测算法，其中的多个共享内存访问操作在松弛内存模型下可能被调换顺序。因此我们首先在顺序一致性的内存模型上阐述算法的原理，然后再对算法在松弛内存模型上的行为进行讨论 (证明其在 x86-TSO 上的正确性，以及在更弱内存模型上的修正方法)。

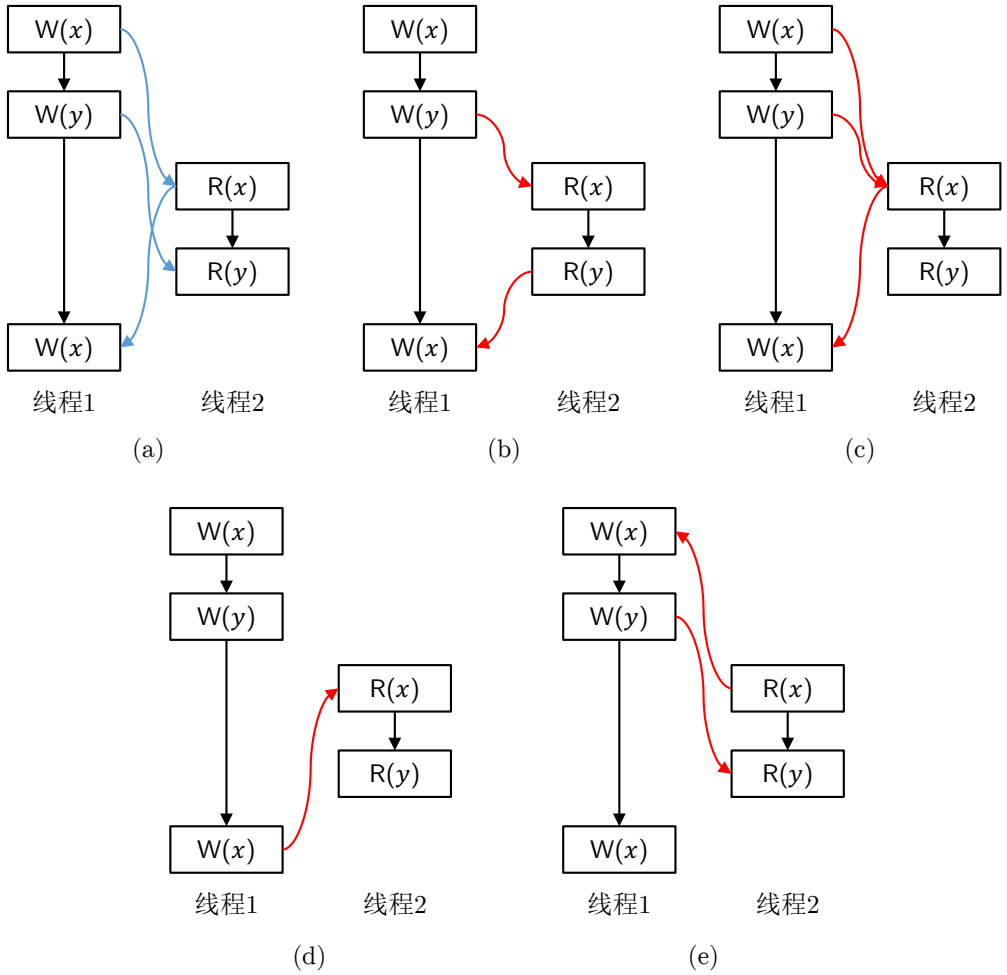

图 2.2 准确/一致/不一致的访存依赖。

1. e_i, e_j 发生在不同线程，即 $e_i.t \neq e_j.t$；
2. e_i, e_j 访问同一共享内存变量，即 $e_i.x = e_j.x$；
3. e_i, e_j 至少有一个是写操作，即 $(e_i.rw = w) \vee (e_j.rw = w)$。

访存依赖定义为满足以下条件的二元关系 $\prec_d \subseteq E \times E$：令 $\prec_{d^*} = tr(\prec_d \cup \prec_{po})$ 为访存依赖 $\prec_d$ 和程序顺序 $\prec_{po}$ 并集的传递闭包，有：

1. 不引发 $\prec_{d^*}$ 中事件的顺序矛盾。即对于任意的 $e_i, e_j \in E$，满足

$$\neg(e_i \prec_{d^*} e_j) \vee \neg(e_j \prec_{d^*} e_i)。$$

2. 确定冲突事件的先后顺序。即对于任意的 $e_i, e_j \in E$ 且 $e_i \asymp e_j$，满足它们的顺序在 $\prec_{d^*}$ 中确定，即

$$(e_i \prec_{d^*} e_j) \vee (e_j \prec_{d^*} e_i)。$$

根据定义，易见 $\prec_{d^*}$ 是满足自反、反对称和传递的偏序关系，$e_i \prec_{d^*} e_j$ 表示 e_i 的发生先于 e_j。换言之，访存依赖蕴含了并发程序共享内存访问事件之间的顺序关系，其能消除由于共享内存访问顺序带来的并发程序执行结果的不确定性。访存依赖的例子如图 2.2 所示。图 2.2a 中的箭头表示原始执行的数据依赖关系，图 2.2b-e 中红色曲线箭头为访存依赖关系，易见其中所有冲突的共享内存访问均被确定顺序。

2.1.2.2 访存依赖的基本性质

显然，$\prec_{tot}$ 满足访存依赖的定义。其次，$\prec_d$ 具有不唯一性，只要满足其定义 (能确定冲突事件发生的顺序) 即可，不必正确反映数据依赖或满足顺序一致性。图 2.2b-e 均展示了访存依赖的例子。

定义 $\prec_d$ 是一致的，仅当 $\prec_d$ 与实际执行中的数据依赖在传递意义上等价。形式地说，$\prec_d$ 是一致的，当且仅当对于任意一对冲突的共享内存访问事件 $e_i \asymp e_j$ $(e_i, e_j \in E)$，满足

$$e_i \prec_d e_j \Leftrightarrow e_i \prec_{tot} e_j。$$

图 2.2b 和图 2.2c 展示了一致的访存依赖，图 2.2d 和图 2.2e 展示了不一致的访存依赖。对于一致的访存依赖 $\prec_d$，按照 $\prec_d$ 所规定的顺序执行程序 P，将保证得到与原执行等价的执行 (每个线程保证执行相同的路径，且所有读写数值将与原执行一致)。这一基本性质是确定性重放并发执行的基础 [112]。

当 $\prec_d$ 与实际的数据依赖完全一致时，$\prec_d$ 被称为是准确的 (对于图 2.2 中的例子，仅有恰好等于图 2.2a 的 $\prec_d$ 才是准确的)。准确的 $\prec_d$ 一定是一致的。

访存依赖 $\prec_d$ 是一个二元关系，因此可以用二元组的集合 $\{(e_i, e_j)\}$ 表示，但也存在其他描述方式，如为每个事件 e 定义时间戳 $tsc(e)$，令 $e_i \prec_d e_j$ 当且仅当 $tsc(e_i) < tsc(e_j)$，或通过记录其他间接信息 (如 $e.x$ 和 $e.v$) 以其他手段 (如使用约束求解器) 计算出访存依赖 $\prec_d$。

2.1.2.3 访存依赖与并发程序的动态分析

一致的访存依赖是众多并发程序动态分析技术正确性 (soundness) 的基础。一致的访存依赖包含了足够的信息以恢复出准确的访存依赖，并实现过往执行的重放 [52, 69, 112]。由于能保证并发缺陷的复现，执行重放技术大幅降低了并发缺陷发生后调试的难度；在运行时检查准确的访存依赖中的数据依赖能检测或预测数据竞争 [49] 等并发缺陷；若在访存事件发生前获得访存依赖，并即时对线程执行顺序予以调控则能实现事务内存 [120] 或确定多线程 [45] 等并发控

$$
\begin{array}{cc}
\texttt{lock}(\ell) & \texttt{lock}(\ell) \\
e: \underline{x_k^t = R(x_{k'})} & e: \underline{W(x_k, x_{k'}^t)} \\
tsc \leftarrow tsc + 1 & tsc \leftarrow tsc + 1 \\
\texttt{log}(\langle e, tsc \rangle) & \texttt{log}(\langle e, tsc \rangle) \\
\texttt{unlock}(\ell) & \texttt{unlock}(\ell)
\end{array}
$$

图 2.3 对读写操作予以包装以获取访存依赖 (ℓ 为全局锁变量)。

制手段。在对性能要求更严苛的场景，如对生产环境的程序实施记录 [12, 109]，则可以适当牺牲访存依赖的一致性，配合搜索等方法得到与原先执行类似的执行轨迹以减少运行时刻的开销。

现实中的并发程序通常会产生海量的共享内存访问，因此在我们所定义的形式化模型上直接记录 (如用朴素的算法，在运行时为每条指令记录其发生的时刻) 的开销对于现实程序通常是难以接受的。尽管这些并发程序的动态分析技术的应用领域各不相同，但其核心挑战均是根据应用场景在各项技术指标之间进行权衡，从而实现有效的轨迹分析或并发控制。这些权衡定义出了本文研究的访存依赖获取的问题空间。

2.1.3 访存依赖获取的问题空间

本文研究的“访存依赖获取”问题定义为通过对运行时系统的修改或程序插装，得到访存依赖的过程。

很容易设计出朴素的获取访存依赖的算法且能保证获得准确或一致的访存依赖。根据 Smile 程序的语义，我们只需对运行时系统进行修改，用同步操作包裹共享内存的读/写指令，即可实现对访存依赖的记录，如图 2.3 所示 (带下划线的指令为共享内存读/写指令)。使用 `lock` 和 `unlock` 保证读写操作和相关记录之间的原子性，从而对于每一个共享内存读写事件 e，都记录下其访问时的时间戳 tsc (以 $\langle e, tsc \rangle$ 的形式保存在日志中)。令 $tsc(e)$ 表示其时间戳，其定义了访存依赖：$e_i \prec_d e_j$ 当且仅当 $tsc(e_i) < tsc(e_j)$。锁的互斥语义保证了 $e_i \prec_{tot} e_j$ 当且仅当 $tsc(e_i) < tsc(e_j)$，因此如此记录的访存依赖是一致的。

然而，上述访存依赖获取算法串行化了对共享内存的访问，与并发程序利用多处理器中并行计算资源的设计初衷背道而驰。对于内存密集型的程序，这样的修改可能带来数千倍的运行时开销。访存依赖只要求确定 $e_i \asymp e_j$ 事件对之间的顺序，而 $e_i \asymp e_j$ 的必要条件之一是 $e_i.x = e_j.x$，因此可以给不同的 x 独立分配锁和时间戳，如图 2.4 所示。对于 $e_i \asymp e_j$，一定有 $e_i.x = e_j.x = x$。因此访

$$
\begin{array}{cc}
\texttt{lock}(\ell_{k'}) & \texttt{lock}(\ell_k) \\
e: \underline{x_k^t = R(x_{k'})} & e: \underline{W(x_k, x_{k'}^t)} \\
tsc_{k'} \leftarrow tsc_{k'} + 1 & tsc_k \leftarrow tsc_k + 1 \\
\texttt{log}(\langle e, tsc_{k'} \rangle) & \texttt{log}(\langle e, tsc_k \rangle) \\
\texttt{unlock}(\ell_{k'}) & \texttt{unlock}(\ell_k)
\end{array}
$$

图 2.4 对不同的变量使用不同的锁和日志，实现更高效的访存依赖获取。

存依赖 $e_i \prec_d e_j$ 当且仅当 $tsc_x(e_i) < tsc_x(e_j)$，易见这一方法也能获取一致的访存依赖，并在研究工作中广泛使用 [69, 112]。

不同类型的动态分析技术有不同的应用场景，因此对访存依赖获取算法的要求也不同。尽管所有类型的应用都希望用尽可能少的时间开销和最少的日志记录下满足应用需求的访存依赖，但根据应用场景不同，具体动态分析技术通常必须在时间开销、日志数量、准确性、即时性等技术指标之间进行权衡。例如在测试环境中，允许付出数十甚至数百倍的开销获取一致 (甚至准确) 的访存依赖从而找到软件中存在的缺陷，但在生产环境中，这样的开销就是难以接受的，哪怕牺牲准确性、即时性，也必须在较少的时间开销内得到可以实现动态分析的访存依赖。

因此，访存依赖获取问题的问题空间本质是设计对程序或运行时系统做怎样的修改、在运行时记录何种信息、在程序执行时或执行后如何推导依赖关系，以满足动态分析技术对各方面性能的要求。图 2.3、图 2.4 中的算法仅在很抽象的层面描述了访存依赖获取的一类基本方法。在实际实现访存依赖的获取时，可以使用不同的机制实现 `lock`/`unlock` 的互斥 (如修改硬件、修改操作系统利用硬件的存储保护机制、程序插装等) 或将多个变量 x 指派到同一个互斥锁等。更进一步的，我们允许放松记录的 $\prec_d$ 的准确性乃至一致性。在此假设下，我们不仅可以直接记录事件之间的先后关系，还可以通过记录间接信息 (如每次共享内存读/写的地址和数值)，对访存依赖予以推导，从而避免在对并发系统的修改中引入额外的同步操作 (共享内存读写的地址和数值可以记录在每个线程本地)。

2.2 共享内存访问的局部性理论

为摆脱现有研究工作“需求驱动 $\rightarrow$ 经验性质 $\rightarrow$ 实验验证”思路的局限，我们提出共享内存访问的局部性理论，用于建模并发程序共享内存的访问模式。只有系统化、量化地理解真实程序是“怎样”访问共享内存的，才能系统地设计出高效的访存依赖获取技术，以实现高效的并发程序动态分析。

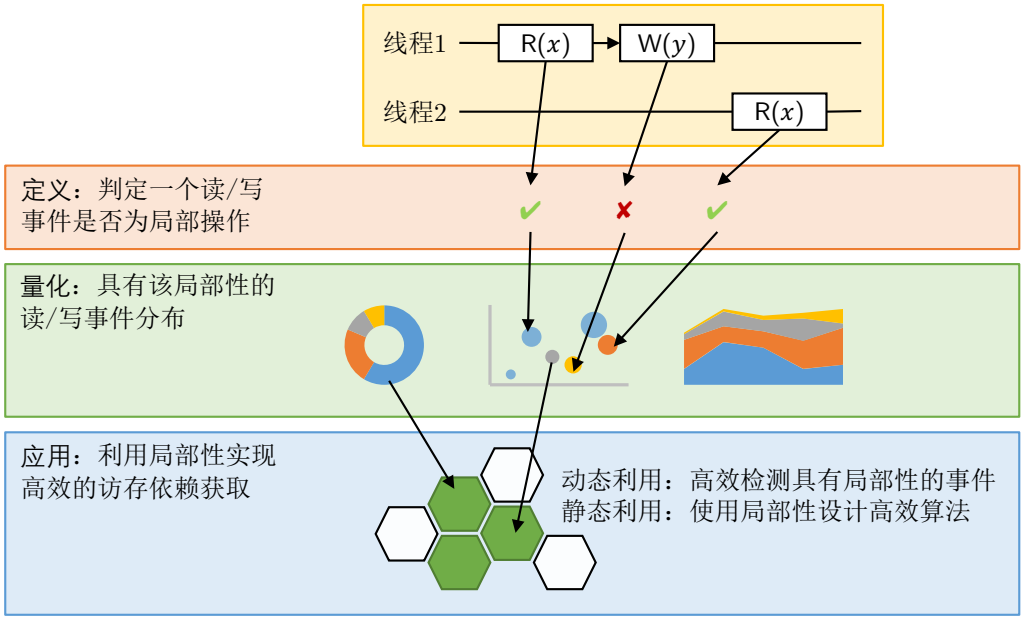

图 2.5 局部性理论的三要素：定义、量化、应用。

共享内存访问的局部性理论提出对“局部性”这一非形式化概念进行系统化、量化研究的框架，并提出现实并发程序具有的线程局部性、空间-线程局部性和同步局部性，讨论局部性与访存依赖获取之间的关系。

2.2.1 局部性理论概述

并发程序共享内存访问的局部性理论用于刻画实际并发程序共享内存访问所具有的特点和模式。局部性自从在 1970 年代提出 [97]，即成为计算机系统设计、实现和优化的核心思想 [44]。“局部性”描述了计算机程序在运行时访问资源的不均匀性 (non-uniformity) 特征：在一段时间内，计算机程序对资源的访问通常只局限在一个较小的工作集 (working set) 中。

共享内存作为并发程序中最常见的共享资源，其访问模式同样具有局部性，且已在并发系统的设计及并发程序的动态分析中得到广泛应用。Stride [144] 利用线程局部性实现高效的执行重放；Octet [28] 利用线程局部性和读/写操作之间的不对称性实现高效的访存依赖获取；合理地利用空间局部性能降低并发程序动态分析的开销 [80]。

尽管局部性在多个领域都得到了广泛应用，程序所具有的“局部性”本身却尚未得到有有针对性地、系统化地研究。现有技术通常基于“试错”型的研究：为了实现具有某一特征的并发系统或动态分析技术而提出非形式化定义的局部性 (或仅考虑局部性的某些特例情况)，再根据对该局部性的直观认识设计

相应的算法，最后在实验中加以验证。Octet [28] 即为此类研究的典型。我们认为，此类试错型的研究模式虽能有效利用并发程序的局部性，但若能对并发程序的运行时特性进行合理的理论建模，从更高的抽象层次上对局部性进行解读，则可以更系统、更深入地理解访存依赖所具有的性质，从而推动访存依赖获取技术和基于访存依赖的动态分析技术的发展。

本文提出并发程序共享内存访问的局部性理论，认为实际并发程序的共享内存访问具有以下三种局部性：

1. *线程局部性*：一个共享变量在一段时间内，要么被一个线程独占访问，要么在多个线程之间以只读的形式共享。
2. *空间-线程局部性*：一段地址空间上连续的共享变量在一段时间内，要么被一个线程独占访问，要么在多个线程之间以只读的形式共享。空间-线程局部性是线程局部性在连续地址空间变量上的扩展。
3. *同步局部性*：真实并发程序中数据竞争罕见，表现为在一段时间内，一个变量在线程间的数据依赖与某一同步变量 (如互斥锁、条件变量) 在线程间的数据依赖保持一致。

为了深入研究这些局部性，我们首先提出并发程序共享内存访问局部性的系统化研究框架，如图 2.5 所示。对于一个特定的局部性 (本文研究上述三种局部性，但此框架可用于未来新的局部性研究)，局部性理论研究这一局部性的三个要素：

1. 局部性的形式化定义，即在给定执行轨迹的基础上，如何判定其中的共享内存访问事件是否为局部事件。
2. 局部性存在的量化证据。由于局部性描述了一般程序所具有的特征 (换言之，可以人为构造不存在局部性的程序)，因此局部性也必须在真实程序的执行轨迹上得到验证。
3. 局部性在访存依赖获取技术上的应用。通过区别对待局部和非局部事件可以实现对访存依赖获取技术的优化。局部性本身也可作为执行轨迹的静态性质使用。

2.2.2 局部性的三要素

2.2.2.1 局部性的定义

为了研究并发程序所具有的某一具体局部性 (如线程局部性)，研究框架首先须对它进行形式化定义。局部性定义在第 2.1 节对并发系统、执行轨迹的形

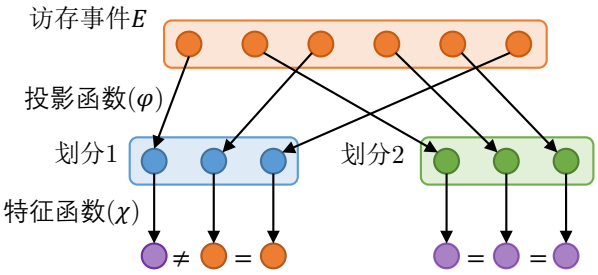

图 2.6 定义局部性的投影函数 φ 和特征函数 χ。

式化模型上。令并发执行的轨迹

$$\tau = \left[\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} \sigma_n\right],$$

其中 $\sigma_i = \langle V_i, PC_i, E_i\rangle$，其访存事件集合 $E = E_n$，τ 定义了 E 事件的全序 $\prec_{tot}$。

局部性原理反应了现实中并发程序执行轨迹的特性：共享内存访问在一段连续时间中的访问具有高度的不均匀性 (non-uniformity)。不同的局部性刻画了不同类型的不均匀性。

如上直观描述必须被形式化：给定执行轨迹中的任意事件，须判定其是否满足这一局部性。给定执行轨迹 τ，被研究的局部性 (如线程局部性) 将能唯一确定二元组 $\langle\Phi, \chi\rangle$，其中：

1. 投影函数族 Φ 为函数的集合，对于任意的 $\varphi \in \Phi$ 和 $e \in E$，$\varphi(e)$ 将其映射到 e 所具有的属性，用以实现对 E 中事件的划分。
2. 特征函数 χ。对于 $e \in E$，特征函数 $\chi(e)$ 给出 e 在某方面的属性，$\chi(e)$ 的值用于确定事件 e 在执行轨迹 τ 中是否具有局部性。

φ 和 χ 的直观解释如图 2.6 所示，$\langle\varphi, \chi\rangle$ 组合为局部性准则，用于确定任意事件 $e \in E$ 是否具有该局部性：

给定某个投影函数 $\varphi \in \Phi$，其指定了 E 的一个划分：$e_i, e_j \in E$ 属于同一划分当且仅当 $\varphi(e_i) = \varphi(e_j)$。换言之，$\varphi$ 将 E 中的所有事件投影到若干子集 $E_v = \{e \in E \mid \varphi(e) = v\}$，其中 E_v 包含 $\varphi(e) = v$ 的事件。直观来说，φ 将所有事件分解成若干事件的子集合，且每一个子集中的事件都有较强的关联性，从而倾向于具有访问的不均匀性。

接下来，特征函数 $\chi(e)$ 将用于确定一个事件是否是“局部”事件。令 $E_v = \{e_1, e_2, \dots, e_m\}$，不妨设这些事件按照实际发生的顺序编号，即 $e_1 \prec_{tot} e_2 \prec_{tot} \dots \prec_{tot} e_m$。对于 $1 < i \le m$，定义事件 e_i 为“局部”事件 (即具有 $\langle\varphi, \chi\rangle$ 对应的局部性)，当且仅当 $\chi(e_{i-1}) = \chi(e_i)$。

并发程序具有局部性可定义为存在 $\varphi \in \Phi$，在 $\langle\varphi, \chi\rangle$ 定义的局部性准则下，相当大部分事件 $e_i \in E_v = \{e_1, e_2, \dots, e_m\}$ 为局部事件，即满足 $\chi(e_{i-1}) = \chi(e_i)$

$(1 < i \leq m)$。这一形式化定义消除了对局部性除“相当大部分”的量化描述之外所有含有歧义的描述。由于局部性本身就是为现实程序建立的模型，并不存在对所有程序皆成立的局部性 (总能够写出特殊的程序违背局部性原理)，因此这一量化也必须针对实际程序完成，其方法如下节所述。

2.2.2.2 局部性的量化

对于局部性准则 $\langle \varphi, \chi \rangle$，令 $E_v = \{e \in E \mid \varphi(e) = v\} = \{e_1, e_2, \ldots, e_m\}$，且 $e_1 \prec_{tot} e_2 \prec_{tot} \ldots \prec_{tot} e_m$。$e_i$ $(1 < i \leq m)$ 具有局部性当且仅当 $\chi(e_{i-1}) = \chi(e_i)$。定义 E_v 中局部事件的数量

$$LC(E_v) = \sum_{i=2}^{m} [\chi(e_{i-1}) = \chi(e_i)],$$

其中 $[cond]$ 为指示函数 (indicator function)，当 $cond$ 为真时 $[cond] = 1$，否则 $[cond] = 0$。类似地，我们定义

$$\mathcal{L}(E) = \frac{1}{|E|} \max_{\varphi \in \Phi} \sum_{v} LC(E_v)$$

给出了所有投影函数具有局部性事件数量的最大值，即在该局部性定义下，具有局部性事件比例的上限。给定执行轨迹 τ，对于每一局部性均可计算其对应的具有局部性事件数量的上限。

此外，针对每一局部性，通过实证研究调查何种特征的 v 具有较大的 $\frac{LC(E_v)}{|E_v|}$ 比值、满足 $\chi(e_{i-1}) = \chi(e_i)$ 的事件具有何种特征，均能增进我们对并发程序访问共享内存模式的理解。这方面的工作将是我们未来工作的重点。

2.2.2.3 局部性的应用

对于一个定义明确且被量化研究证实存在的局部性，其应用主要关注于如何利用该局部性以实现高效的访存依赖获取技术。局部性的动态应用主要关注于如何识别出具有特殊性质的“局部”事件，通过简化对局部事件的处理而达到提高效率的目的 (例如，即便不记录具有线程局部性的事件，也不会影响访存依赖的一致性)。局部性的静态应用则将局部性看作是并发执行轨迹固有的性质，并利用此性质实现算法的优化。

形式化地说，局部性的*动态应用*是对运行系统的修改和一组算法，使得给定任意执行轨迹 τ 及其 $\langle \varphi, \chi \rangle$，能在运行时高效地判断当前发生的访存事件是否具有局部性。具体来说，在运行时维护一组 $\langle \varphi, \chi_L \rangle$，且满足：

1. 对于任意事件 e，在运行时能高效地计算 $\varphi(e)$ 和 $\chi_L(e)$。

2. χ_L 是对特征函数 χ 的正确 (sound) 近似，即不会将实际不具有局部性的事件认为具有局部性。即对于 $E_v = \{e_1, e_2, \ldots, e_m\}$，若 $\chi_L(e_{i-1}) = \chi_L(e_i)$，则一定有 $\chi(e_{i-1}) = \chi(e_i)$。

由于一次并发程序执行中共享内存访问数量众多，若无法在运行时高效地计算 $\varphi(e)$ 和 $\chi_L(e)$，即便能正确识别出具有局部性的事件，其带来的代价也可能是得不偿失的。另一方面，若不正确的 χ_L 将实际不具有局部性的事件认为具有局部性，将会导致应用该局部性的算法遗漏实际应当检查的事件而导致错误的发生。因此，上述两点是动态应用局部性的必要条件。

局部性的静态应用将局部性作为并发程序轨迹的固有属性加以研究。局部性理论表明，并发程序的共享内存访问具有不均匀的特性，即 E 中的事件及其发生的时序关系 $\prec_{tot}$ 并不服从均匀随机的分布。具体的局部性会量化这一不均匀的方式和程度，即确定 E 和 $\prec_{tot}$ 的一些特征。对执行轨迹的这些特征加以利用可实现高效的并发程序动态分析算法。

2.2.3 线程局部性、空间-线程局部性与同步局部性

2.2.3.1 线程局部性

线程局部性反映并发程序的如下性质：一个共享变量在一段时间内，要么被一个线程独占访问，要么在多个线程之间以只读的形式共享。换言之，在一段时间内，在一个线程内对共享变量的访问，将具有相同的线程间数据依赖。根据这一性质，我们容易给出线程局部性的定义：

投影函数族 $\Phi_{\mathrm{TL}} = \{\varphi_{\mathrm{TL}}\}$ 只包含一个投影函数 $\varphi_{\mathrm{TL}}(e) = \langle e.t, e.x\rangle$。给定执行轨迹 τ 中的事件集合 E，φ_{TL} 制导的划分将得到若干独立的集合 $E_{\langle t,x\rangle} = \{e \in E \mid e.t = t \wedge e.x = x\}$，其中包含所有线程 t 对变量 x 的访存事件。

特征函数 $\chi_{\mathrm{TL}}(e)$ 定义为发生在 e 之前、发生在另一个线程且与 e 数据流相关的事件的集合。定义 e 的“上一次写者”集合

$$E_w = \{e_w \in E \mid (e_w.rw = w) \wedge (e.x = e_w.x) \wedge (e_w \prec_{tot} e) \wedge \\ \neg\exists e_{w'} \in E.\, ((e_{w'}.rw = w) \wedge (x = e_{w'}.x) \wedge (e_w \prec_{tot} e_{w'} \prec_{tot} e))\}。$$

对于读事件 $e_r = R(t, x, v)$，由于线程间的读操作不会引起与之相关的数据依赖，令 $E_t = \{e \in E \mid e_r.t = e.t\}$，定义

$$\chi_{\mathrm{TL}}(e_r) = E_w \setminus E_t。$$

对于写事件 $e_w = W(t, x, v)$，取出从 E_w 中的事件到 e_w 之间的读取变量 x 的所有事件集合

$$E_r = \{e_r \in E \mid (e_r.rw = r) \wedge (e_w.x = e_r.x) \wedge \\ (\forall e'_w \in E_w.\, e'_w \prec_{tot} e_r \prec_{tot} e_w)\},$$

当 E_r 为空时，e 的数据依赖来自 E_w，否则 e 的数据依赖来自 E_r，据此定义

$$\chi_{\mathrm{TL}}(e) = \begin{cases} E_w \setminus E_t, & E_r = \varnothing, \\ E_r \setminus E_t, & E_r \neq \varnothing。 \end{cases}$$

线程局部性表明，对于一组 $E_{\langle t,x\rangle} = \{e_1, e_2, \ldots, e_m\}$，对于大部分的事件 e_i 均满足 $\chi_{\mathrm{TL}}(e_i) = \chi_{\mathrm{TL}}(e_{i-1})$，即 e_i 的数据依赖要么是线程本地的 ($\chi_{\mathrm{TL}}(e)$ 排除了来自本地的数据依赖)，要么与 e_{i-1} 的数据依赖相同。通常，具有线程局部性的事件可以在访存依赖获取技术中被忽略，因此线程局部性的检测一直以来都是访存依赖获取技术的研究热点 [28, 79]。

2.2.3.2 空间-线程局部性

空间-线程局部性是线程局部性在连续地址空间上的自然扩展，反映为在一段时间内，一段连续的内存通常要么被一个线程独占访问，要么在线程间以只读的方式共享。

设并发程序的地址空间为 $[0, M)$，其中 M 为地址空间的最大值。令 $\mathcal{G}$ 为对地址空间所有可能的区间划分，其中任意 $G \in \mathcal{G}$ 将地址空间分为若干组，即 G 能够写成若干首尾相接区间的并 $G = [l_1, r_1) \cup [l_2, r_2) \cup [l_m, r_m)$，满足 $l_1 = 0$, $r_m = M$ 且 $\forall 1 \leq i < m.\, (r_i = l_{i+1})$。给定共享内存变量 x，令 $G(x)$ 为唯一包含 x 的区间，即唯一的 $[l_i, r_i)$ 满足 $x \in [l_i, r_i)$。空间-线程局部性定义如下：

投影函数族 $\Phi_{\mathrm{STL}} = \{\varphi_G \mid G \in \mathcal{G}\}$，即允许对地址空间进行任意划分。$\varphi_G(e) = \langle e.t, G(e.x)\rangle$。$\varphi_G(e)$ 的形式与线程局部性类似。线程局部性考虑一个线程访问同一变量的事件集合，类似的，空间-线程局部性考虑一个线程访问一个变量组 (即一个区间) 的事件集合。

特征函数 χ_{STL} 的定义与线程局部性类似。线程局部性考虑每一变量 x 的线程间数据依赖，空间局部性则将变量组的全部变量视作一个整体，并考虑变量组的线程间数据依赖。类似地，定义

$$E_w = \{e_w \in E \mid (e_w.rw = w) \wedge (G(e.x) = G(e_w.x)) \wedge (e_w \prec_{tot} e) \wedge \\ \neg \exists e_{w'} \in E.\, ((e_{w'}.rw = w) \wedge G(e.x) = G(e_{w'}.x) \wedge (e_w \prec_{tot} e_{w'} \prec_{tot} e))\},$$

$$E_r = \{e_r \in E \mid (e_r.rw = r) \wedge (G(e_w.x) = G(e_r.x)) \wedge$$
$$(\forall e'_w \in E_w.\, e'_w \prec_{tot} e_r \prec_{tot} e_w)\}。$$

对于读事件 $e_r = R(t, x, v)$，

$$\chi_{\mathrm{STL}}(e) = E_w \setminus E_t。$$

对于写事件 $e_w = W(t, x, v)$，

$$\chi_{\mathrm{STL}}(e) = \begin{cases} E_w \setminus E_t, & E_r = \varnothing, \\ E_r \setminus E_t, & E_r \neq \varnothing。 \end{cases}$$

2.2.3.3 同步局部性

同步局部性反映了真实并发程序中数据竞争罕见的特性。不同于线程局部性和空间-线程局部性考虑一段时间内一个共享内存变量倾向于保持较为恒定的数据依赖，同步局部性的主要观察来自于现实程序中开发者通常会用同步机制保护共享内存变量而使其免于数据竞争的现象 [25, 109]：对于一个共享内存变量 x，在程序执行的过程中，通常有与之相伴的同步变量 (如锁、信号量、管程)，且 x 的线程间数据流能够由这一同步变量推导。同步局部性定义如下：

投影函数族 $\Phi_{\mathrm{SynL}} = \{\varphi_{\mathrm{SynL}}\}$，其中 $\varphi_{\mathrm{SynL}}(e) = e.x$。$\varphi_{\mathrm{SynL}}$ 制导的划分将执行轨迹中所有事件 $e \in E$ 按照其访问的变量分组。

特征函数 $\chi_{\mathrm{SynL}}(e)$ 是由程序语义决定的同步变量 x_s，代表 e 事件所对应的同步变量，表明在当前并发程序执行轨迹中，x_s 在一段时间内消除了对 $e.x$ 访问的数据竞争。

形式化地说，对于划分后的一组事件 $E_x = \{e_1, e_2, \ldots, e_m\}$ ($e_1 \prec_{tot} e_2 \prec_{tot} \ldots \prec_{tot} e_m$)，令 τ 中对 x_s 的访问事件 (将同步变量操作看作共享内存的写操作) 为 $E' = \{e'_1, e'_2, \ldots, e'_k\}$，定义同步变量对应的 happens-before 关系

$$\prec_{syn} = tr(\{(e'_i, e'_{i+1}) \mid 1 \leq i < k\} \cup \prec_{po}),$$

e_{i+1} 能映射到 x_s 仅当 $e_i \prec_{syn} e_{i+1}$。若 e_i 不存在其对应的同步变量，则令 $\chi_{\mathrm{SynL}}(e_i) = i$ 以确保其不具有同步局部性。

虽然 e 所对应的同步变量 (即 $\chi_{\mathrm{SynL}}(e)$) 与程序的语义相关而无法自动判定，但我们仍可在量化研究时对这一数值予以近似，例如为每个变量 x 求解最少数量的同步变量，使得能够推导出 x 的所有线程间数据依赖。这给出了实际具有同步局部性事件的一个上界。

表 2.2 用于局部性实证量化研究的实验对象。

实验程序	任务描述	代码行数	实验设定
ocean	海洋模拟	9.1K	16 线程，1026×1026 网格
water	分子动力学模拟	3.6K	16 线程，10 步，10^3 个分子
fft	快速傅立叶变换	1.4K	16 线程，2^{24} 数据点
radix	基数排序	1.9K	16 线程，2^{24} 元素
fluid	流体动力学模拟	1.2K	16 线程，10 帧，100KB 输入
qsort	快速排序	0.9K	16 线程，2^{24} 元素
x264	视频编码压缩	37K	16 线程，128 帧，640×360 分辨率

2.2.3.4 局部性的实证量化研究

在给出局部性的形式化定义后，我们即可以对现实并发程序进行插装得到其执行轨迹，并根据定义判定执行轨迹中的每一事件是否具有局部性，从而实现局部性的量化研究。我们从 PARSEC [24]、SPLASH-2 [128] 中选取 7 个有代表性的并发基准程序 (并发程序的详细信息在表 2.2 中描述)，完成如下实证研究：

1. 对于线程局部性，直接对执行轨迹中的事件 E 按照 φ_{TL} 的划分结果和 χ_{TL} 的判定结果对 $\mathcal{L}(E)$ 进行统计。
2. 对于空间局部性，$\Phi_{\mathrm{STL}} = \{\varphi_G\}$ 允许对地址空间进行任意划分。我们考虑其中一类具有代表性的情况：将地址空间分解为长度为 2^k $(k \in \mathbb{N})$ 的首尾相接的子区间的划分，即给定 k，其划分 $G_k = [l_1, r_1) \cup [l_2, r_2) \cup [l_m, r_m)$，其中 $l_i = (i-1)2^k$, $r_i = i \cdot 2^k$, $m = \lceil M/2^k \rceil$，即我们使用投影函数族 $\Phi = \{\varphi_{G_k} \mid 0 \le k \le \log_2 M\}$ 对空间局部性进行量化统计 (计算 $\mathcal{L}(E)$)。由于 $\Phi \subset \Phi_{\mathrm{STL}}$，实际程序中潜在的空间局部性将高于我们使用 Φ 进行统计的结果。
3. 同步局部性与线程局部性类似，Φ_{SynL} 中只有一个投影函数 φ_{SynL}，故其只需为每个事件 e 计算 $\chi_{\mathrm{SynL}}(e)$ 即可获得统计结果。因共享内存访问具有空间-线程局部性，我们将连续 $2^6 = 64$ 字节的变量分为一组进行同步局部性的实证研究。这一数值等于硬件一级缓存线的宽度，在程序不存在假共享缺陷的前提下，这一分组方法能有效降低元数据维护的开销。

   对于每一变量 x 及其访问事件 $E_x = \{e_1, e_2, \ldots, e_m\}$，我们使用动态规划算法求解出 $\chi_{\mathrm{SynL}}(e)$ 的近似值。用 f_i 表示推导出 $\{e_1, \ldots e_i\}$ 数据依赖所需的最少同步变量更换次数，则 $f_0 = 0$，

$$f_i = \min_{0 \le j < i} \{f_j + c_{j+1,i}\}。$$

表 2.3 线程、空间-线程和同步局部性的量化对比。

实验程序	局部性量化 $\mathcal{L}(E)$			
	线程	空间-线程 ($k=6$)	空间-线程 ($k\in\mathbb{N}$)	同步 ($k=6$)
ocean	96.50%	99.73%	>99.99%	99.80%
water	99.77%	99.95%	99.99%	99.99%
fft	80.71%	98.27%	>99.99%	99.32%
radix	72.19%	97.30%	99.70%	98.61%
fluid	99.44%	99.91%	99.91%	99.99%
qsort	76.47%	99.42%	99.79%	99.56%
x264	96.09%	99.90%	>99.99%	99.99%

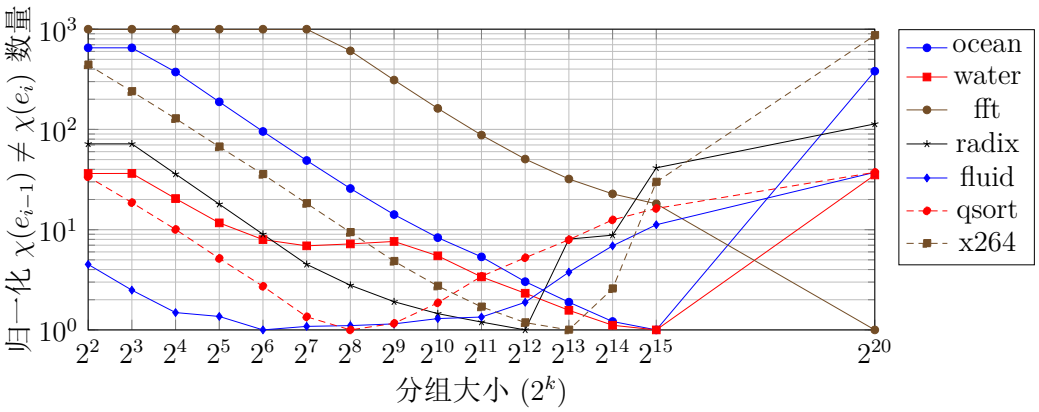

图 2.7 区间大小 k 对具有空间-线程局部性共享内存访问数量的影响。

其中代价函数 c 定义如下：若存在同步变量 x_s 能推导出 $\{e_i,\dots e_j\}$ 的数据依赖，则 $c_{i,j}=1$，否则 $c_{i,j}=\infty$。$\chi_{\mathrm{SynL}}(e)$ 定义为 f_m 取最优解时 e 所对应的同步变量，并据此计算出同步局部性的量化统计 $\mathcal{L}(E)$。

我们使用 LLVM 工具对其共享内存访问进行插装并获得所有对静态数据区和堆区的内存访问，以实现执行轨迹的获取。所有实验结果均在一台有 4 个 6 核心 Intel Xeon X7460 处理器 (共 24 个处理器核心) 和 64GB 内存的服务器上运行得到。每个实验对象均运行 10 次，并取统计数据的平均值予以展示。

对线程、空间-线程、同步局部性的实证量化研究结果如表 2.3 和图 2.7 所示。表 2.3 展示了线程、空间-线程和同步局部性的量化研究结果 (对空间-线程局部性，我们展示 $k=6$ 的情形和对任意 $k\in\mathbb{N}$ 的最优值)。图 2.7 包含了划分区间大小 k 对具有空间-线程局部性事件数量的影响。根据 $\mathcal{L}(E)$ 的统计结果，可总结得出以下结论：

1. 现实中并发程序对共享内存的访问确实具有相当程度的局部性，即便考虑 $\mathcal{L}(E)$ 数量最小的一列 (线程局部性)，也有 72.19-99.77% (中位数 96.09%) 的共享内存访问是“局部”事件。
2. 空间-线程局部性具有优秀的应用潜力。假如我们能在程序运行前就知道其局部性最优的 k，即便使用简单的静态空间分组 (区间长度为 2^k)，具有空间局部性的事件仍然达到了 99.70-99.99% (中位数 99.99%)，即如能充分利用空间-线程局部性，平均意义上仅有万分之一的共享内存访问是“非局部”事件。
3. 对于同步局部性与空间-线程局部性之间正交属性的同时利用可展现出更优秀的应用潜力。在使用与空间-线程局部性同等长度的区间分组 ($k = 6$) 时，具有同步局部性的事件数量 (98.61-99.99%，中位数 99.80%) 显著多于空间-线程局部性的事件数量 (97.30-99.95% 中位数 99.73%)。

我们将在未来工作中对并发程序共享内存访问的局部性进行更深入的量化研究，以实现更高效的访存依赖获取技术。

2.2.3.5 局部性与访存依赖获取

局部性，作为现实中并发程序共享内存访问的固有特性，可以用于指导高效访存依赖获取技术。易见，获取访存依赖的算法在不同的访问模式下有不同的表现。例如若整个程序中只有一个共享变量，图 2.3 与图 2.4 中的算法就是等价的。但在实际系统中，由于大量共享变量的存在，图 2.4 的算法即因使用细粒度锁 (对不同共享变量使用不同的锁和时间戳)，性能会显著优于图 2.3 [69]。因此，利用现实并发程序访问共享内存的特性，有助于实现更高效的访存依赖获取技术，进而实现高效的并发程序动态分析技术。

局部性的动态应用可用于实现高效的访存依赖获取技术：

1. 如果事件 $e \in E$ 具有线程局部性或空间-线程局部性，则在记录所有不具有局部性事件的数据依赖关系的前提下，省略对局部事件的记录，也能推导出一致的访存依赖 [28]。由于绝大部分内存访问都有线程局部性或空间-线程局部性，因而对其高效的检测亦为减少访存依赖获取事件开销和日志数量的重要手段。这两类局部性也是目前研究工作关注的焦点 [80]。
2. 如果变量 x 在一段时间内具有同步局部性，则仅记录同步事件发生的顺序，就能够推导出这段时间内 x 的数据依赖关系。多份研究工作 [80, 109] 均表明，并发程序中的同步事件数量远少于访存事件，且记录同步事件发生的顺序不会引入额外的同步。若能有效地利用变量的同步局部性，将能在获得一致访存依赖的前提下大幅降低其运行时开销。

对局部性进行动态利用的典型技术是访存依赖追踪技术 Octet [28]，利用局部性提高访存依赖获取的效率。其在运行时利用乐观锁快速检测内存访问事件是否具有线程局部性，并在具有线程局部性的事件上忽略访存依赖的记录。此项技术能高效实现如执行重放 [27]、事务内存 [137]、数据竞争避免 [117] 等并发程序的动态分析技术。

局部性的静态应用可以指导访存依赖的合成。有文献 [57] 证明，如果只在线程本地记录读写事件的数值，得到任一满足顺序一致性的访存依赖是 NP-完全问题。但由于现实程序同时具有线程局部性和同步局部性，在现实程序的执行轨迹中，大部分事件的时序关系是能够确定的，因而约束求解器通常可以在较短的时间内找到一组可行解，以实现现实程序的执行重放 [87]。

利用局部性实现的高效访存依赖获取技术将在第 3 章提出的技术框架中与其他访存依赖获取技术共同探讨。

2.3 小结

在本章中，我们首先对本文的研究对象访存依赖及其相关概念 (如并发系统、执行轨迹等) 进行了形式化定义，探讨了访存依赖与并发程序动态分析技术之间的关系。这一模型为我们后续的技术框架 (第 3 章) 对来自多个研究领域的研究工作进行系统化地总结奠定了基础。其次，本章还提出了并发程序共享内存访问的局部性理论。针对现有技术对“局部性”这一概念缺乏系统性认识的现状，我们提出了局部性所具有的三要素：定义、量化、应用，并基于这一框架总结了已被广泛应用的线程局部性和空间-线程局部性，在此基础上又提出了全新的同步局部性，其量化结果显示出巨大的应用潜力。此外，对局部性的应用是实现高效访存依赖获取乃至并发程序动态分析的重要手段。利用局部性实现高效访存依赖获取是本文主要技术贡献 [78, 79, 80] 的基石。

第三章　技术框架与相关工作综述

近年来，来自体系结构、计算机系统、程序设计语言和软件工程等多个领域的研究者都对并发程序的动态分析技术进行了探讨。然而，目前这些工作并非围绕“访存依赖”这一核心概念展开，缺乏统一的技术框架以系统性地总结和探讨现有技术的优势与不足，无法系统化地指导新的访存依赖相关技术的发展。

第 2 章提出的理论框架明确了访存依赖及其获取问题，并提出了并发程序共享内存访问的局部性理论。在理论框架的指导下，我们已经具备了提出具有指导意义的技术框架的条件，其帮助我们系统性地理解了现有工作如何获取访存依赖、在哪些方面作出了权衡、实现了怎样的动态分析、解决了何种并发程序质量保障的挑战，从而指导我们更好地认识现有研究工作的不足和空白。

本章提出访存依赖相关研究的技术框架，该框架由三要素构成：评价指标、实现技术和技术应用。这一技术框架成功地容纳了来自多个研究领域对访存依赖获取和应用的探讨，从更高层次上阐明了现有技术在各个方面作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。第 3.1 节首先讨论贯穿技术框架的访存依赖获取技术评价指标，然后对技术框架进行概述。第 3.2 节对获取访存依赖的两类技术：在线追踪 (第 3.2.1 节)、离线合成 (第 3.2.2 节) 中的关键问题和解决手段进行综述；第 3.3 节探讨两类并发程序的动态分析技术——轨迹分析 (第 3.3.1 节) 和并发控制 (第 3.3.2 节) 与访存依赖之间的关系。最后，第 3.4 节总结技术框架并讨论其中的研究契机。

3.1　技术框架概述

针对访存依赖相关研究，我们提出包含三要素的技术框架：

1. 四个评价指标：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应实际程序执行的访存顺序；高效性要求其获取开销尽可能小；简化性要求获得的日志尽可能精简。
2. 两类现有访存依赖获取技术：在线追踪和离线合成。在线追踪技术在程序运行时即时捕获访存依赖，而离线合成技术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。

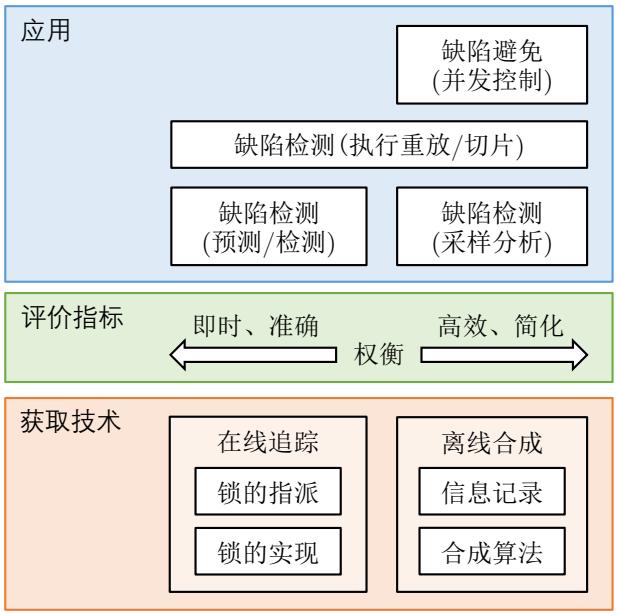

图 3.1 技术框架的三要素：评价指标、技术、应用之间的关系。

3. 两类访存依赖的应用：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，以在运行时避免并发缺陷。

技术框架总结如图 3.1 所示：获取访存依赖的技术在评价指标的指导下，对即时、准确、高效、简化四个特性进行权衡，并最终实现为高效的并发程序动态分析技术。

3.1.1 访存依赖获取技术的评价指标

我们首先对四个评价指标予以定义：

1. 即时性反映是否在访存事件 e 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 [120]) 是至关重要的，而轨迹分析类的应用 (如测试、调试 [84]) 则通常对即时性没有要求。
2. 准确性反映获取的 $\prec_d$ 是否正确反映执行中的实际数据依赖。获取完全准确访存依赖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 [55]) 中又是不可避免的；对于调试、执行重放类的应用，只要求获取一致的访存依赖 (即对于一对冲突的访存事件 $e_i \asymp e_j$, $e_i \prec_d e_j$ 当且仅当 $e_i \prec_{tot} e_j$)，此时按 $\prec_d$ 执行程序后每一线程的执行路径与 $\prec_{tot}$ 相同 (从而足以诊断轨迹中的缺陷)，再次执行即可推导出准确的访存依赖；对性能极为敏感的应用 [134] 甚至允许与 $\prec_{tot}$ 不一致。

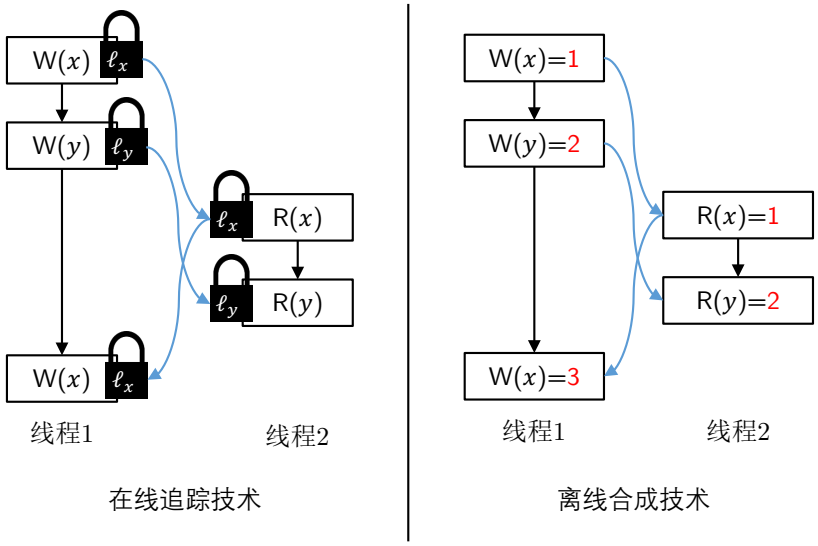

图 3.2 访存依赖获取的两类技术：在线追踪、离线合成。

3. 高效性反映获取访存依赖时由于程序插装或修改并发系统导致的程序运行性能降低的程度。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容忍运行时开销，随着软件系统规模的增长、执行轨迹长度的增大，运行时开销也会逐渐成为动态分析技术的瓶颈 [127]。
4. 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。因访存依赖只需确定冲突事件 $e_1 \asymp e_2$ 发生的顺序，所以并不是唯一的。即便对于同一个 $\prec_{d^*}$ 也有多种等价的 $\prec_d$ 表示。访存依赖最终为并发程序的动态分析所使用，获取访存依赖所产生的日志记录数量会直接影响动态分析的性能。若动态分析无需满足一致性的访存依赖，$\prec_d$ 和满足其需求的最简记录数量可相差数个数量级 [72]。研究者已对运行时 [105] 或运行后 [76] 简化访存依赖作出了尝试，同时也证明了获取最简的访存依赖是 NP-完全的 [76]。

尽管在理想情况下，我们希望能在访存事件发生前即获得准确或一致的访存依赖 (即时、准确)，同时不引起任何额外开销和记录 (高效、简化)，但现实中同时满足这些性质的技术是不存在的：获取即时、准确的访存依赖意味着需要更大的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而对这些评价指标加以权衡。

3.1.2 访存依赖获取的两类技术

获取访存依赖的方法是插装程序 (编译时插装 [69] 或二进制改写 [22]) 或修改运行系统 (虚拟机 [28] 或定制硬件 [17])，以在程序执行时获得与其执行轨迹

相关的记录信息。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 (图 3.2)。概言之，在线追踪技术能即时得到准确或一致的访存依赖，但通常对运行系统需要作出较大修改，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，但无法即时得到访存依赖。

访存依赖追踪技术直接在内存访问事件 e 发生前即得到 e 相关的访存依赖。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 保护所有对 x 的访问。此时记录解锁和上锁之间的依赖关系，即能获得满足一致性的访存依赖。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 [84]，且已得到广泛应用。

然而，在程序运行时增加额外的同步操作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线程间通信方式，线程访问共享内存无需任何同步操作，从而可最大程度利用多核处理器的并行计算资源。而为了获取准确的访存依赖，信息更新 (如时间戳更新) 和共享内存读写需要封装为原子操作。分布式理论已经证明，如需实现互斥，锁或与之等价的同步手段是无法避免的 [15]。因而高效、简化地获取访存依赖是在线追踪技术的最大挑战。

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及如何实现锁机制降低其运行时开销。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 [129]、按对象指派 [124] 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 [127])。锁可以借助定制硬件 [129]、存储保护 [52] 实现，也可以使用软件实现，其中具有代表性的技术是利用共享内存访问局部性实现的偏向锁 [28]。此外，静态线程逃逸分析 [37]、数据竞争分析 [125] 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 [72, 76]。由于这两类技术与访存依赖获取完全正交，本文不再赘述。

除在运行时追踪外，访存依赖的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) 后根据间接记录予以离线合成。间接获取法适用于事后分析型的应用，如调试 [136] 和轨迹预测分析 [71]。由于不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，因而间接记录也更加灵活且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 $\prec_d$ 的代价极大或无法提供准确性保障，但因其开销小，适用于生产应用场景 [136, 109]。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)，

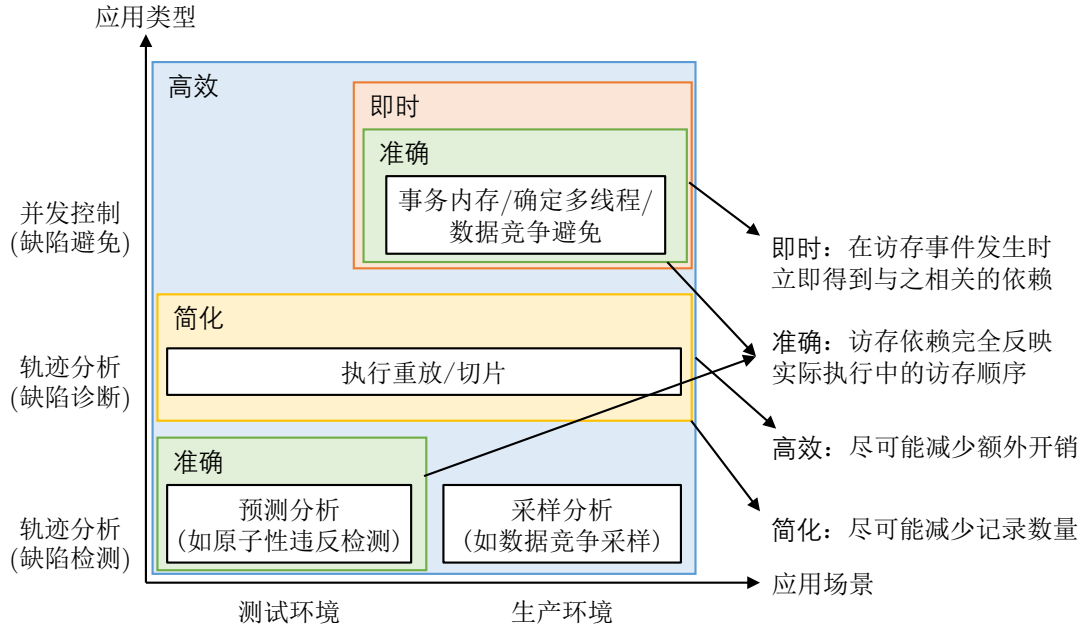

图 3.3 访存依赖应用与评价指标之间的关系。

则可以使用约束求解器合成与 $\prec_{tot}$ 在每个线程执行效果等价的 $\prec_d$，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)，推导出的 $\prec_d$ 能够提供保证重现等价执行的准确性 [144]。

3.1.3 访存依赖的两类应用

我们将访存依赖的应用分为两类：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执行进行重放、诊断或缺陷预测，亦是众多并发程序测试技术的基础；并发控制技术则在开发者指导下或全自动地对程序运行时可能出现的缺陷予以规避。访存依赖应用须根据其自身对访存依赖获取技术评价指标的需求选择或设计相应的技术 (图 3.3)。

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，以实现测试、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。因共享内存是并发程序不确定性的主要来源，获取访存依赖便成为这类技术的核心组成部分。

为实现并发程序调试相关的缺陷诊断技术，首先需要保证能够重现并发程序的执行结果，其充分条件是记录满足一致性的 $\prec_d$ 并严格按照 $\prec_{d^*} = tr(\prec_d \cup \prec_{po})$ 的顺序执行程序 [84]。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缺陷 (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 [115, 49]，使用准确的 $\prec_d$ 即可在运行时或运行后预

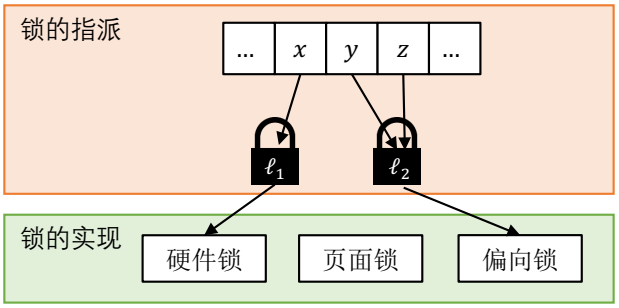

图 3.4 访存依赖追踪技术的关键问题：指派变量对应的锁及高效地实现锁机制。

测此类缺陷。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 [118, 122]，并将该模型实现为预测能力更强的并发程序缺陷检测技术 [71]。此外，执行轨迹中的访存依赖信息还能指导如原子性违反 [95] 等其他类型的缺陷发现，或为后续主动测试提供目标 [116]。

并发控制技术通过改变程序运行时的访存顺序，达到事务处理、缺陷避免、消除不确定性等功效。并发控制技术实现的关键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且要求在线获取一致的访存依赖，因而是最具挑战性的一类技术。

并发控制为开发者提供系统级的支持，以提高并发软件的质量、减少并发缺陷。软件/硬件事务内存相关研究近年来已取得一定的进展 [50, 60]，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中取得了显著的性能提升 [133]；为进一步避免并发缺陷的发生，系统还可以主动对程序行为在运行时予以修正，例如 TXRace [137] 技术将基本块构成的极大无环子图作为事务处理，从而避免了有害数据竞争导致的并发缺陷。消灭并发缺陷的终极目标是将具有不确定性的并发程序确定化，又称确定多线程 (deterministic multi-threading) 技术 [45, 40]——采取确定的规则对系统内的共享资源进行调度，从而尽可能降低不确定性带来并发错误的可能。

3.2 获取访存依赖的技术

3.2.1 在线追踪

为在运行时即时、准确地获得访存依赖，最直接的手段是为每一共享变量 x 分配其对应的锁 ℓ_x 并用 $\texttt{lock}(\ell_x)$ 和 $\texttt{unlock}(\ell_x)$ 包裹所有对 x 的读/写操作。记录每个锁上锁、解锁之间的依赖关系所得的 $\prec_d$ 保证能推导出 $\prec_{tot}$ 中的所有数据依赖。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在访存依赖中实现。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 3.4 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。

3.2.1.1 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 $\prec_d$。然而，变量锁的开销通常是较大的。若实现在硬件层面，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而大幅降低高效性。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以更好地利用现有硬件和访存局部性 (并发程序共享内存访问的线程、空间-线程和同步局部性已在第 2.2 节证实)，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，在准确性 (使用锁指派获取访存依赖仍满足一致性) 和高效性/简化性 (使用锁指派能减少元数据维护的开销和访存依赖数量) 之间进行权衡 [28]。

静态指派的一类代表方式是将地址空间上连续的变量指派为同一锁，以实现空间-线程局部性的利用。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) [13, 67, 96, 100, 101, 129]。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 [52]，或根据静态时的访问名称进行指派 [69, 144]，但也存在其他指派方式。在面向对象程序设计语言中，同一对象内变量的并发访问模式通常是类似的。文献 [124] 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 [28, 132]。

锁也可以在运行时动态指派。Ceze 等人基于硬件的锁实现 [31, 32] 将连续的内存访问打包，使用 Bloom filter 近似地确定其读/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 $\prec_d$ 而必须使用变量指派 [53, 55]，这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用于提高数据竞争检测的高效性。若使用对象指派检测对象之间的数据竞争，虽能提高高效性但存在误报风险 [124]。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，以实现高效性和准确性之间的权衡 [134]。考虑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，对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 [127]。

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，此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 [36]。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分别使用函数锁、基本块锁或循环锁，使之能够高效地获取访存依赖 [85]。

3.2.1.2 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，因此此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 [35]。但由于系统缓存一致性事件发生频繁，短时间内即产生大量依赖，因此研究工作的核心关注在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 $\prec_{tot}$ 的前提下对记录的偏序进行调整，删除冗余的依赖 [129, 130]；Ceze 等人提出记录指令块依赖以实现记录精简 [32, 31]。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，从而实现大幅的记录简化 [34]。

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，通过在有访存依赖发生时触发缺页异常而实施记录 [52]。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 [21, 92, 20]。

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。由锁保护临界区通常只需少量机器指令，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 [129] 使用二进制改写在软件层次实现 [110]，但这一做法在很多应用中都表现出巨大的运行时开销。

偏向锁技术 [113] 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 [33]。Bond 等人利用线程局部性设计乐观锁 [28]，其基本假设

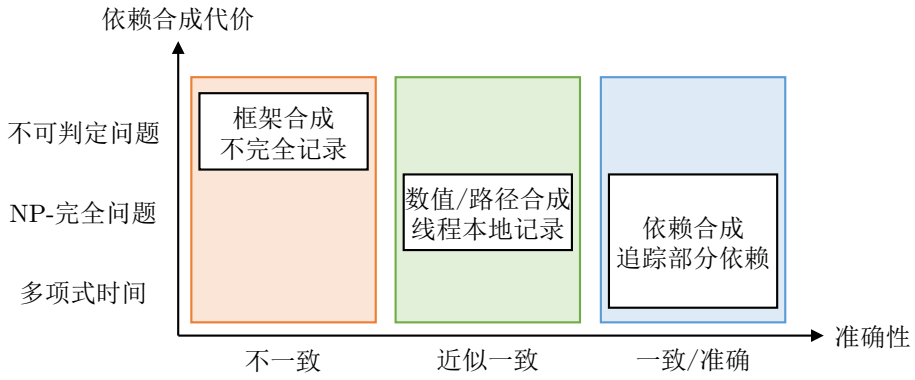

图 3.5 访存依赖合成技术的关键问题：权衡高效性/简化性与合成代价。

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，并不立即释放该锁，而是在虚拟机安全点检查到有等待该锁的线程时释放。这一技术能大幅提高线程局部性强之应用的访存依赖追踪高效性，但在依赖产生频繁的应用中会造成较大的性能问题。因此该技术的扩展 [30] 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)。

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。大部分访存依赖追踪技术在对运行系统修改后保证了程序执行满足顺序一致性 [69, 28, 32, 31]。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 [46]) 以实现高效的依赖获取。基于硬件定制的技术能够捕获硬件底层乱序执行、写缓存、总线事务等行为，从而记录松弛内存系统上的并发程序行为 [34, 130, 67, 101, 66]。

3.2.2 离线合成

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并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之信息可能不足以合成一致的依赖。原则上说，只要有极少量的信息 (如程序崩溃的现场信息 [136])，就能通过搜索的方式合成出用于重现并发缺陷的执行。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实现记录的高效性/简化性与合成的准确性/高效性之间的权衡 (图 3.5)。

3.2.2.1 框架合成

框架合成技术放弃准确性，只记录运行时的框架信息，实现记录高效性和简化性的同时牺牲合成的准确性和高效性。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 [136]。由于在程序执行过程中完全没有实施记录，恢复 $\prec_{tot}$ 是不可能的，但搜索算法在理论上保证能返回一个与 $\prec_{tot}$ 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)，以达到调试的目的。通过记录对错误重现有重大意义的信息 (如 Park 等人提出记录程序中的同步顺序 [109]；Altekar 等人提出记录程序的输出 [12]) 并约束生成的访存依赖满足所记录的信息且以这些信息制导搜索，则能有效地减少搜索空间、降低依赖合成的代价。

3.2.2.2 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 $\prec_d$。虽然 $\prec_d$ 可能不完全准确地反应 $\prec_{tot}$ 中事件执行的顺序，但合成得到的 $\prec_d$ 保证每个线程执行与 $\prec_{tot}$ 相同的路径，从而可保证并发程序行为 (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不需锁、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 访问的变量 $e_i.x$ 以及相应读/写操作的值 $e_i.v$。Netzer 等人证明了若所有写操作的 $e_i.v$ 均不同，则存在一个 $O(n \log n)$ 的算法推导出 $\prec_d$ (n 为总事件数量) [106]；但若允许写入相同值，则满足条件的 $\prec_d$ 不唯一且合成的时间复杂性是 NP-完全的 [57]。

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，以减小问题的规模 [87, 103]。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 $\prec_{po}$ 的约束可用 $O_i < O_j$ 表示，访存事件 e_i 和 e_j ($e_i.x = e_j.x$) 的数据依赖关系可用

$$(e_i.v = e_j.v) \wedge (O_i < O_j) \wedge \forall e_k \in E.\, (e_k.x \neq e_i.x) \vee (O_k < O_i) \vee (O_k > O_j)$$

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 $\prec_d$。

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^t) goto i_j 指令时记录条件判断的真/假值，以得到高度可压缩的记录 [74]。这一记录可以用于确定每一线程唯一的执行路径 (从而推导出每一线程的访存事件)。通过在文献 [87] 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。由于现实程序具有各类局部性，通过记录一定的额外信息，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，并根据此信息推断出具有物理时间先后关系的访存事件以删除冗余的约束，提高约束求解的效率 [135]。

由于数值/路径合成的记录完全存放于线程本地，因而其对运行系统的修改通常不会对内存模型造成影响。所以在对访存语义进行适当建模后能实现松弛内存模型意义上的重放。例如 Lee 等人扩展了文献 [103] 的建模方式以适用于 TSO 内存模型 [86]；另一文献 [74] 提出的技术则能生成在 PSO 内存模型下的访存事件调度。

3.2.2.3 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写后读、写后写和读后写依赖。在偏向锁的设计中，写后读和写后写依赖通常可以较小的开销得到，而记录读后写依赖的代价则可能超过其他二者之和 [28]。在依赖合成中，写后读和写后写依赖可以推导出读后写依赖。因此 Zhou 等人提出了使用写后写依赖和读值合成写后读、读后写依赖的技术，能够在多项式时间内实现 $\prec_d$ 的合成 [144]。Liu 等人进一步证明了即使在运行时只记录写后读依赖，也可保证合成 $\prec_d$ 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 [90]，然后使用类似于文献 [87] 的技术构造约束进行求解，并且隐式地使用了同步局部性进行优化。

3.3 访存依赖的应用

3.3.1 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，轨迹分析技术需要完全准确或一致的访存依赖以正确地恢复执行轨迹、检出并发缺陷。

3.3.1.1 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能复现执行并予以单步调试：要求获得满足一致性的访存

依赖 $\prec_d$，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，能保证重现 $\prec_{tot}$ 所对应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，而高效性和简化性只要满足应用需求即可，可以使用依赖追踪或依赖合成技术实现 [52, 69, 144]。最后，由于对运行系统的修改不可避免会改变程序的语义 (如增加额外的同步操作)，而导致部分缺陷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 [129])。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技术框架和实现 [66]。

执行重放的另一应用场景是生产系统缺陷的复现。在这一场景中，获取一致的访存依赖带来的开销通常是难以接受的。此时依赖合成技术 [136, 109, 12] 可以较少的运行时开销帮助开发者复现生产环境中的并发缺陷。

3.3.1.2 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缺陷。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，以预测潜在的并发缺陷。

数据竞争检测是访存依赖的一类直接应用。定义数据竞争为 $e_i \asymp e_j$ 且 e_i 和 e_j 之间没有发生任何直接或间接的同步操作。数据竞争是多种并发缺陷的诱因 [94]，应为开发者所尽力避免。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，即确定每一个访存事件 e 的数据依赖，因而其记录难以实现高效、简化。长久以来，这类预测分析都带来较大的运行时开销。Flanagan 等人提出了一个时钟维护方法，大大降低了开销 [55] 并被广泛使用 [25]。Wilcox 等人在其基础上提升了数组访问的高效性 [127]，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，通过检测准确的数据竞争而提升高效性 [134]。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 [82] 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 [122]。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 [118] 及基于此的预测分析算法 [71]。Huang 等人则提出将一般性的缺陷模式 (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，使用约束求解器找出轨迹中的并发缺陷 [70]。

这些基于访存依赖的动态分析技术不仅能检测数据竞争，还能检测原子性违反 [73, 95]、假共享 [91] 等并发缺陷，在并发程序质量保障方面起到了卓著的成效，但因其具体技术超出本文的讨论范围而不再叙述。

3.3.1.3 轨迹的其他应用

一次执行轨迹分析的结果也能为更复杂的并发程序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 [116]；无状态模型检验技术 [102, 108] 通过反转执行轨迹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 [29]；Lee 等人提出利用轨迹信息决定静态变量应当指派的锁 [85]。

3.3.2 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，从而实现并发缺陷的避免。此类技术可实现为编程语言的扩展以供开发者在编程时使用 (如事务内存)，也可直接在运行时系统上实现 (如数据竞争避免、确定多线程)。由于并发控制技术通常用于生产系统，其主要挑战便成为高效地在运行时获取准确或一致的访存依赖。

3.3.2.1 事务内存

事务内存允许开发者使用 atomic 关键字包裹一段需要实现原子性的程序代码，并由运行时系统保证这段代码的原子性 (即满足可序列化 [60])。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。访存依赖追踪技术的两个关键技术 (锁指派、锁实现) 也可以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 [47]，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是十分类似的。Dice 等人还提出使用变量指派与用位压缩存储字节数组实现的偏向锁，大幅提高了读操作的性能 [48]；Dalessandro 等人提出将所有变量指派为同一锁，从而能实现最简的冲突检测和解决 [43]；Zhang 等人提出基于偏向锁 [28] 实现的事务内存，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 [137]。

3.3.2.2 数据竞争避免

数据竞争带来并发缺陷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，避免数据竞争所致并发缺陷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以保证其原子性，从而降低并发缺陷发生的概率。为避免数据竞争及其引发的其他后果，Berger 等人提出使用分页保护技术实现页面指派 [21] 并建立了松弛内存模型系统，使竞争写操作发生在线程本地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，以避免数据竞争 [117]。

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，在数据竞争发生时抛出异常 [96]。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读后写依赖的追踪，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 (仅能检出区域间的读后写数据竞争) [25]。

3.3.2.3 确定多线程

鉴于并发缺陷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，以降低并发程序测试、调试的难度，并在运行时避免并发缺陷的发生。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，以此实现系统的确定化 [45]。Bergan 等人扩展文献 [21] 的技术将写操作缓存在本地从而减少访存依赖的数量，在实现确定多线程的同时减少等待令牌的时间 [19]。另一类工作 [46, 92, 20] 则利用存储保护机制以页面粒度实施锁的指派来实现确定化。

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，从而降低可能导致出错的调度 (如数据竞争和原子性违反) 的概率，实现并发缺陷的避免 [42, 41]。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 [40]。

3.4 研究契机

本章提出的技术框架以“访存依赖获取”为主线，综述了来自计算机体系结构、计算机系统、程序设计语言和软件工程四个领域的研究工作，探讨了访存依赖获取技术在即时性、准确性、高效性和简化性四个方面作出的权衡，并且研究了现有基于访存依赖的并发程序动态分析技术。

在这一过程中，我们发现已有技术的研究思路通常是从具体的动态分析技术出发，在一个特定的问题空间中设计访存依赖获取的技术，其视角具有局限性。只有少量工作认识到访存依赖获取问题的重要性并予以单独研究 [28]。即便如此，其研究也缺乏系统性。回顾我们已经完成的访存依赖获取问题的理论框架，其中既包含对问题的形式化定义，也包含并发程序对共享内存访问的局部性理论。我们以这一理论框架为指导，在技术框架中发现如下研究契机：

1. 现有访存依赖获取技术对现实中并发程序所具有的性质利用不足。特别地，虽然有一些工作使用了现实并发程序所具有的局部性 [28, 144]，但因缺乏系统化的局部性理论指导，其对局部性的利用是较为初步的 (仅利用了一些比较明显易用的性质，例如具有线程局部性的读操作占绝大部分)。
2. 在技术框架中，我们发现访存依赖虽在并发程序动态分析中得到广泛应用，但相关的技术仍有不足之处，需通过新技术实现更高效/更有效的动态分析。此外，由于软件产业的增长，社会对软件可靠性提出日益增长的需求，新兴类型软件 (如移动应用) 的质量保障仍然未能有效实现，存在研究契机。技术框架正能指导我们系统性地总结现有工作、解决在新型应用场景中遇到的新问题。
3. 纵观二十多年来对访存依赖获取问题的研究，我们发现尚未有人能在不引入额外同步的前提下、在多项式时间内获取访存依赖。已有的研究表明，只通过线程本地的记录实现访存依赖合成是 NP-完全问题 [57]；只通过读/写实现互斥则是不可能的 [15]。在这两个结论的基础上可能引发对访存依赖获取复杂性的深入探讨。

我们分别对这三点展开了研究。鉴于目前访存依赖获取技术对局部性利用不足的局限，我们在第 4 章提出基于局部性理论的高效访存依赖获取技术：基于乐观锁的访存依赖追踪和基于二分组协议的访存依赖约减。针对技术框架中动态分析技术方面存在的空白，我们在第 5 章提出多种基于访存依赖的动态分析技术——基于缓存的执行重放、基于二分组协议的动态分析、崩溃一致性的自动检测、移动应用的并发缺陷暴露，以实现软件质量保障。针对缺乏访存依赖获取问题理论探讨的研究现状，我们在第 6 章提出“没有免费午餐”猜想：在

仅对程序进行 wait-free 修改且对读/写事件的记录限于线程本地的前提下，获取满足顺序一致性的访存依赖是 NP-完全问题。我们对这一猜想的两个特殊情形提出了证明，并探讨了这些理论结果对未来工作的启示。

第四章　高效的访存依赖获取

基于第 2.2 节提出的局部性理论，并发程序的共享内存访问具有线程局部性和空间-线程局部性的特征：在一段时间内，在同一线程内对同一变量 (或一段地址空间上的连续变量) 的连续访问倾向于具有相同的线程间数据依赖。根据对第 3 章中技术框架的进一步总结，我们发现已有的访存依赖获取技术对并发程序共享内存访问的局部性利用不足。因此，本章介绍我们设计、实现的利用并发程序共享内存访问局部性的高效访存依赖获取技术：

1. 基于并发程序具有的线程局部性，我们提出一种 $O(1)$ wait-free 的算法用于检测读操作是否满足线程局部性，称之为“乐观锁”技术。这一检测技术仅会漏报而不会误报实际具有线程局部性的事件，因而能高效地过滤具有线程局部性的访存事件，同时保证获得一致的访存依赖。第 4.1 节描述基于乐观锁的访存依赖追踪技术，包括问题定义与解决思路 (第 4.1.1 节)、利用线程局部性实现的访存依赖追踪算法 (第 4.1.2 节) 与在顺序一致性内存模型和松弛内存模型上的正确性讨论 (第 4.1.3 节)。
2. 基于并发程序具有的空间-线程局部性，我们提出二分组协议在运行时动态维护满足空间-线程局部性的地址空间划分，并在空间-线程局部性被违反时对划分进行调整，以实现空间-线程局部性的利用，而在得到一致访存依赖的前提下，实现访存依赖记录数量的大幅减少。第 4.2 节描述基于二分组协议的访存依赖约减，包括问题定义与解决思路 (第 4.2.1 节)、二分组协议的基本原理和协议描述 (第 4.2.2 节)。

最后，我们在 LLVM 工具上实现了乐观锁技术，并在此基础上实现了二分组协议，将其用于 C/C++ 程序的在线访存依赖追踪，并使用来自桌面计算、科学计算和服务器的程序对其性能进行评估。详细的系统实现和实验评估描述将在第 4.3 节予以展示。

4.1　基于乐观锁的访存依赖追踪

4.1.1　问题定义与解决思路

问题定义. 在技术框架中，我们已经认识到在共享内存访问事件发生前即时获得较为准确 (满足一致性) 的访存依赖是众多并发程序动态分析的基础 [28]。这类具有即时性和准确性的访存依赖在线追踪技术需要使用锁对共享内存访问进行包裹，因而满足高效性 (较少的运行时开销) 就成为一项非常具有挑战性的工

作。朴素的算法在访问共享内存变量 x 时使用互斥锁保证访存事件和数据更新的原子性，但由于这一处理串行化了所有对 x 的访问 (在现代多处理器系统中，多个处理器核心可以同时读取同一共享变量)，会造成较大的运行时开销，拖慢并发程序的执行。

为了实现即时、准确、高效的在线访存依赖追踪，我们希望保证临界区原子性的 $\texttt{lock}(\ell_x)/\texttt{unlock}(\ell_x)$ 操作能够尽可能地高效。虽然分布式理论表明，使用共享内存读/写操作在 wait-free 的前提下实现互斥是不可能的 [15]，但我们仍然能够利用并发程序实际执行所具有的特征 (如局部性) 来减少锁操作在大部分情况下的开销。

一类有代表性的高效访存依赖获取技术使用乐观锁 [28] 利用并发程序的线程局部性以减少访存依赖追踪的运行时开销。线程局部性表明，在一段时间内，对同一变量的访问要么被某一线程独占，要么被多个线程以只读的方式共享。因此，如果我们能有效地识别出这些情况 (如只使用 $O(1)$ 的时间开销并不引入额外的同步)，并根据绝大部分共享内存访问具有局部性的事实，就能大幅提高访存依赖获取的效率。

乐观锁技术最早被用于实现共享内存多处理器中的缓存一致性 [107]。MESI 或类似的协议在现代共享内存多处理器系统中被广泛应用：对于线程独占的缓存线，它们均处于 M (Modified) 或 E (Exclusive) 状态因而并不引发处理器间的通信；处于 S (Shared) 状态的缓存线可以在多个处理器间共享数据；仅当产生处理器间数据依赖时才在处理器间进行总线通信。类似地，Octet [28] 在软件上实现了类似的协议：当线程获得对于共享变量 x 的互斥锁 ℓ_x 后并不立即释放，而是在另一线程需要对 ℓ_x 上锁时才对持有锁的线程发出请求，在持有锁的线程运行到虚拟机安全点 (VM Safe Point) 时检查请求并释放锁。然而，这类技术在共享内存访问不满足线程局部性时将带来巨大的开销 [30]，且需要虚拟机安全点的支持，具有一定实用上的缺陷。

本节讨论如何对并发程序的线程局部性加以利用，以实现即时、准确 (满足一致性)、高效的访存依赖在线追踪。

解决思路. 本节实现的高效访存依赖在线追踪算法基于如下两点观察：

1. *对共享内存的读操作可以多次进行，且取任意一次的结果作为实际读取的数值，都不破坏程序的语义*。换言之，对于共享变量 x 的读指令 $x_k^t = R(x)$，我们在其前后插入若干对共享变量 x 的读取指令，只要其不改变线程局部变量 X_t 的状态，就不会改变程序的状态、执行路径和输出结果。

```
1  r_x ← r_x[t ↦ e_r];
2  e_r : x_k^t = R(x);
3  if ¬thread_locality_test(e_r) then
4      lock(ℓ_x); // e_r 可能不满足线程局部性
5      begin
6          e'_r : x_k^t = R(x); // 在临界区中再次读取 x
7          e_w ← w_x;
8          r_x ← r_x[t ↦ e_r];
9      unlock(ℓ_x);
10     thread_locality_update(e_w); // 更新用于线程局部性测试的状态
11     记录访存依赖 e_w ≺_{w→r} e_r;
```

图 4.1 乐观锁对读指令作出的修改，下划线为被修改的读指令。

2. 具有线程局部性的读事件占并发程序共享内存访问的绝大部分。这是由于 (1) 并发程序的共享内存访问具有线程局部性；(2) 现实并发程序执行轨迹中的读事件通常远远多于写事件。

基于这两点观察，我们设计乐观的锁机制，对读和写操作分别处理，通过避免对具有线程局部性的读操作使用同步操作，以达到高效访存依赖追踪的目的：

1. 对于共享变量 x 的写操作，我们使用互斥锁 ℓ_x 进行保护，并对访存依赖予以记录。由于写操作只占共享内存访问中的少部分，这一处理的开销通常在可以接受的范围内 [144]。

2. 对于共享变量 x 的读操作，我们首先执行这一读操作，并对读操作是否具有线程局部性进行快速检测。由于大部分读操作均具有线程局部性，检测通过则意味着无需记录访存依赖。若检测不通过，根据我们可以多次读同一变量而不破坏程序语义的性质，我们使用互斥锁 ℓ_x 包裹对 x 的再次读操作，并于互斥锁的保护下获取满足一致性的访存依赖。

因此，实现这一技术的主要挑战是实现一个只会引起单向错误的事件线程局部性检测：若检测认为读事件 e_r 具有线程局部性，则这一结果一定是准确的；反之，若检测认为 e_r 不具有线程局部性，其仍然有可能是一个具有线程局部性的读事件。同时，这一检测应满足能在 $O(1)$ 时间内完成，并不使用任何同步/原子操作。

```
1  lock(ℓ_x);
2  begin
3      e'_w ← w_x; w_x ← e_w;
4      e_w : W(x, x_k^t);
5      r ← r_x; r_x ← ∅;
6  unlock(ℓ_x);
7  thread_locality_update(e_w);
8  if e_w.t ≠ e'_w.t then
9      记录访存依赖 e'_w ≺_{w→w} e_w;
10 for t ∈ T do
11     if r(t) ≠ ⊥ then
12         记录访存依赖 e_r ≺_{r→w} e_w;
```

图 4.2 乐观锁对写指令作出的修改，下划线为被修改的写指令。

4.1.2 基于乐观锁的访存依赖追踪

为了实现访存依赖的追踪，乐观锁技术在运行时为每一共享变量 x 维护以下数据结构：

1. 最近一次对共享变量 x 的写入事件 w_x (w_x 可用线程号和线程本地的时间戳表示)。虽然在算法描述中我们为表述明确令 $w_x = W(t, x, v)$，但实际实现时只需在 w_x 中记录写入 x 时的时间戳和写入的线程号，即可实现访存依赖的追踪。
2. 在最近一次对共享变量 x 的写入事件之后、当前事件之前发生的读事件集合 r_x。对于发生在同一个线程的多个对 x 的读取事件，我们只需要保留其中发生时间最晚 (由于这些事件发生在同一线程，因此 $\prec_{po}$ 决定了它们发生的先后顺序) 的读事件即可获得满足一致性的访存依赖。因此，r_x 是 $T \to E$ 的映射，$r_x(t)$ 为 t 最后读取 x 的事件 (初始映射 $r_0 = \varnothing$ 满足 $\forall t.\, r_0(t) = \bot$)。

我们暂时假设对 w_x 和 r_x 的操作 (映射的更新和重置) 能够原子地完成，并在算法描述之后讨论如何对这些操作进行高效、无锁的实现。

为实现访存依赖追踪，我们对读/写共享内存变量 x 的指令进行修改 (通过程序插装实现)，分别如图 4.1/图 4.2 所示：

1. 对于读事件 e_r: $R(t, x, v)$，我们在执行读操作前首先更新 r_x，使得与 e_r 相关的读后写依赖能被后续的写指令正确捕获 ($r_x \leftarrow r_x[t \mapsto e_r]$ 更新 t 所对

```
1 Function thread_locality_test(e_r)
2 begin
3     e_w ← w_x;
4     if C_t(e_r.x) ≠ e_w then
5         return True;
6     return False;
7 Function thread_locality_update(e)
8 begin
9     C_t(e.x) ← e;
```

图 4.3 检测读事件 e 是否具有线程局部性的算法。

应的映射值 $r_x(t) = e_r$)，在 r_x 更新完毕后，才实际执行读操作。

读操作执行完毕后，调用 thread_locality_test 对线程局部性进行检查。仅当无法确定 e_r 具有线程局部性时，才在锁 ℓ_x 保护下再次对共享变量 x 进行读取，以本次读取的数值替代第一次读取的数值 (第 6 行)，并得到写后读依赖。

2. 对于写事件 e_w: $W(t, x, v)$，我们使用互斥锁 ℓ_x 对其进行保护，在临界区中完成 r_x, w_x 的更新和写后写、读后写依赖的获取。

对于记录的一条访存依赖 $e_i \prec_d e_j$ 中 e_i 和 e_j 的读/写类型，我们可以将 $\prec_d$ 分为写后写 (write-after-write) 依赖 $\prec_{w\to w}$、写后读 (read-after-write) 依赖 $\prec_{w\to r}$ 和读后写 (write-after-read) 依赖 $\prec_{r\to w}$ 三种类型。根据图 4.1 和图 4.2 中的算法描述，易见这一划分是完备的：$\prec_d = \prec_{w\to w} \cup \prec_{w\to r} \cup \prec_{r\to w}$。

下文我们根据访存依赖的类型分别讨论其获取方法。

追踪写后写依赖 $\prec_{w\to w}$. 若对 x 的连续两次共享内存写事件 $e_{w_1} \prec_{tot} e_{w_2}$ ($e_{w_1}.rw = e_{w_2}.rw = w$, $e_{w_1}.x = e_{w_2}.x$ 且不存在其他对 x 的写事件 e 满足 $e_{w_1} \prec_{tot} e \prec_{tot} e_{w_2}$) 发生在不同线程 ($e_{w_1}.t \neq e_{w_2}.t$)，则它们满足冲突事件的定义 ($e_{w_1} \asymp e_{w_2}$)，因此它们之间的依赖必须被记录。

图 4.2 第 2-5 行的临界区串行化了所有对 x 的写指令 (第 4 行)。相应的写事件 e_{w_1} 在临界区中赋值给 w_x (第 3 行)，因此能保证在 e_{w_2} 发生时，执行到算法第 3 行的 $e'_w \leftarrow w_x$ 时读出 e_{w_1} (即 $e'_w = e_{w_1}$)。由于 $e_{w_1}.t \neq e_{w_2}.t$，第 8 行的判断必定为 False，从而记录 $e_{w_1} \prec_d e_{w_2}$。

追踪写后读依赖 $\prec_{w\to r}$. 为了实现写后读依赖的追踪，我们使用图 4.3 中的算法在对 x 的读操作之后对其是否具有线程局部性进行检查。若共享内存的读事

件 e_r (图 4.1 第 2 行) 具有线程局部性，考虑在 $\prec_{tot}$ 中距 e_r 最近的一次写事件 e_w ($e_w \prec_{tot} e_r$, $e_w.rw = w$, $e_w.x = e_r.x$ 且不存在其他向 x 的写事件 e 满足 $e_w \prec_{tot} e \prec_{tot} e_r$)，则以下二者之一必然成立：

1. e_w 与 e_r 发生在同一线程，即 $e_w.t = e_r.t$，因此 e_r 仅包含线程本地的数据依赖，写后读依赖$e_w \prec_{w\to r} e_r$ 能被 $\prec_{po}$ 推出从而无需记录；或
2. 存在与 e_r 发生在同一线程的读事件 e'_r ($e'_r \prec_{po} e_r$, $e'_r.rw = r$, $e'_r.x = e_r.x$)，且其发生在 e_w 之后，即 $e_w \prec_{tot} e'_r \prec_{tot} e_r$。在此情况下该线程在过去已经进行过对 x 的读操作 (因此 $e_r.v = e'_r.v$)，写后读依赖$e_w \prec_{w\to r} e_r$ 能被 $e_w \prec_{tot} e'_r \prec_{po} e_r$ 推出，因此也无需记录。

为了检出这两种情况，我们在每一线程 t 维护映射 C_t，$C_t(x)$ 为线程 t 对共享变量 x 已知的最晚写事件。在每次线程 t 写入 x 时 (图 4.2 第 7 行) 或产生一次不具有线程局部性的读事件时 (图 4.1 第 10 行)，我们都调用 thread_locality_update 对这一数值进行更新。

我们在检测读事件 e 是否具有线程局部性时没有使用任何同步操作。为了保证访存依赖追踪的正确性，我们巧妙地设置内存访问的顺序以确保 thread_locality_test 不会将实际不具有线程局部性的事件判定为通过：

1. 在执行写操作时，我们首先更新元数据 w_x (图 4.2 第 3 行，对应事件为 $\boxed{W(w_x)}$[1])，再写入 x 的数值 (图 4.2 第 4 行，对应事件为 $\boxed{W(x)}$)，因此 $\boxed{W(w_x)} \prec_{po} \boxed{W(x)}$；
2. 在执行读操作时，我们首先读出 x 的数值 (图 4.1 第 2 行，对应事件为 $\boxed{R(x)}$)，再读出 w_x (图 4.3 第 3 行，对应事件为 $\boxed{R(w_x)}$)，并与线程本地记录的事件 $C_t(x)$ 进行比较。因此 $\boxed{R(x)} \prec_{po} \boxed{R(w_x)}$。

对于一个不具有线程局部性的读事件 e_r 和写入 $e_r.v$ 数值的写事件 e_w 满足 $e_w \prec_{tot} e_r$，即 $\boxed{W(x)} \prec_{tot} \boxed{R(x)}$。根据偏序关系的传递性，这些事件的顺序被确定为：

$$\boxed{W(w_x)} \prec_{po} \boxed{W(x)} \prec_{tot} \boxed{R(x)} \prec_{po} \boxed{R(w_x)}。$$

换言之，在 e_w 发生前，对 w_x 的更新就已经发生了。因此，在 e_r 之后发生的对 w_x 的读取保证获得与线程本地缓存的 $C_{e_r.t}(x)$ 不同的值 (由每次向 w_x 写入不同的值保证)。

至此，我们实现了 $O(1)$ 不使用任何同步操作或原子操作的读事件线程局部性检测。这一算法被称为“乐观锁”，是因为其首先乐观地假设所有内存读取

[1]在不产生歧义的前提下，我们使用 $\boxed{R(x)}$ 表示一个对 x 的读事件、$\boxed{W(x)}$ 表示一个对 x 的写事件以简化描述。

均具有线程局部性。在我们的算法中，读操作将被直接执行，然后“e_r 具有线程局部性”这一假设被立即验证。若验证成功则无需记录访存依赖 (fast path)，否则回退到“悲观”的互斥锁并重做读操作 (slow path)。由于现实程序的大部分共享内存访问是具有线程局部性的读操作，这一处理能有效减少运行时的时间开销和访存依赖记录的数量。

乐观锁实现访存依赖获取的正确性将在第 4.1.3 节统一讨论。

追踪读后写依赖 $\prec_{r\to w}$. 对于任意事件 e (无论读写)，与它相关的写后读依赖 $e_w \prec_{w\to r} e$ 或写后写依赖 $e_w \prec_{w\to w} e$ 至多仅有一个 (数据依赖仅发生在最近一次对 x 的写事件 e_w 和 e 之间)，但在两次对 x 的写入事件 e_{w_1} 和 e_{w_2} 之间，可能存在发生在不同线程的多个读事件 e_r，此时每个 $e_r \prec_{r\to w} e_{w_2}$ 都必须被准确记录。由于读后写依赖是一对多的关系，因此其追踪也更具有挑战性。

我们使用类似于追踪写后读依赖的内存访问顺序以追踪读后写依赖。用 r_x 维护在上一次对 x 的写事件之后每一线程最后一次对 x 的读事件。r_x 实现为一个 $T \to E$ 的映射 (我们暂时假设其更新和读取能够原子地完成)。由于对 r_x 的更新 ($r_x \leftarrow r_x[t \mapsto e_r]$) 发生在读操作之前 (图 4.1 第 1 行)，我们保证 $\boxed{W(r_x)} \prec_{po} \boxed{R(x)}$；由于对 r_x 的读取发生在写操作之后 (图 4.2 第 5 行)，我们保证 $\boxed{W(x)} \prec_{po} \boxed{R(r_x)}$。对于一个读后写依赖 $e_r \prec_{r\to w} e_w$ (即 $\boxed{R(x)} \prec_{tot} \boxed{W(x)}$)，根据偏序关系的传递性，我们有

$$\boxed{W(r_x)} \prec_{po} \boxed{R(x)} \prec_{tot} \boxed{W(x)} \prec_{po} \boxed{R(r_x)},$$

因此对于任意的读后写依赖 $e_r \prec_{r\to w} e_w$，只要 r_x 维护的原子性得到保证，在 e_w 发生时就能保证从 r_x 中得到 e_r，不会遗漏任何读后写依赖。

与 w_x 的维护类似，我们在维护 r_x 时没有保证 r_x 的读写和其他操作之间的原子性。因此可能存在读事件 e_r 发生前更新了 r_x 并与某个写事件 w_x 形成 $e_r \prec_{r\to w} e_w$ 的读后写依赖，但由于 e_r 的线程局部性检测 thread_locality_test 失败，导致 $e_w \prec_{w\to r} e_r$ 被记录，从而在 $\prec_d$ 中形成 e_r, e_w 两个事件之间的循环依赖。这一情形的避免将在正确性讨论中予以说明。

实现高效的 w_x 和 r_x 维护. 我们之前的讨论要求对 w_x 和 r_x 的读取、写入和更新满足原子性。若在对 w_x 和 r_x 的维护中引入额外的同步操作，则会导致性能瓶颈。因此，必须高效 (无锁) 地对这两个数据结构进行维护。

对 w_x 的维护是较为直接的。首先，w_x 仅需要存储一个数值 (某个写入事件 $e_w \in E$)，因此可以用固定长度的数据结构表示。易见所有对 w_x 的写入都被串行化 (对 w_x 的写入仅发生在图 4.2 第 3 行，且处于 ℓ_x 保护的临界区中)，因此直接将 w_x 作为普通的共享内存变量进行读取/写入即可。在现代多处理器系

```
1  function hash_lookup(r_x, t)
2  begin
3      h = R(r_x);
4      i ← hash_probe(h, t);
5      if 查找失败 then
6          return FAIL;
7      return i;
8  function hash_reclaim(r_x)
9  begin
10     W(r_x, new());
11 function hash_insert(r_x, ⟨t, e⟩)
12 begin
13     h = R(r_x);
14     i ← hash_lookup(h, t);
15     if 查找失败 then
16         lock(ℓ_x);
17         begin
18             j ← find_insert_pos(t);
19             W(h(j).key, t);
20             W(h(j).value, e);
21         unlock(ℓ_x);
22     else
23         W(h(i).value, e);
```

图 4.4 读事件集合 r_x 的维护算法。

统中，只要 w_x 在内存中不跨越缓存线的边界，其更新就能原子性地完成，对 w_x 的读取不会读到部分值。

不同于 w_x 仅存储一个访存事件，r_x 中存储 $T \to E$ 的映射，因此其维护更具挑战性。由于在每次共享内存读操作之前，我们都要将 r_x 中当前线程 t 对应的读事件更新，即 $r_x \leftarrow r_x[t \mapsto e]$，而具有线程局部性的读操作又占共享内存访问的绝大部分，因此我们必须实现 r_x 的高效 (常数时间、无锁) 维护。能实现 $O(1)$ 对映射进行维护的数据结构是散列表 (Hash Table)，然而，目前仍然没有满足 wait-free 性质的通用并发散列表。分布式计算领域目前最好的实现 [93] 在性能上仍然难以满足访存依赖追踪的需求。

为了实现 r_x 的高效维护，我们利用了其访问的特殊性。对 r_x 的操作仅发生在图 4.1 第 1 行和图 4.2 第 5 行，总共分为三种：

1. 线程 t 在无同步的情况下更新其对应的写者：$r_x \leftarrow r_x[t \mapsto e_r]$；
2. 线程 t 在有锁保护的情况下对 r_x 进行扫描，以获取其全部元素；
3. 线程 t 在有锁保护的情况下对 r_x 进行初始化操作 $(r_x \leftarrow \varnothing)$；

在所有情形中，没有锁保护的情形仅有线程 t 对 $r_x(t)$ 进行更新 (对 r_x 的读取和初始化均被串行化)，即线程 t 永远只更新 r_x 中键为 t 所对应的值。在这一限制条件下，我们只需保证数据结构的单调性 (若某次访问 $r_x(t)$ 能获得 t 对应的键-值对，则后续任意访问 $r_x(t)$ 均能获得同一键-值对)，即可实现高效的 r_x 维护：只需在查找 t 失败时使用原子操作/同步操作进行散列表插入，而在查找 t 成功时则可以直接更新键-值对中的值。显然，使用闭地址、拉链法解决冲突的散列表满足这一单调性，线程 t 在将 t 插入链表时使用原子操作对链表头进行更新，此后散列表对 t 的查找将总是成功，以实现 r_x 的正确维护。另一个可行的开放地址散列实现如图 4.4 所示。这些实现均能保证在 t 已在散列表时能在 $O(1)$ 时间内无同步、无原子操作地实现 r_x 的更新。

4.1.3 正确性讨论

4.1.3.1 在顺序一致内存模型上的正确性

我们首先假设并发系统的内存模型满足顺序一致性，即与我们在第 2.1 节定义的系统模型相同，存在访存事件的全序 $\prec_{tot}$，并发系统严格按照 $\prec_{tot}$ 规定的顺序执行访存指令：在写事件 e_w 发生后，数值 $e_w.v$ 被立即写入共享内存变量 $e_w.x$、读事件 e_r 的读值 $e_r.v$ 总是来自 $\prec_{tot}$ 中最近一次对 $e_r.x$ 的写事件。

写后写依赖追踪的正确性. 我们使用互斥锁保证了所有对同一变量 x 的写操作以及相应的 r_x, w_x 更新的原子性 (图 4.2 第 1-6 行)。由于写后写依赖的追踪 (读取和更新 w_x) 发生在临界区中，并且对于任意连续两个发生在不同线程的访存事件，都有 $e'_w \prec_{w\to w} e_w$ 被记录 (图 4.2 第 9 行)，因此 $\prec_{d^*} = tr(\prec_d \cup \prec_{po})$ 包含了对所有 x 写入事件的全序。

写后读依赖追踪的正确性. 首先，对于 thread_locality_test 失败的情形 (图 4.1 第 4-11 行)，我们使用锁保护了对共享内存的读指令，并在临界区中读取对 x 的上一次写事件 w_x 的值。这一情形与写后写依赖追踪类似，由于对 w_x 的写操作被同样的临界区保护 (图 4.2 第 1-6 行)，因此对 w_x 的操作不存在数据竞争，能够保证获得与读事件 e_r 相对应的写事件 e_w，从而记录的访存依赖 $e_w \prec_d e_r$ 代表 $\prec_{tot}$ 中这一对读写事件发生的实际顺序。

因此，写后读依赖追踪的正确性只需论证对于实际产生写后读依赖的读事件 e_r，在 e_r 发生后 thread_locality_test 是否一定返回失败。对于 $e_r = \boxed{R(x)}$，$e_w = \boxed{W(x)}$ ($e_r.t \neq e_w.t$ 且不存在对变量 $e_r.x$ 的写事件 e'_w 满足 $e_w \prec_{tot} e'_w \prec_{tot} e_r$)，其写后读依赖及其相应的 w_x 写入/读取事件排列如下：

$$\boxed{W(w_x)} \prec_{po} \boxed{W(x)} \prec_{tot} \boxed{R(x)} \prec_{po} \boxed{R(w_x)},$$

由于 $\boxed{W(x)}$ 和 $\boxed{R(x)}$ 之间产生了实际的写后读依赖，因此不存在对 $e_r.x$ 的读事件 e'_r 满足 $e_w \prec_{tot} e'_r \prec_{tot} e_r$。由于 $\boxed{W(w_x)} \prec_{po} \boxed{W(x)}$ 处于同一临界区，其间不会插入其他对 x 或 w_x 的写入事件。又由于 $\boxed{W(x)} \prec_{tot} \boxed{R(x)}$ 之间包含实际的数据依赖，其间也不存在其他对 x 或 w_x 的写事件。因此，$e_r = \boxed{R(w_x)}$ 读到的数值仅存在两种情况：

1. 读到 $e_w = \boxed{W(w_x)}$ 写入的 w_x；
2. 读到在 $\boxed{R(x)} \prec_{po} \boxed{R(w_x)}$ 之间另一线程 $t \neq e_r.t$ 写入的 w_x。

基于我们每一访存事件都具有全局唯一标识符的假设 (标识符可以用线程号和线程本地的事件计数器实现)，每次写入 e_w 的数值均是全局唯一的。在上述两种情况下，$\boxed{R(w_x)}$ 均读到另一线程对 w_x 写入的数值，由于其全局唯一性，其一定不等于线程 t 本地缓存的数值 $C_t(e_r.x)$。换言之，在存在写后读依赖时，线程局部性检测 thread_locality_test 总是返回 False。

因此，对于存在写后读依赖的读事件 e_r，我们总是会使用同步再次对 $e_r.x$ 进行读取，从而获取准确的写后读依赖。

读后写依赖追踪的正确性. 读后写依赖的正确性与写后读类似。此时 r_x 的更新发生在 x 的读取之前 (图 4.1 第 1 行)，而 r_x 的读取发生在 x 的写入之后 (图 4.2 第 5 行)。因此，对于读事件 $e_r = \boxed{R(x)}$ 和写事件 $e_w = \boxed{W(x)}$ 之间真实存在的读后写依赖，有

$$\boxed{W(r_x)} \prec_{po} \boxed{R(x)} \prec_{tot} \boxed{W(x)} \prec_{po} \boxed{R(r_x)}。$$

假如不存在任何与这一事件序列并发的其他事件，偏序关系的传递性保证了 $\boxed{W(r_x)} \prec_{tot} \boxed{R(r_x)}$，即在写事件发生时能获取正确的读后写依赖。因此，我们只需考虑在这一事件序列中可能发生的并发事件。

首先，因 $e_r \prec_{r\to w} e_w$ 存在实际的数据依赖，序列中 $e_r = \boxed{R(x)}$ 和 $e_w = \boxed{W(x)}$ 之间不存在其他对 x 的写入事件 (否则于读后写依赖矛盾)。由于所有对 x 的写入和 r_x 的读取/更新被包含在同一临界区中 (从而原子性得到保证)，因此 $\boxed{W(x)} \prec_{po} \boxed{R(r_x)}$ 可以看作是瞬时发生的，期间不会插入其他产生数据依赖的事件。在 $\boxed{W(x)}$ 发生前还可能存在其他线程写入 r_x 的事件，但只要 r_x 能正

确支持多个线程的并发写入，就能被 $\boxed{R(r_x)}$ 正确读出。因此，需要考虑的剩余情形是：

1. $\boxed{W(x)} \prec_{po} \boxed{R(r_x)}$ 之间对 r_x 的更新事件。这些更新事件一定来自其他线程对 x 的读取事件 e'_r ($e_r.t \neq e'_r.t$)。图 4.2 第 3 行对 w_x 的更新发生在 e_w 之前 (即 $\boxed{W(w_x)} \prec_{po} \boxed{W(x)}$)，因此 e'_r 的 thread_locality_test 一定返回失败并导致同步。这一情况会导致 $e'_r \prec_{r\to w} e_w \prec_{w\to r} e'_r$ 的成环情况，我们只需将这一环移除即可。
2. $\boxed{W(r_x)} \prec_{po} \boxed{R(x)}$ 之间发生的来自其他线程的 $\boxed{W(x)}$ 事件。使用之前对写后读依赖追踪的正确性分析，对 w_x 和 x 的访存事件满足

$$\boxed{W(w_x)} \prec_{po} \boxed{W(x)} \prec_{tot} \boxed{R(x)} \prec_{po} \boxed{R(w_x)},$$

因此在 $e_r = \boxed{R(x)}$ 之后执行的 thread_locality_test 也必然返回 False，从而进入临界区保护的 slow path。此时，e_r 对应的读事件将作废 (以图 4.1 第 6 行的 e'_r 为准，从而 e_r 对应的依赖无需记录)。e'_r 发生在临界区中因此能获得正确的写后读依赖，并且图 4.1 第 8 行中再次对 r_x 进行的更新能保证 e'_r 之后的读后写依赖能被正确追踪。

因此，我们可以保证在具有线程局部性的读操作中不使用同步操作的前提下获取准确的读后写依赖。

4.1.3.2 在松弛内存模型上的正确性

乐观锁实现不使用同步/原子操作的访存依赖追踪的核心技术是对并发程序/系统进行修改，加入访存依赖追踪相关的代码，其中事件记录和共享内存读写按照如下顺序进行：

$$\boxed{W(a)} \prec_{po} \boxed{A} \prec_{tot} \boxed{B} \prec_{po} \boxed{R(a)},$$

其中 $\boxed{A}$, $\boxed{B}$ 为对共享变量 x 的读/写事件，a 是用于记录访存依赖维护的数据结构 (w_x 或 r_x)。在写后读 ($\prec_{w\to r}$) 依赖的追踪中，$A = W(x)$, $B = R(x)$, $a = w_x$；在读后写 ($\prec_{r\to w}$) 依赖追踪中，$A = R(x)$, $B = W(x)$, $a = r_x$。偏序关系的传递性保证了 $\boxed{W(a)} \prec_{d^*} \boxed{R(a)}$，因此对元数据的更新一定发生在元数据的读取之前，从而实现访存依赖的在线追踪。

这一正确性论证依赖于并发系统满足顺序一致性的内存模型，各个线程按照程序顺序 $\prec_{po}$ 顺次执行所有读/写操作，并且读事件的值保证等于最近一次写事件写入的数值。

然而，现代多处理器系统为了弥补缓存缺失造成的延迟 (维护多处理器间的缓存一致性和从其他存储器调取数值均会造成比指令执行时间长得多的延迟)，

通常都使用了动态流水线技术，以实现指令的动态调度和乱序执行。从应用程序的角度看，这一行为虽然保证了单线程程序的顺序一致性，但由于允许交换共享内存访问的顺序，在这些系统上执行多线程程序得到的执行轨迹将不再满足顺序一致性，因而本节的上述正确性论证也不再成立。

常见的松弛内存模型 (relaxed memory model) 有 Total Store Order, Partial Store Order 等 [75]。我们首先在目前最广泛使用的计算机系统：Intel x86 系统上讨论访存依赖追踪算法的正确性，再讨论更弱内存模型下实现的修改。

x86-TSO 上的正确性. 由于我们对写操作进行了同步，因此写后写 ($\prec_{w\to w}$) 依赖获取的正确性是显然的。接下来我们证明获取写后读 ($\prec_{w\to r}$) 依赖在 x86-TSO [119] 内存模型上的正确性。对于 thread_locality_test 检测出 e_r 不具有线程局部性的情况 (图 4.1 第 4-9 行)，由于我们使用同步操作保护了再次尝试的读操作，因此在这一情形下也能准确地追踪访存依赖。

因此，我们只需证明对于不具有线程局部性的读事件，thread_locality_test 总是返回 False，即能保证在 x86-TSO 上的正确性：

定理 4.1. *在 x86-TSO 内存模型中，若 $e_r = \boxed{R(x)}$ (图 4.1 第 2 行) 读取到另一线程写入的数值，线程局部性测试将返回 False。*

证明. 在 x86-TSO 中，读事件 e_r 必满足如下两种情况之一：(1) 从本线程的写缓存中读出数值；(2) 从另一线程读出数值。对于从本线程写缓存中读出数值的情况，其实际上具有线程局部性 (写缓存中的数值一定是本线程写入的)。因此，无论 thread_locality_test 成功与否，其正确性都能得到保证。

对于 e_r 从共享内存中读出数值的情况，我们依旧考虑修改后的程序所产生的事件序列

$$\boxed{W(w_x)} \prec_{po} \boxed{W(x)} \prec_{tot} \boxed{R(x)} \prec_{po} \boxed{R(w_x)},$$

其中 $e_r = \boxed{R(x)}$，与之相关的写事件 $e_w = \boxed{W(x)}$。

若 e_r 从共享内存中读出另一线程写入的数值，则 w_x 在 x 被更新前更新，即 $\boxed{W(w_x)} \prec_{po} \boxed{W(x)}$。由于 x86-TSO 保证同一线程满足程序顺序的写入操作按 FIFO 顺序进入写缓存，并且写缓存也按照 FIFO 顺序写入共享内存，因而如果 $\boxed{R(x)}$ 读到了这一数值，说明写缓存中 x 的数值已经进入了共享内存。因写缓存的 FIFO 特性，w_x 的数值也到达了共享内存。而对于 $\boxed{R(x)}$ 事件，x86-TSO 中指令按程序顺序执行，$\boxed{R(x)} \prec_{po} \boxed{R(w_x)}$ 保证了此时 w_x 不会在写缓存中 (否则将导致对 x 的写也在写缓存中，与从共享内存读取 x 矛盾)，故线程局部性测试将返回 False。 □

这一证明即便对于不冲刷硬件写缓冲实现的自旋锁 (此类锁是 pthread 线程库和 Linux 内核中的标准实现) 依然成立 [119]。在此情形下，整个系统的行为可能不满足顺序一致性 (文献 [119] 中的 SB 例子)。即使如此，我们的算法仍然能准确地获得每一读事件对应的写者。因此在不记录 $\prec_{r\to w}$ 依赖的前提下，我们即便不对算法作出任何修改 (不插入内存障碍)，仍能通过离线再次运行程序获得准确的访存依赖 ($\prec_{w\to r}$ 依赖能从 $\prec_{w\to r}$ 和 $\prec_{w\to w}$ 依赖离线推导得到，因此这一算法是具有实际意义的)。

如需在追踪访存依赖的同时保证程序在 x86-TSO 内存模型上的顺序一致性，则可通过增加内存障碍来实现：我们只需在写时的锁释放前 (图 4.2 第 6 行) 使用内存障碍清空写缓存，保证在 $\boxed{R(x)} \prec_{po} \boxed{R(w_x)}$ 事件发生时线程本地的写缓存为空，从而确保所有读事件都从共享内存中读出。由于 x86-TSO 保证所有读操作的原子性且不被乱序，则顺序一致性可以得到保证。

对于一对读后写 ($\prec_{r\to w}$) 依赖，修改后程序的共享内存访问事件序列为

$$\boxed{W(r_x)} \prec_{po} \boxed{R(x)} \prec_{tot} \boxed{W(x)} \prec_{po} \boxed{R(r_x)},$$

由于 x86-TSO 不保证对不同变量读和写之间的顺序，因而若不增加额外的障碍将会导致访存依赖的丢失。我们需要使用 MFENCE 指令实现上述 $\prec_{po}$ 关系 (在图 4.2 第 4 行和第 5 行之间、图 4.1 第 1 行和第 2 行之间插入障碍)，以保证访存依赖追踪的正确性。

在更弱内存模型上的正确性. 基于乐观锁的访存依赖追踪技术的核心是使用满足如下顺序的事件序列：

$$\boxed{W(a)} \prec_{po} \boxed{A} \prec_{tot} \boxed{B} \prec_{po} \boxed{R(a)},$$

对于依赖 $\boxed{A} \prec_{tot} \boxed{B}$，通过传递性推导出 $\boxed{W(a)} \prec_{tot} \boxed{R(a)}$，从而在不使用同步的情况下获得访存依赖。

在更弱的内存模型 (如 PSO) 下，用于追踪读后写、写后读依赖时的访存事件 $\boxed{W(a)} \prec_{po} \boxed{A}$ 和 $\boxed{B} \prec_{po} \boxed{R(a)}$ 均可能被乱序。因此，我们只需按如下方式增加内存障碍：

$$\boxed{W(a)} \prec_{po} \textsf{BARRIER} \prec_{po} \boxed{A} \text{ 和 } \boxed{B} \prec_{po} \textsf{BARRIER} \prec_{po} \boxed{R(a)},$$

即可满足算法的正确性和程序的顺序一致性语义。虽然增加了内存障碍，但这一算法在具有线程局部性读事件发生时仍然不会引入额外的同步或原子操作且能在 $O(1)$ 时间内完成，是十分优秀的算法。

4.2 基于二分组协议的在线访存依赖数量约减

4.2.1 问题定义与解决思路

问题定义. 在上一节中我们利用并发程序共享内存访问的线程局部性设计了基于乐观锁的高效访存依赖获取技术，其对于具有线程局部性的读操作可以不引入额外的同步或原子操作，以实现即时、高效、准确的访存依赖在线追踪。

访存依赖最终用于实现各类并发程序的动态分析技术。各类动态分析技术对访存依赖获取技术评价指标的需求不同，即时、高效、准确的访存依赖获取技术并不能满足所有动态分析技术的需求，访存依赖记录的简化性同样对实现有效的并发程序动态分析至关重要。例如，现实并发程序的执行轨迹可能包含海量的共享内存访问，仅仅将这些访存依赖存储都可能导致难以容忍的巨大开销 (例如，将事件记录写入速度较慢的磁盘将成为性能瓶颈)；处理大量访存依赖记录的时间复杂性有时也是难以忍受的，例如基于约束求解的算法通常无法解析过于庞大的访存事件记录。

因此，并发程序的动态分析技术必须在即时性、高效性、准确性和简化性之间做出权衡。并不是所有的分析都需要完全准确反映线程间数据依赖的访存依赖 (即完全准确的访存依赖)，因此放松准确性是一个可行的权衡：如能在运行时追踪满足一致性的访存依赖，就能满足很大一类动态分析技术 (如执行重放调试、并发控制技术等) 的需求。

因此，本节关注高效访存依赖获取技术在即时性、高效性、准确性和简化性之间的权衡，又称为访存依赖的在线数量约减问题：在满足即时性、不降低或轻微降低高效性、轻微降低准确性 (获得的访存依赖仍满足一致性) 的前提下，实现访存依赖数量的大幅减少，提高依赖日志的简化性。形式化地说，我们要求记录的访存依赖 $\prec_d$ 满足对于 $e_i \asymp e_j$，$e_i \prec_{d^*} e_j$ 当且仅当 $e_i \prec_{tot} e_j$，在此前提下尽可能地高效 (具有较少的运行时时间开销)、简化 (减少 $|\prec_d|$ 记录集合的大小)。

相关工作. 为了提高依赖日志的简化性，其约减可以通过在访存依赖 $e_i \prec_d e_j$ 被捕获后对其冗余性进行检查实现，即通过排除一些不影响动态分析技术正确性的依赖以达到减少记录数量的目的。我们称这类技术为在线约减技术。另一方面，依赖约减也可以在程序执行结束后 (此时访存依赖记录已经获得) 进行，把“获得等价并且更简化访存依赖”这一问题建模成组合优化问题实现，称为离线约减技术。

对比在线、离线两类访存依赖约减技术，在线约减能实现运行时访存依赖记录的减少 (离线约减需先记录日志再进行求解)，从而其应用场景通常不受限

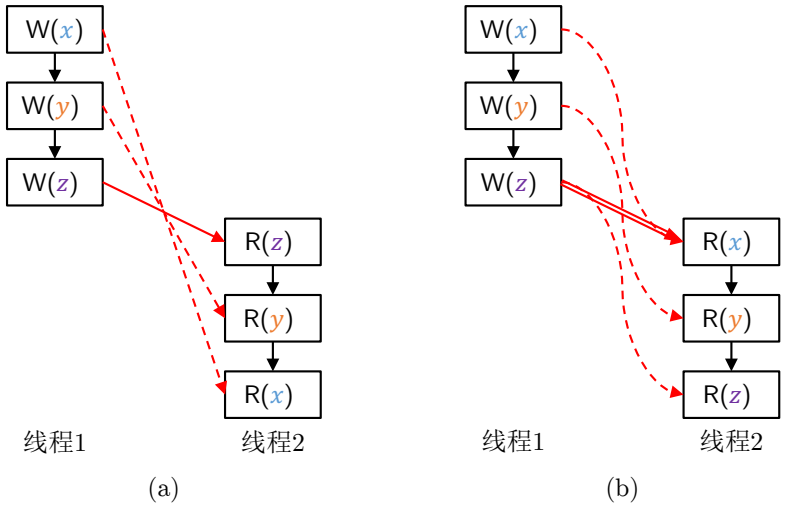

图 4.5 访存依赖的约减：(a) 传递约减 (TR), (b) 正则传递约减 (RTR)。虚线为被约减的依赖，双线为添加的依赖。

制。其挑战主要在于访存依赖获取是服务于动态分析技术的，因此在程序运行时实现访存依赖约减的同时不能大幅降低其即时性、有效性、准确性等指标，否则将无法达到动态分析技术的预期。离线约减技术由于直接根据完整的访存依赖记录 $\prec_d$，能够使用专门算法或组合优化求解器对依赖关系进行化简 [72, 76]，因此通常能得到较在线约减技术更为简化的记录，但缺点是在运行时仍要产生大量的记录。本节关注访存依赖的在线约减问题，能在即时获得访存依赖的前提下获得简化的记录，从而能直接用于使用访存依赖的动态分析技术。

一个直观的在线访存依赖约减技术是利用 $\prec_{d^*}$ 关系的传递性实现冗余访存依赖的过滤，因此被称为传递约减法 (Transitive Reduction, TR) [105]。对于访存事件 e_i, e_j, e_k，如果我们已知 $e_i \prec e_j$, $e_j \prec e_k$，那么对 $e_i \prec e_k$ 进行记录就是多余的：只要 $\prec$ 满足传递性，$e_i \prec e_k$ 就可以从 $e_i \prec e_j$ 和 $e_j \prec e_k$ 推导得到。可以证明，对于任意偏序关系，都存在唯一最简的偏序满足传递不可约，且这一最小偏序可以通过迭代删除传递冗余的依赖关系得到。TR 技术在运行时维护当前访存依赖的传递关系 $\prec_{d^*} = tr(\prec_d \cup \prec_{po})$。对于每一个捕获的访存依赖 $e_i \prec_d e_j$，都检查 $e_i \prec_{d^*} e_j$ 是否成立。若 $e_i \prec_{d^*} e_j$ 则将 $e_i \prec_d e_j$ 这一记录丢弃，否则记录 $e_i \prec_d e_j$ 并更新 $\prec_{d^*}$ 中的传递关系。在并发系统或分布式系统中，传递关系可以用向量时钟 (vector clock) [82] 在运行时维护，以实现访存依赖的在线约减。向量时钟不仅可以在软件层面维护 [105]，还可以在缓存一致性协议层面予以近似 [129]。

图 4.5a 展示了传递约减法 TR 实现的依赖约减。执行轨迹中的事件按照

$$\boxed{W(x)} \prec_{po} \boxed{W(y)} \prec_{po} \boxed{W(z)} \prec_{tot} \boxed{R(z)} \prec_{po} \boxed{R(y)} \prec_{po} \boxed{R(x)}$$

顺序发生，因此 x, y, z 三个变量之间存在数据依赖。但是我们只需记录一个依赖关系 $\boxed{W(z)} \prec_d \boxed{R(z)}$，就能根据偏序关系的传递性推导出其他数据依赖，如

$$\boxed{W(x)} \prec_{po} \boxed{W(z)} \prec_d \boxed{R(z)} \prec_{po} \boxed{R(x)},$$

因此 TR 在此例子中能在仅记录一对事件的前提下得到满足一致性的访存依赖。

传递不可约 (transitively irreducible) 的执行轨迹并不意味着它是满足一致性的最小执行轨迹。图 4.5b 包含了传递不可约的例子，执行轨迹中的事件按照

$$\boxed{W(x)} \prec_{po} \boxed{W(y)} \prec_{po} \boxed{W(z)} \prec_{tot} \boxed{R(x)} \prec_{po} \boxed{R(y)} \prec_{po} \boxed{R(z)}$$

顺序发生 (本例中按 x, y, z 的顺序读出共享变量的数值)，因此 $\boxed{W(x)} \prec_d \boxed{R(x)}$, $\boxed{W(y)} \prec_d \boxed{R(y)}$, $\boxed{W(z)} \prec_d \boxed{R(z)}$ 形成“平行”的依赖关系，无法用传递性互相推导得出。但若我们人为地添加 $\boxed{W(z)} \prec_d \boxed{R(x)}$ 的依赖关系 (在一致的 $\prec_d$ 中添加任意的 $e_i \prec_d e_j$，只要保持 $\prec_{d^*}$ 仍是一个合法的偏序，就不会影响 $\prec_d$ 的一致性)，则上述三个依赖均可以被传递约减。

然而，求解满足一致性的最小执行轨迹已被证明是 NP-完全问题 [76]。为进一步实现在线的访存依赖约减，正则传递约减法 (Regulated Transitive Reduction, RTR) 使用启发式的算法，在访存依赖 $e_i \prec_d e_j$ 被捕获时，试图寻找另一个不引发矛盾、更“强”的依赖关系替代 $e_i \prec_d e_j$。具体而言，其寻找事件 e'_j 满足: (1) e'_j 与 e_j 在同一线程且在 e_j 之前发生，即 $e'_j \prec_{po} e_j$; (2) 依赖 $a \prec b'$ 仍然能保证 $\prec_{d^*}$ 是合法的偏序。由于 $a \prec b'$ 在传递关系上能推导 $a \prec b$，因此 RTR 记录 $a \prec b'$。

对于图 4.5b 中的例子，RTR 在获得访存依赖 $\boxed{W(z)} \prec_d \boxed{R(z)}$ 后，使用 $\boxed{W(z)} \prec_d \boxed{R(x)}$ 对其进行替代 (具有更强的顺序约束且不会引发矛盾)，并利用 TR 技术实现访存依赖的约减。

研究挑战. 尽管 TR 和 RTR 都能实现有效的访存依赖约减，但在运行时维护事件之间的传递关系是非常耗时的操作。对于一般的有向无环图，求解其传递闭包 (transitive closure) 目前已知的最坏情况下最优的算法是将其归约到矩阵乘法，因而具有 $O(n^\omega)$ 的时间复杂性 (目前已知最好的 $\omega = 2.376$ [38])。在分布式系统中，通常线程数远小于事件数，因此维护 Lamport 向量时钟 [82] 能实现更有效的传递性维护。然而，维护向量时钟需要 $O(|T|)$ 的时间 (且已被证明此复杂性为最优下界)。尽管利用并发程序共享内存访问的局部性能大幅降低这一开销 [55]，但其有效维护要么需要使用定制化的硬件 [130]，要么需要付出远远高于访存依赖追踪的开销 [28, 79]。

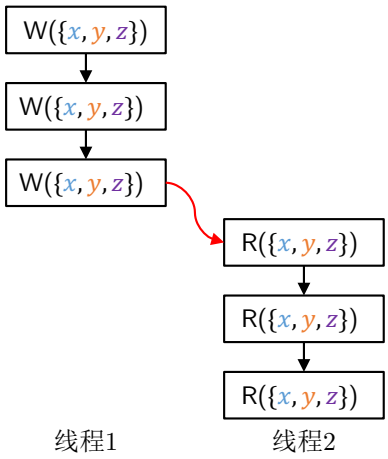

图 4.6 将变量 x, y, z 组合为变量组 $\{x, y, z\}$ 以实现访存依赖约减。

因此，在已有的软件系统上应用 TR 或 RTR 实现访存依赖约减，其带来的简化性优势将同时导致高效性的大幅降低。如何在不降低高效性的前提下实现访存依赖的在线约减 (如实现 TR 或 RTR 的依赖约减效果) 是主要的技术挑战。

解决思路. 我们发现，利用并发程序具有的空间-线程局部性对变量进行分组，在分组合适的前提下，能不引起较多运行时开销并实现 TR 和 RTR 的访存依赖约减效果。

首先，TR 和 RTR 能够实现访存依赖约减的基础是并发程序在执行过程中会访问多个共享变量，而其中一些变量的依赖可以用其他依赖 (或其变形) 推导得到。如果并发程序仅有一个变量，即 $|X_s| = 1$，则只要过滤掉具有线程局部性的事件，访存依赖 $\prec_d$ 就是传递不可约的，而现有的访存依赖追踪算法通常都能实现此类过滤 [28, 79]。

TR 和 RTR 实现访存依赖约减的主要原理是识别出了并发程序中常见的两种访存依赖可约减模式，如图 4.5 所示。图 4.5a 展示了 TR 实现依赖约减的模式：识别出“传递可约”的事件，在已经存在 $e_i \prec_{d^*} e_j, e_j \prec_{d^*} e_k$ 的前提下实现依赖 $e_i \prec e_k$ 的约减；图 4.5b 则展示了 RTR 实现依赖约减的模式：将“平行”的依赖关系使用一个更强的依赖关系替代，使得平行的依赖关系满足传递可约减的条件。

我们发现，若传递可约或平行可约的依赖关系产生于两个线程之间 (对应了图 4.5 中的情形)，则可将变量 x, y, z 组合为一个变量组 $\{x, y, z\}$，并以变量组为单位进行访存依赖追踪[1]，以同时实现 TR 和 RTR 的访存依赖约减效果，

[1]无论对地址空间中的变量进行怎样的分组，我们只要记录变量组上满足一致性的访存依赖，就一定能获得对任意变量满足一致性的访存依赖。

如图 4.6 所示。此时，无论对 x, y, z 的操作以何种顺序发生，其访问都相当于对变量组 $\{x, y, z\}$ 的访问。在线程 2 中首次访问变量组中的变量，将会产生访存依赖，但随后对变量组中的其他变量进行访问都将被视作对该变量组的线程局部访问，不会产生依赖记录，从而实现 TR 和 RTR 的访存依赖约减效果。

这类可约减的情形是符合现实并发程序的共享内存访问模式的：并发程序使用共享内存实现线程间通信时，一个线程通常首先准备一系列数据，然后使用同步操作与另一线程同步，之后另一线程再对这些数据进行操作。这恰好对应了 TR 和 RTR 实现访存依赖约减的场景。因此，我们考虑用变量分组的技术实现访存依赖的在线约减。

现有研究工作已对变量分组进行了尝试 (相当于将空间上连续的变量指派为同一锁)，例如 Octet [28] 将属于同一对象的变量分为同一组以实现依赖数量的约减，但目前的研究通常局限于使用固定的策略进行静态变量分组。根据第 2.2 节中证实的并发程序具有空间-线程局部性的实验结果，我们发现现有基于静态分组的技术并不能充分利用现实并发程序共享内存访问具有的空间-线程局部性。因此，我们提出在运行时将地址空间进行动态可调整的区间划分，并于运行时维护这一区间划分以使每一区间中的变量具有空间-线程局部性，从而将区间中的变量视作一个整体进行访存依赖追踪，以实现访存依赖数量的大幅减少。

4.2.2 二分组协议

利用空间-线程局部性实现访存依赖约减. 假设并发程序运行在大小为 M 的地址空间 (M 为内存地址的最大值)，程序中的每一全局变量 $x \in X_s$ 均有一个在区间 $[0, M)$ 内的地址。在不引起歧义的前提下，我们既用 x 表示一个共享变量，也用 x 表示它的地址。

为了实现空间-线程局部性的利用，我们将变量进行区间分组：将 X_s 划分为 k 个不相交的变量组 $\mathcal{P} = \{G_1, G_2, \ldots G_k\}$，使用映射函数 $f : [0, M) \to [k]$ 将地址 x 映射到其对应的变量组 $G_{f(x)}$。当线程访问 x 时，我们认为其访问了 $G_{f(x)}$ 中的所有变量。换言之，在获取访存依赖时，我们不再将 X_s 作为共享变量集合，而是将变量组 $G_i \in \mathcal{P}$ 视作共享变量，对 x 的读写将视作对 $G_{f(x)}$ 的读写——只要两个事件属于同一变量组、发生在不同线程且至少有一个是写事件，就认为其为冲突事件。

将变量进行分组并按组获取访存依赖不会影响访存依赖的一致性。访存依赖满足变量组意义上的一致性，当且仅当对于任意的事件对 e_i, e_j，若 $e_i.t \neq e_j.t$, $(e_i.rw = w) \vee (e_j.rw = w)$ 且 $G_{f(e_i.x)} = G_{f(e_j.x)}$，则有

$$e_i \prec_{d^*} e_j \Leftrightarrow e_i \prec_{tot} e_j$$

成立。因此，对于任意实际冲突的事件对 $e'_i \asymp e'_j$，根据其定义有 $e'_i.x = e'_j.x$，必有 $G_{f(e'_i.x)} = G_{f(e'_j.x)}$，因此 $e'_i \prec_{d^*} e'_j \Leftrightarrow e'_i \prec_{tot} e'_j$ 成立。

将变量进行区间分组以利用空间-线程局部性实现高效访存依赖获取的技术已得到广泛应用 [28, 52, 69, 79, 132, 144]。变量分组属于第 3.2.1 节中讨论的锁指派技术中的区间指派技术，常见的区间有属于同一对象的数据域 [28]、连续一段固定长度的地址 [79]、物理内存页面 [52] 等。此类分组技术能有效减少元数据维护的数量 (对于一个分组只需维护一份元数据)，通常能实现运行时开销的减少和部分访存依赖的约减。

然而，这类技术仅仅发挥了空间-线程局部性的少部分潜能。目前的技术为了防止产生不代表数据依赖的访存依赖 (若 x, y 被两个线程分别独立访问但被分为同一组，则会导致很多冗余的依赖)，通常在区间大小上较为保守。然而在第 2.2 节对空间-线程的实证研究中，对于一些程序，只有使用较大的分组 (如 2^{20} 字节) 才能实现较优的访存依赖约减效果。尽管我们知道将地址空间进行划分能实现空间-线程局部性的利用从而约减访存依赖的数量，但由于不同程序在不同时刻访问共享内存的模式不同，如何准确地划分地址空间以实现访存依赖的约减就成为一个难题。

动态变量分组技术. 目前使用启发性经验规则进行区间划分的技术都使用一般性准则指导划分 (如实现锁的对象指派、物理页指派等) 而无法根据并发程序访问共享内存的实际行为模式进行调整。为了克服这一局限，我们创造性地提出在程序运行时动态维护地址空间的划分，使每一区间中的共享变量访问均满足空间-线程局部性，并在这一假设被违反时重新调整划分。

对于地址空间 $[0, M)$，我们在运行时维护其划分 $\mathcal{P} = \{G_1, G_2, \dots G_k\}$，且满足如下性质：

1. 每一变量组可以视为一个左开右闭区间 $G_i = [l_i, r_i)$，对于地址 x, $x \in G_i$ 当且仅当 $l_i \le x < r_i$。由于 $\mathcal{P}$ 是地址空间 $[0, M)$ 的划分，因此 $l_1 = 0$, $r_k = M$；
2. 各个变量组首尾相接，即对于 $1 \le i < k$, $r_i = l_{i+1}$，这两点共同保证了 $\mathcal{P} = \{G_1, G_2, \dots G_k\}$ 是 $[0, M)$ 的一个划分。

理论上说，我们并不需要维护变量的区间分组 (可以维护 X_s 的任意划分)，在地址空间上不连续的变量同样可能具有完全相同的线程间数据依赖模式，因此将它们分为同一组能实现更佳的访存依赖约减效果。然而，我们必须在维护分组的代价和访存依赖约减的效果之间进行权衡。我们选择区间分组是因为它不仅能自然地反映并发程序的空间-线程局部性 (从而实现访存依赖的约减)，

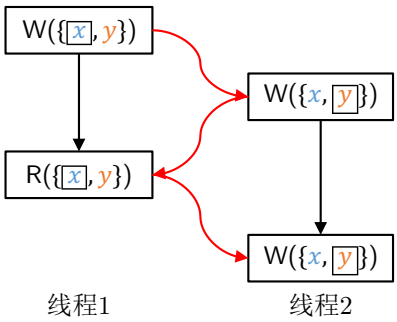

图 4.7 不合适的变量分组 $\{x, y\}$ 导致冗余的依赖。

还能被高效地维护 (我们必须在共享内存访问时根据变量的地址高效地计算出 $f(x)$，否则将会引起巨大的运行时开销)。

我们使用下述二分组协议对动态分组进行维护。初始时，我们乐观地认为所有变量都具有空间-线程局部性，并将所有变量指派到同一分组。在程序运行过程中，我们希望保证每一分组中的共享内存访问都具有空间-线程局部性：在一段时间内，在同一组中的变量要么被某个线程独占访问，要么在多个线程之间以只读的方式共享。在程序运行时，我们使用获取的访存依赖对“分组中访存事件具有空间-线程局部性”这一假设进行验证，并在假设被违反 (即发现不合适分组) 时对分组进行修改，以逐步使得对地址空间的区间划分符合程序运行时的共享内存访问模式，实现访存依赖的约减。

初始分组. 初始时，我们假设并发程序不存在任何线程间的数据依赖：其只包含一个线程，它以独占的方式访问所有共享内存变量。在此情形下，并发程序等同于串行程序，无需记录任何访存依赖。为了将元数据的开销减到最小，初始分组 $\mathcal{P} = \{[0, M)\}$ 仅包含一个区间。无论 X_s 中包含多少变量，$f(x) = 1$ 都映射到同一个变量组 $G_1 = [0, M)$。在实际中，初始划分 $\mathcal{P}$ 将 $[0, M)$ 划分为若干长度为 2^{20} 字节的大区间，以在充分利用空间-线程局部性的同时避免程序启动后频繁的分组调整。

在程序的执行过程中，“不存在线程间数据依赖”的假设随着多个线程开始访问共享内存而不再成立。我们使用访存依赖识别出不再具有空间-线程局部性的分组，并用二分的方式对其进行调整，描述如下。

不合适分组的检测. 非形式地说，“不合适分组”表示一个共享变量组 $G_i \in \mathcal{P}$，其中的变量不满足空间-线程局部性。图 4.7 展示了不合适分组的例子。图中线程 1 仅访问共享变量 x，线程 2 仅访问共享变量 y，因此不存在线程间的数据依赖。但如果不合适地将 x, y 指派为同一分组，则两个线程都会访问同一变量组，导致大量冗余的访存依赖。

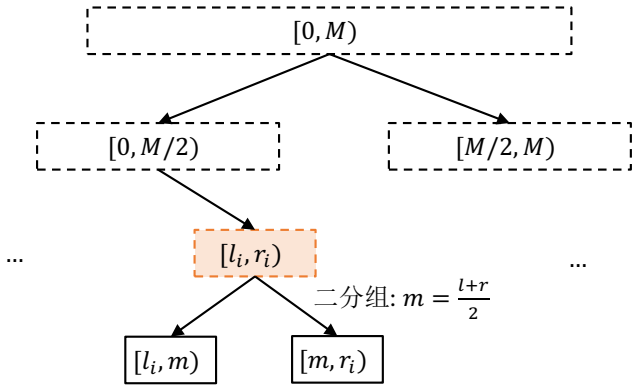

图 4.8 二分组的历史构成一棵二叉树。

我们定义此类由不合适分组导致的不包含实际数据依赖的访存依赖为“假依赖”(false dependence)。对于 $e_i \prec_d e_j$，我们按如下规则判定其为假依赖：我们首先找到 $\prec_d$ 中线程 $e_i.t$ 发生的最近一次对 $e_i.x$ 访问的事件 e ($e.x = e_i.x$, $e \prec_{po} e_i$ 且 $\prec_d$ 中不存在 e' 满足 $e'.x = e_i.x$ 且 $e \prec_{po} e' \prec_{po} e_i$)。然后检查是否对于所有 $e \prec_{po} e^* \prec_{po} e_i$ 都有 $e^*.x \neq e_j.x$。若此条件满足则 $e_i \prec_d e_j$ 被判定为假依赖。易见图 4.7 中的依赖均满足此定义并被判定为假依赖。

变量组 G_i 上的假依赖表明存在变量 $x, y \in G_i$，其分别被两组线程并发访问，但却被包含在同一变量组中。因此，若在 G_i 上频繁检测到假依赖 (数量超过某一阈值) 则预示着我们需要对该分组进行调整，使得 x, y 不再属于同一区间，维持“变量组中变量具有空间-线程局部性”的性质。

运行时分组调整. 当确认 $G_i = [l_i, r_i)$ 由于假依赖的存在需要调整后，我们就将 G_i 分解为更小的子区间，使得假依赖的变量 x, y 移动到两个不同的变量组中。

为了权衡分组维护的高效性和实现分组调整的有效性，我们的运行时分组调整总是在变量组 $G_i = [l_i, r_i)$ 区间的中点进行区间分裂，即分为两个新的变量组 $[l_i, m)$ 和 $[m, r_i)$，其中

$$m = \frac{l_i + r_i}{2}$$

是区间 G_i 的中点。对 $\mathcal{P} = \{G_1, G_2, \ldots, G_k\}$ 中的变量组 $G_i = [l_i, r_i)$ 进行二分组将得到新的变量组 $\mathcal{P}' = \{G'_1, G'_2, \ldots, G'_{k+1}\}$，其中

$$G'_1 = G_1,\ G'_2 = G_2,\ \ldots,\ G'_i = [l_i, m),\ G'_{i+1} = [m, r_i),\ \ldots,\ G'_{k+1} = G_k。$$

由于我们总是在区间中点进行划分，这一访存依赖约减技术也称为二分组 (bisectional coordination) 协议。在区间中点二分并不能保证分离导致假依赖的共享变量 (若 x, y 恰好同时处于 $[l_i, m)$ 或 $[m, r_i)$ 的区间内，则未来可能继续产生关于 x, y 的假依赖)。我们选择在区间中点进行划分是因为：(1) 直接按照产

生假依赖的变量 x, y 进行区间分裂并非最优策略 (由于访存依赖按照变量组获取，在一段时间内会发生多次对变量组中不同变量的访问，因此 x, y 并不能代表这段时间内的共享内存访问模式)；(2) 在区间中点进行分裂能保证分离两个持续产生假依赖的变量至多只需 $\lceil \log_2 M \rceil$ 次二分组，其开销在可接受的范围内；(3) 在区间中点进行二分能极大地简化实现的难度，使其在乐观锁的基础上被高效实现成为可能。

对地址空间进行连续多次二分组的过程如图 4.8 所示：虚线区间表示已进行过二分组的区间。图中区间 $[l_i, r_i)$ 被二分组，形成两个子区间 $[l_i, m)$ 和 $[m, r_i)$。经过一系列的二分组操作之后，二分组的历史记录形成一棵二叉树，树中每个非叶子结点都代表一个过去被二分组的区间，树中叶子结点的集合恰好构成地址空间的当前区间划分 $\mathcal{P}$，类似于一棵线段树 (segment tree)。

二分组协议与访存依赖约减. 二分组协议是一个“谨慎乐观”的策略：它首先乐观地假设并发程序的共享内存访问具有相当程度的空间-线程局部性。随着并发程序的执行，这一假设随着访存依赖的获取被反复验证。除非我们找到证据 (获取足够数量的假依赖) 表明我们对一段连续内存的空间-线程局部性假设不成立，相当数量的变量在一段时间内都将处于同一个分组，从而实现访存依赖的约减。

我们以数组为例定性分析二分组协议实现访存依赖约减的有效性。通常，开发者在编写并发程序时，只对数组使用有限的一些并发模式：

1. *只读共享*：整个数组在初始化时写一次，之后所有其他线程以只读方式访问这一数组。静态查找表等数据结构属于这一类型。
2. *分组访问*：将数组分为若干连续区域，将连续区域分配给不同线程计算，最后合并结果。科学计算或数据分析类并发程序常用此数组并发访问模式。
3. *随机访问*：所有数组元素在各个线程均有随机 (或近似随机) 的访问模式。并发数据结构 (如散列表、树结构) 通常属于这一类型。

对于使用静态变量分组的技术 [28, 52, 69, 79, 132, 144]，若将整个数组视为一个整体指派为同一锁 [28]，则无论是否写入相同的数组元素，所有对该数组的写操作都将被串行化。这与数组共享/并发数据结构的设计背道而驰，会在 (2) 和 (3) 类的并发模式将产生大量冗余的假依赖。若对每一数组元素分别记录访存依赖 [121]，虽然在随机访问并发模式下不会产生冗余的假依赖，在 (1) 和 (2) 类并发模式下将产生大量平行可约减的依赖。

二分组协议则可以灵活处理这几类情形。对于模式 (1)，运行时数组中的所有元素都处于同一区间中。由于这一并发模式实际具有空间-线程局部性，在访存依赖获取的过程中不会产生假依赖，因而不会导致区间二分。若系统中有 T 个读线程对数组进行只读访问，则只产生 T 个访存依赖。对于模式 (2)，若分

属于两个线程的两个共享变量被指派到同一区间，则会产生假依赖导致区间分组调整。分组调整会不断进行，直到每一区间中的变量均由同一线程独占访问。对于模式 (3)，由于访问模式随机，只要存在两个数组元素处于同一区间，假依赖就总是会发生。在运行时间足够长的前提下，区间会不断进行分组调整，直到退化为每一区间恰好包含一个数组元素 (类似于变量指派)。

可见，二分组协议能自动根据数组中共享变量的实际访问模式对变量分组进行调整，在粗/细粒度的区间划分之间进行切换。此外，上述论证不仅对数组成立，对任意连续的内存 (如对象中的域、静态变量等) 也成立。

综上所述，二分组协议通过自动检测共享内存访问的模式，“自适应”地选择适合该访问模式的区间划分，使得每一区间中的变量具有空间-线程局部性从而实现访存依赖的约减。

4.3 系统实现与实验评估

4.3.1 系统实现

4.3.1.1 基于乐观锁的访存依赖追踪技术 RWTrace

我们在 LLVM [3] 编译器工具链上实现了基于乐观锁的访存依赖追踪工具 RWTrace。RWTrace 能实现 C/C++ 程序的访存依赖追踪，我们已将 RWTrace 的实现公开发布[1]。

程序插装框架. 我们首先使用 clang 工具对应用程序进行编译，生成 LLVM 中间代码 (IR) 并进行插装。插装框架实现为 LLVM 的指令优化器 (optimization pass)，它不仅能实现对 load/store IR 指令的插装，还能实现对库函数调用的拦截 (hooking)。

为了防止程序插装 (尤其是对 load/store 指令的插装) 影响编译器优化 (在指令前后插入访存依赖追踪相关的代码会产生额外的数据流/控制流，导致在未插装代码中可以实现的优化不再有效)，我们首先将所有 C/C++ 源文件分别编译成 LLVM 中间代码 (`.bc` 文件)，然后使用 `llvm-link` 将应用程序所有中间代码进行链接并进行全局优化，得到一个合并、优化的中间代码 (直接对其进行链接能得到应用的可执行文件)，再对合并后的中间代码进行插装：

1. 对于 load 指令，乐观锁可能进行两次读操作，因此我们将其复制两份，并且在第一次读之前调用 `_before_tryread` 进行尝试，在之后调用

[1]开源项目地址：`http://github.com/jiangyy/rwtrace`。

_after_tryread，根据尝试的返回值决定是否需要第二次读操作。在第二次读指令之前插入对 _before_read 的调用、之后插入对 _after_read 的调用以实现同步操作包裹的共享内存读和访存依赖追踪。

2. 对于 store 指令，我们在指令前插入对 _before_write 的调用，在指令后插入对 _after_write 的调用。
3. 对于调用 pthread_create 创建线程的 call 指令，设线程入口地址为 ptr，参数为 arg，我们将函数调用的目标进行拦截，并替换为插装框架中的函数，以完成如下功能：(1) 创建一个新线程；(2) 与新线程同步并记录线程创建和线程启动事件之间的 happens-before 关系；(3) 新线程调用 ptr(arg) 执行，保持与 pthread_create 函数行为的一致性。
4. 对于其他调用 pthread 相关函数的 call 指令 (如互斥锁、条件变量、线程 join 等操作)，我们对其进行与 store 指令类似的包裹以便能在 RWTrace 框架上实现基于访存依赖的动态分析。

为实现具体的访存依赖获取或动态分析技术 (如 RWTrace、基于 happens-before 关系的数据竞争检测等),只需向插装框架提供 _before_read、_after_write 等函数的实现。这些实现最终和插装过的应用代码统一链接成一个 LLVM 中间代码文件，包含所有中间代码的文件将经过再次优化 (从而插装相关的代码将被内联并优化)，生成能够实现动态分析的可执行文件。

最后，使用 RWTrace 并不需要对应用的源代码或编译脚本 (如 Makefile) 作出修改。我们使用一个自定义的脚本替代 CC，其使用包裹了 clang 的脚本，在内部两次调用 clang，一次使用原参数进行编译，另一次输出 LLVM IR 以供插装使用。这一技术同样用于我们实现 C/C++ 程序加速编译的技术 [140]。

追踪访存依赖. 我们按照第 4.1 节提出的算法描述对 load 和 store 指令进行的插装，算法部分未能列举的实现细节描述如下：

1. 为了利用空间-线程局部性、减少元数据维护的开销，我们使用静态的锁指派。在程序开始时分配 2^{20} 个锁对象，对于共享内存地址 x，我们将其映射到编号为 $(x$>>6)&0xfffff 的锁对象[1]。在锁对象充足 (不使用超过 2^{26} 字节内存) 的前提下，相当于将内存地址位于连续 64 字节内的变量指派为同一锁。64 字节是典型计算机系统 L1 缓存线的宽度。如并发程序中不存在假共享缺陷，使用这一大小能减少元数据的开销和记录依赖的数量。
2. 在 RWTrace 的算法描述中，每一线程 t 都需要在本地维护每一变量 x (实际实现时为 x 对应的锁对象) 上次发生在线程 t 的访存事件 $C_t(x)$。C_t 可

[1] >> 为二进制位右移运算，& 为二进制与运算。

以用散列表在 $O(1)$ 时间内实现，但其仍然会引起相当可观的时间和空间开销。为了获得更优的性能，我们利用并发程序的线程局部性，使用一个有限大小的缓存来作为 C_t。

线程 t 的 C_t 实现为大约包含 $m \approx 10^4$ 个元素的直接映射缓存 (我们取略小于 10^4 的素数作为模数以取得最佳直接映射缓存的命中率)。编号为 i 锁对象的版本号存储在编号为 $i \bmod m$ 的缓存项中。在缓存命中时，只需少量线程本地指令即可实现缓存的读取和更改。

3. 维护读事件映射 $r_x(t)$ 的数据结构需要支持高效并发查找和修改。我们使用闭地址散列表实现 r_x，并使用链表解决散列冲突。由于闭地址散列表的散列函数不会变化，其每一散列桶中的链表是单调递增的，因而保证了一旦 $t \in r_x$，后续所有对 t 的查找都一定成功。此外，由于闭地址散列表所有内存分配和释放都是固定长度 (所有散列表的长度都相同、每一键-值对数据结构的大小也相同)，因此可使用环状缓冲区实现均摊 $O(1)$ 时间的内存分配和释放。

4.3.1.2 在 RWTrace 基础上实现的二分组协议

第 4.2 节描述了实现在线访存依赖约减的二分组协议。为了在追踪访存依赖的同时实现二分组协议，必须解决以下技术挑战：

1. 在共享内存访问时确定变量 x 对应的分组编号 $f(x)$。由于分组在运行时不断变化，必须实现 $f(x)$ 的高效动态维护；
2. 根据定义，判定 $e_i \prec_d e_j$ 是否是假依赖需要对访存依赖记录进行扫描，必须减少这一算法的开销。

若使用朴素的算法实现二分组协议 (使用二叉树查找确定变量分组、扫描访存依赖记录检测假依赖)，相比于乐观的访存依赖追踪算法 RWTrace，将会引起 5-10 倍的额外运行时开销，此时二分组协议虽然能够实现有效的访存依赖约减，但却带来了实际中无法容忍的开销。

因此我们应用了一系列技术在 RWTrace 的基础上高效地实现了二分组协议，描述如下 (其评估结果在第 4.3.2 节展示)。

确定变量对应的分组. 由于并发程序具有海量的共享内存访问，虽然绝大部分具有空间-线程局部性而无需记录与之相关的访存依赖，但在每次共享内存访问时，都必须确定变量 x 对应的变量组 $f(x)$，因此计算 $f(x)$ 的时间开销对二分组协议的高效性是至关重要的。

给定共享变量 x，$f(x)$ 返回唯一的区间 $G_i = [l_i, r_i) \in \mathcal{P}$ 满足 $x \in G_i$。如图 4.8 所示，二分组的历史构成一棵二叉树。一个直接计算 $f(x)$ 的算法是从树根

出发，根据 x 在区间 $[l_i, r_i)$ 中的位置 ($x < \frac{l_i+r_i}{2}$ 或 $x \geq \frac{l_i+r_i}{2}$) 在子树中进行递归查找，直到找到包含 x 的叶子结点。由于每次二分组都将区间的长度减少一半，因此二叉树的最大深度为 $\lceil \log_2 M \rceil$，这一算法的时间复杂性为 $O(\log M)$。虽然表面上看这是一个优秀的算法，但由于在每次访存时都需要执行查找操作，引入 $O(\log M)$ 的时间开销会大幅降低访存依赖获取的高效性。

现实程序中二分组的数量通常较少，我们使用缓存对变量分组查找进行优化，使得大部分查找能在 $O(1)$ 时间内完成。我们使用一个全局共享的并发散列表对变量的分组进行缓存。散列表将内存地址映射到其对应分组，仅当缓存缺失或从缓存中读取到的区间不再合法 (已经被分裂为小区间) 时，我们才使用 $O(\log M)$ 的查找算法确定 x 所对应的变量组。我们使用类似 RWTrace 中的事件排序技术实现了无锁的查找，以保证在查找成功时能在不使用同步或原子操作的前提下在 $O(1)$ 时间内完成。

实现高效的假依赖检测. 如按照假依赖的定义直接进行检测，在访问变量组 $G_i = [l_i, r_i)$ 时记录访存依赖 $e_i \prec_d e_j$ 后，需要扫描 G_i 的历史记录，在线程 $e_i.t$ 的记录中查找是否有对共享变量 $e_j.x$ 的访问。维护、扫描访存依赖记录的开销是巨大的。

为了实现高效的假依赖检测，我们使用 Bloom Filter [26] 对假依赖进行近似检测。首先，假依赖的检测可以通过在每一线程维护变量组 G_i 的共享变量访问集合 VS 实现，在发生与 G_i 相关的依赖后将集合清空 ($VS \leftarrow \varnothing$)。当发生未产生访存依赖的事件 e 时，则将相应的变量添加到集合中 ($VS \leftarrow VS \cup \{e.x\}$)。在依赖 $e_i \prec_d e_j$ 被捕获时，查询 $e_i.t$ 中的 VS 即可实现假依赖的检测。

进一步地说，我们只需要写后读 ($\prec_{w \to r}$) 依赖就能实现假依赖的有效检测 (平均意义上，假依赖至少有一半是写后读依赖)，因此只需在 VS 中维护发生的写事件，并仅在写后读依赖发生时检测假依赖。基于第 2.2 节中证实的空间-线程局部性 (并发程序大部分共享内存访问是具有线程局部性的读事件)，这一简化在不损失假依赖检测精度的前提下大幅减少了更新 VS 的事件数量，提高了二分组协议的实现效率。在此基础上，我们为每一变量组 G_i 维护 256 字节的 Bloom Filter 维护 VS，可保证对其的更新和查询能在 $O(1)$ 时间内完成。

当一个分组 G_i 累积达到 20 个假依赖时，我们对其进行分组。实验表明，即便对于完整程序的运行，二分组的数量通常仍是远少于共享内存访问数量的，因此调节这一阈值不会大幅改变实验结果。

应对冷启动. 二分组协议中初始时乐观地将共享内存中的全部变量分为一组 (即初始划分 $\mathcal{P} = \{[0, M)\}$ 仅包含一个区间)，因此对于实际不存在并发性的串行程序能实现最少的元数据开销。然而二分组协议通常用于实际并发程序的访存

表 4.1 评估 RWTrace 和二分组协议所用基准程序及其设置。

实验程序		任务描述	代码行数	实验设定
桌面计算	aget	并行数据获取	2.5K	16 线程，64MB 数据
	pfscan	并行文件扫描	1.1K	16 线程，640MB 数据
	pbzip2	并行压缩	1.9K	16 线程，64MB 数据
科学计算	ocean	海洋模拟	9.1K	16 线程，1026×1026 网格
	water	分子动力学模拟	3.6K	16 线程，10 步，10^3 个分子
	fft	快速傅立叶变换	1.4K	16 线程，2^{24} 数据点
	radix	基数排序	1.9K	16 线程，2^{24} 元素
	fluid	流体动力学模拟	1.2K	16 线程，10 帧，100KB 输入
	qsort	快速排序	0.9K	16 线程，2^{24} 元素
	x264	视频编码压缩	37K	16 线程，128 帧，640×360 分辨率
服务器	knot	HTTP 服务器	1.2K	16 线程，2^{14} 请求
	apache	HTTP 服务器	339K	16 线程，2^{14} 请求

依赖约减。对于并发程序而言，并发运行的多个线程必将导致过于乐观的分组上假依赖的累积。这一情形在程序启动时尤为明显 (多个线程同时访问内存，由于初始分组过于乐观从而导致大量假依赖)。对于相邻但被不同线程互斥访问的变量 x, y，需要记录 $O(\log_2 M)$ 个访存依赖后才能将其分离。

为了避免这一“冷启动”的情形，我们初始时将地址空间分为 2^{20} 字节 (1MB) 大小，这一数值既足够大而能反映具有空间-线程局部性的内存，同时又减少了冷启动带来的时间和元数据开销。

4.3.2 实验评估

4.3.2.1 实验设计

实验对象与实验环境. 我们使用桌面计算、科学计算和服务器三类共 12 个基准程序对 RWTrace 及在其基础上实现的二分组协议进行评估。实验对象来自基准程序集 PARSEC [24], SPLASH-2 [128] 和已有访存依赖追踪相关的研究工作 [85]。实验对象、任务类型、代码行数和输入设置在表 4.1 中描述。

所有实验结果均在一台有 4 个 6 核心 Intel Xeon X7460 处理器 (共 24 个处理器核心) 和 64GB 内存的服务器上运行得到。对每个实验对象都运行 10 次，取统计数据的平均值予以展示。

表 4.2 RWTrace 与 MutexLock 和 CREWLock 对比实验结果。

实验程序	时间	RWTrace (1)		RWTrace (2)		MutexLock		CREWLock	
		时间	依赖	时间	依赖	时间	依赖	时间	依赖
aget	4.53	4.59	3.12K	4.60	3.19K	4.89	2.84K	4.53	3.13K
pfscan	0.99	7.76	0.15K	18.45	0.21K	30.51	0.15K	28.61	0.12K
pbzip2	1.99	2.01	0.66K	2.04	1.03K	2.07	1.32K	2.04	0.67K
ocean	1.61	1.87	24.7K	2.80	45.7K	54.56	49.6M	14.63	25.2K
water	0.50	1.10	822K	2.67	1.09M	21.86	74.0M	6.97	872K
fft	3.36	4.28	2.34K	5.19	2.56K	5.52	3.09K	5.48	2.34K
radix	0.79	0.91	2.34K	1.17	2.35K	8.46	9.17M	3.30	2.33K
fluid	0.53	1.46	11.5K	2.49	20.2K	111.17	119M	28.14	11.7K
qsort	1.34	3.30	761K	5.56	1.56M	3.54	1.66M	3.35	783K
x264	4.26	29.41	3.75M	58.29	5.17M	69.25	84.4M	58.36	5.78M
knot	1.80	1.81	38.2K	1.83	79.6K	1.83	68.4K	1.83	38.6K
apache	3.29	3.28	3.50K	3.29	3.61K	3.28	3.22K	3.28	3.53K

乐观锁的实验评估. 我们对基于乐观锁的访存依赖追踪技术实现 RWTrace 进行实验评估。我们对以下每一技术分别运行每一实验程序，并对比其运行时间开销和访存依赖数量两方面的性能。所有技术均在同一框架下实现，并且使用相同的静态区间锁指派 (与第 4.3.1.1 节描述相同)：

1. RWTrace (1)，使用乐观锁技术仅追踪写后读 ($\prec_{w\to r}$) 和写后写 ($\prec_{w\to w}$) 依赖，因此只需在运行时维护 w_x。在 x86-TSO 内存模型下无需同步/原子操作或内存障碍。
2. RWTrace (2)，使用乐观锁技术追踪所有类型依赖 (写后读 $\prec_{w\to r}$、写后写 $\prec_{w\to w}$、读后写 $\prec_{r\to w}$)。r_x 和 w_x 都在运行时维护。
3. MutexLock，使用互斥锁串行化所有对指派到同一锁的共享内存访问，并记录所有类型依赖 (包括读后读依赖)。
4. CREWLock，使用 pthread 线程库中提供的读-写锁实现并发读、互斥写语义。由于允许并发读且我们的基准实现不使用并发数据结构维护读事件映射，从而只能获得写后读 ($\prec_{w\to r}$) 和写后写 ($\prec_{w\to w}$) 依赖。

二分组协议的实验评估. 我们在与 RWTrace 相同的插装框架和设定下实现了二分组协议 (均追踪写后读 $\prec_{w\to r}$ 和写后写 $\prec_{w\to w}$ 依赖，通过这两类依赖能经过

表 4.3 二分组协议实现的依赖约减与 RWTrace 和 MutexLock 的对比实验结果。

实验程序	访存依赖数量			二分	同步	访存
	BC (RWT / MutexLock)	RWT	MutexLock			
aget	7.40K (↓9.0% / ↓16%)	8.13K	8.82K	0	8.10K	39.9K
pfscan	116K (↓34% / N.A.)	175K	溢出	12	323K	9.82G
pbzip2	0.30K (↓55% / ↓76%)	0.67K	1.25K	28	0.54K	5.21K
ocean	27.1K (↓5.2% / ↓99%)	28.6K	57.4M	60	24.3K	138M
water	53.8K (↓97% / ↓99%)	1.69M	78.1M	52	99.0K	112M
fft	0.23K (↓90% / ↓99%)	2.35K	29.9K	4	0.23K	40.0M
radix	0.16K (↓93% / ↓99%)	2.34K	9.19M	34	0.56K	112M
fluid	9.52K (↓39% / ↓99%)	15.7K	170M	16	813K	463M
qsort	319K (↓55% / ↓79%)	706K	1.50M	72	206K	15.3M
x264	1.63M (↓91% / ↓98%)	18.9M	95.5M	954	20.3K	6.80G
knot	37.6K (↓0.5% / ↓45%)	37.8K	68.2K	18	0.03K	159K
apache	44.2K (↓79% / ↓98%)	214K	1.96M	89	32.3K	6.64M

执行重放推导出其他类型的访存依赖)，并评估二分组协议实现访存依赖约减的性能，将以下三类技术进行对比：

1. RWTrace (简称 RWT)，使用乐观锁的访存依赖追踪技术。
2. 二分组协议 (bisectional coordination，简称 BC)，在 RWTrace 基础上实现的空间-线程局部性敏感的动态区间分组维护。
3. MutexLock，使用互斥锁的访存依赖追踪技术，作为基准技术进行对比。

与 RWTrace 的实验评估类似，我们对每一需要评估的技术分别运行每一实验程序，收集程序的运行时间与访存依赖数量。

4.3.2.2 实验结果

乐观锁的实验评估结果. 对 RWTrace 的实验评估结果在表 4.2 中展示。RWTrace (1) 仅仅获取 $\prec_{w\to r}$ 和 $\prec_{w\to w}$ 两类依赖 (获得这两类依赖已足够实现准确访存依赖的推导)，由于使用了乐观锁技术，具有线程局部性的读事件不引入额外的同步/原子操作，因此 RWTrace (1) 在所有实验程序上都获得了最佳的性能。相比于需要串行化大量能够并行读操作的 MutexLock，RWTrace 实现了高达 209 倍的性能提升 (性能提升中位数为 6.7 倍) 和大幅的访存依赖数量减少 (由于

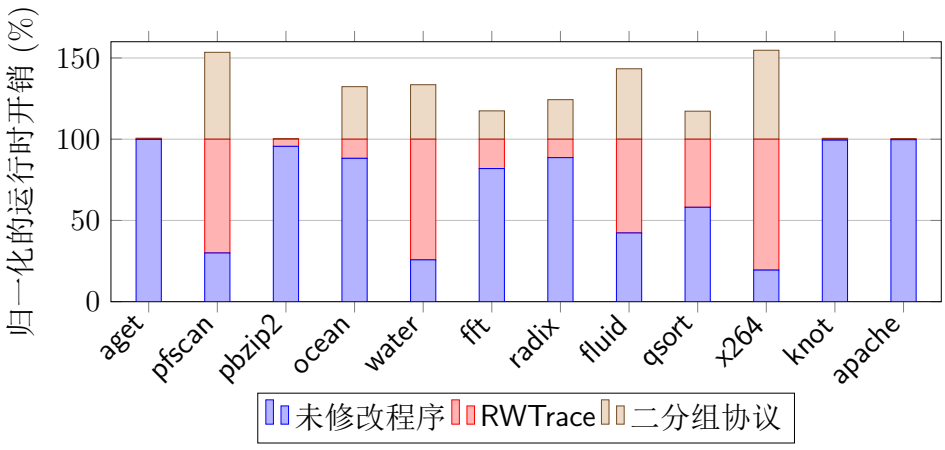

图 4.9 二分组协议相对 RWTrace 的运行时开销。

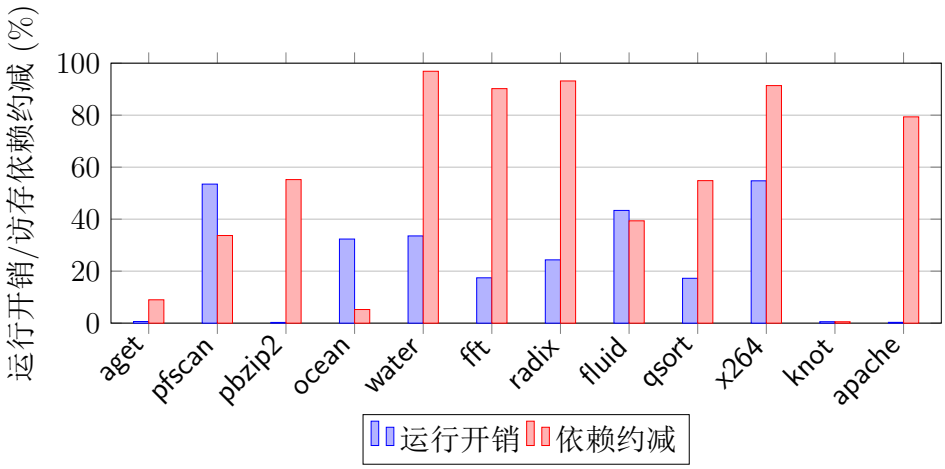

图 4.10 二分组协议实现依赖约减和运行时开销之间的权衡。

MutexLock 记录了冗余的读后读依赖)；相比于读-写锁 CREWLock，RWTrace 实现了高达 52 倍的性能提升 (性能提升中位数为 2.3 倍)。

如果使用 RWTrace (2) 追踪额外的 $\prec_{r\to w}$ 依赖，则需要付出额外的 0.3%-142% (中位数 39%) 的运行时开销。即便在这一设定下，RWTrace 仍在 11/12 个实验程序中取得了最佳的性能，仅在 qsort 程序中慢于读-写锁实现的 CREWLock，但 CREWLock 不具备追踪 $\prec_{r\to w}$ 依赖的能力，与之能力等同的技术是运行时开销远远小于 CREWLock 的 RWTrace (1)。因此，实验结果证实基于线程局部性和乐观锁的技术能有效降低访存依赖获取的运行时开销，实现即时、高效、准确的访存依赖获取。

二分组协议的实验评估结果. 二分组协议的实验评估结果在表 4.3 中展示。相比于使用静态变量分组的 RWTrace，二分组协议实现了 0.5-97% (中位数 55%)

的访存依赖约减；相对于 MutexLock，二分组协议实现了 16-99.99% (中位数 99%) 的访存依赖约减。比较两份能够实现在线依赖约减的工作 (它们都使用 MutexLock 作为比较对象) 中的数据，CARE [78] 实现了 11-99.6% (中位数 86%)、CLAP [74] 实现了 72-97.7% (中位数 91%) 的依赖约减，因此二分组协议实现的依赖约减更具优势。

对于将数据分组的科学计算类程序 (如 water, fft) 或对数据进行线性处理的程序 (radix)，二分组协议实现了超过 90% 的依赖约减；对于数据密集型的复杂类型共享内存访问应用如 qsort (递归排序) 和 x264 (流水线生产者-消费者)，二分组协议实现了 55%-91% 的约减。对其他类型的应用也实现了不同程度的依赖约减。比较表 4.3 中第 6 列的数值 (程序运行时执行同步操作的数量)，我们发现二分组协议已将访存依赖数量降低至与同步操作类似的量级。由于动态分析通常无法避免对同步操作的分析，因此我们相信二分组协议已经将访存依赖的数量降低到不再成为并发程序动态分析瓶颈的数量级。

对二分组协议的性能评估结果如图 4.9 所示。我们使用 RWTrace 的运行时开销对二分组协议的运行时间进行归一化 (RWTrace 的运行时间记为 100%)。相比于 RWTrace，二分组协议引入了 0-54.7% (中位数 21%) 的额外时间开销。因此，二分组协议实现了访存依赖在线约减和运行时间开销之间的权衡，如图 4.10 所示。在大部分实验程序中，二分组协议都只带来了可以忽略或较少的运行时间开销。考虑到其多达 97% 的访存依赖约减，这一技术能对并发程序的动态分析产生有益的促进，即二分组协议能在付出合理时间开销的前提下实现即时、准确、简化的访存依赖获取。

4.4 小结

本章提出了两项基于并发程序共享内存访问局部性理论实现的高效访存依赖获取新技术：基于线程局部性的乐观锁技术，可实现即时、准确、高效的访存依赖追踪；基于空间-线程局部性和运行时动态变量分组的二分组协议，可在只轻微降低高效性的前提下实现访存依赖的大幅约减。实验表明在同样获得准确访存依赖的前提下，乐观锁相比互斥锁能实现高达 200 倍的性能提升。二分组协议在 RWTrace 中引入 0-54.7% (中位数 21%) 运行时开销的前提下，能实现 0.5-97% 的访存依赖数量减少。

第五章　基于访存依赖的动态分析

本文提出的理论框架和技术框架成功地指导了两项高效访存依赖获取技术的发现：基于乐观锁的访存依赖追踪技术和基于二分组协议的访存依赖约减技术。回顾技术框架中的三要素 (评价指标、技术、应用)，除理解访存依赖获取技术的评价指标权衡、研究高效的依赖获取技术之外，访存依赖支持的应用，也就是实现并发程序质量保障的动态分析技术，亦是非常重要的研究问题。只有将访存依赖获取技术真正应用于工程实践中，才能体现其实用价值。

鉴于此，本文开展了基于访存依赖的动态分析技术研究，主要用于实现软件质量保障相关的测试/调试技术：

1. 基于我们对“访存依赖”这一概念的深入认识及其获取技术研究中获得的经验，提出访存依赖在并发程序动态分析中的新应用。基于缓存的执行重放技术 CARE (第 5.1.1 节) 通过类似于乐观锁的技术，使用数值预测缓存对写后写 ($\prec_{w\to w}$) 和部分写后读 ($\prec_{w\to r}$) 依赖进行记录，以实现高效的并发程序调试。此外，我们还探讨二分组协议在两类并发程序动态分析中的典型技术——数据竞争检测和假共享检测上的应用 (第 5.1.2 节)，主要关注如何利用简化的依赖记录实现更高效的动态分析。
2. 基于访存依赖对应事件在执行轨迹中发生顺序的性质，将访存依赖 $e_i \prec_d e_j$ 反转并再次执行程序 (即按照 $e_j \prec e_i$ 的顺序执行)，就能得到一条新的执行轨迹。这一反转技术能实现调度空间的探索 (换言之，实现测试用例的生成)，并暴露程序中潜在的缺陷。基于这一技术，我们提出了应用程序的崩溃一致性全自动检测技术 (第 5.2.1 节) 和移动应用的并发缺陷暴露技术 (第 5.2.2 节)，该两项技术在知名开源软件中发现了前所未知的缺陷。

5.1　并发程序的动态分析

5.1.1　基于缓存和依赖推导的执行重放

5.1.1.1　问题定义与解决思路

并发程序的执行重放. 并发程序执行的不确定性导致了并发缺陷难触发、难复现、难检测。针对其中并发缺陷难复现的问题，研究者提出了执行重放 (record and replay，又称为确定性重放 deterministic replay) 技术，在并发程序运行时

直接记录或在运行后间接推导出满足一致性的访存依赖 $\prec_d$，在重放时严格按照 $\prec_d$ 再次执行同一并发程序，以实现对过往并发程序执行的重现。

执行重放技术能确定性地重现被记录执行轨迹中的程序状态、程序输出等信息，从而实现并发程序的“时间穿梭调试”，帮助开发者诊断执行轨迹中导致缺陷的原因 (root cause)，极大地降低开发者诊断并发缺陷的难度 [52]。我们已在第 3 章中技术框架的基础上对基于访存依赖的执行重放技术进行了简要综述，根据其获取访存依赖的方式不同，主要分为基于追踪 (在线记录一致的访存依赖，保证能成功重放并发程序的执行) 和基于合成 (在线记录间接信息或少量信息，使用推导、搜索等技术得到访存依赖) 两类。

为了确保并发程序的执行重放，对全部或部分访存依赖进行追踪，进而获得满足一致性的访存依赖是目前唯一可靠的途径。基于部分执行信息进行访存依赖合成的技术虽然在高效性上具有优势，但存在推导失败的可能 [109]，不能保证重现并发程序的执行轨迹。而为获取满足一致性的访存依赖，现有的技术需要在并发程序中引入大量同步操作 [69]，造成内存密集型应用难以容忍的开销 (在我们的实验中，可能导致程序运行速度下降为原先的数百分之一)。

如何在确定重放并发程序执行的前提下，降低时间和记录的开销是一项具有挑战性的研究问题。执行重放的目标是重现并发程序执行的行为 (如并发性导致的缺陷)。为达到这一目标的访存依赖获取技术对即时性并没有要求 (即不必在运行时即获得访存依赖)，因而可通过仅记录部分依赖关系或间接信息来实现记录的近似准确、高效和简化。

本节关注如何高效、简化地实现并发程序的执行重放技术。

解决思路. 为了实现并发程序的执行重放，我们首先对其进行简化，考察单个线程的执行重放问题：给定线程 t，在程序运行时记录信息，在程序结束后重现执行轨迹中线程 t 的所有程序状态。

在此情形下，设执行轨迹中访存事件集合为 E，我们只需确定

$$E_t = \{e \in E \,|\, (e.t = t) \wedge (e.rw = r)\},$$

即线程 t 中所有读事件的数值 ($e \in E_t$ 中的 $e.v$)，就能在不记录访存依赖的前提下实现对线程 t 的执行重放。图 5.1 展示了使用读事件数值进行线程执行轨迹重放的例子。为了重现线程 1 在某次执行中的行为，我们只需重新执行线程 1 的代码，当执行到具有线程间数据依赖的读操作时，将其正确的数值赋值给相应变量，因执行其他指令所需的操作数均能从线程本地得到，这一技术能保证重现线程 1 的所有程序状态 (包括输出、崩溃等所有程序行为)。

然而，记录执行轨迹中读事件数值实现的执行重放虽在理论上成立，但应用于实际程序时具有以下缺陷：

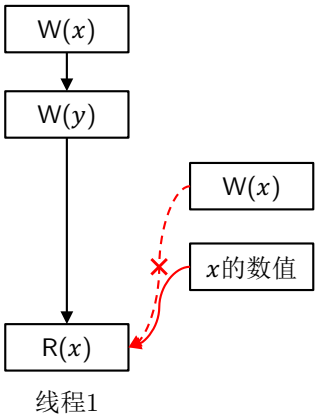

图 5.1 使用变量数值进行线程行为的重放。

1. 真实程序的多次执行由于运行环境、配置等不同，有些读事件的数值在重放时无法用过往执行中的数值替代 (如指针或引用的地址)。在执行重放时直接使用记录的读值将导致不正确的重放或程序崩溃，而保证运行环境的确定性在实现中面临较大挑战；
2. 即便能正确重现每一线程的执行轨迹，推导出满足一致性的 $\prec_d$ 仍然十分困难：已经证明，仅有线程读/写事件数值的情况下，获得满足顺序一致性的事件排序是 NP-完全问题 [57]。这意味着开发者虽然可以重现线程本地计算的行为，但却难以对线程间的数据依赖进行解释，不利于调试由线程交互引发的并发缺陷。

因此，我们考虑将访存依赖追踪技术和读事件数值记录技术二者融合。访存依赖追踪技术开销较大、记录较多，但能提供满足一致性的访存依赖，进而提供重放保障；线程读数值记录推导满足一致性的访存依赖较为困难，但所有记录操作均可在线程本地完成，无需引入额外的同步/原子操作，因此运行时开销较少。我们使用数值预测缓存实现只在少部分共享内存访问时记录依赖以综合两类技术，以此实现高效的并发程序执行重放。

5.1.1.2 基于缓存的执行重放

使用缓存预测读值. 确定 $E_t = \{e \in E \,|\, (e.t = t) \wedge (e.rw = r)\}$ 中事件 e 的读值 $e.v$ 能够实现线程 t 执行轨迹的重放。因此，如果线程 t 的读事件集合在执行中从未读出其他线程写入的数值 (即不存在写后读 $\prec_{w\to r}$ 依赖)，便无需记录任何访存依赖或读值，直接执行该线程 t 的代码即可重现 t 的执行轨迹。

从另一角度考虑，只要我们能“预测”出线程每一读操作的值，就能在不产生任何记录的前提下重现线程的执行轨迹。因此，我们希望寻找一个能快速 (不

引入同步操作)、有效 (在大部分情形下都预测正确) 地预测读操作数值的算法。若能实现 100% 的预测准确率，则可以实现零记录开销的线程轨迹执行重放。

鉴于并发程序共享内存访问的线程局部性 (参考第 2.2 节中的讨论)，绝大部分读/写操作都具有相同的线程间数据依赖，因此我们很自然地考虑使用数值预测缓存 [89] 来预测线程读出的数值。

数值预测缓存的典型实现是在线程本地维护这一线程已知每一变量 x 最新的数值 $V_t(x)$：

1. 对于读事件 $R(t,x,v)$，若 $V_t(x)=v$，则说明数值预测缓存命中 (预测成功)。
2. 对于读事件 $R(t,x,v)$，若 $V_t(x)\neq v$，数值预测缓存缺失 (miss)，可能是因之前 x 被替换出缓存或数值 v 由另一个线程写入，此时应用 $V_t(x)\leftarrow v$ 更新数值预测缓存。
3. 对于写事件 $W(t,x,v)$，更新数值预测缓存使未来的预测满足 $V_t(x)=v$。若缓存容量不足则使用替换算法 (如 LRU 算法) 进行缓存替换。

对于缓存命中 (数值预测成功) 的事件，无需做出任何记录。在对该线程进行重放时，我们维护与记录时完全等价的缓存，即对于数值预测缓命中的读事件，直接从缓存中取出相应数值。这同时也避免了由执行不确定性 (如数据地址发生改变) 而导致重放失败的可能性 (在没有绝对指针运算相关数据依赖的假设下其正确性成立。又因本技术关注 Java 语言程序的执行重放而 Java 语言不包含指针运算，因此本假设成立)。

数值预测缓存能帮助我们实现高效的访存依赖获取，是因为：(1) 并发程序共享内存访问具有空间-线程局部性，使用固定大小 (如 64KB) 的数值预测缓存 (使用 LRU 等启发式算法实现旧缓存数值的替换) 就能实现较高的命中率；(2) 缓存在每一线程单独维护，无需引入额外的同步/原子操作。

记录部分访存依赖. 上述讨论表明，在数值预测缓存命中时无需作出任何记录。但数值预测缓存并不能保证对每一读事件均预测正确，故必须在缓存缺失的情况下进行额外的记录，以保证能复现并发程序的行为。

我们在数值预测缓存缺失时获取满足一致性的访存依赖，以实现执行重放技术。其工作原理如图 5.2 所示，大方框代表一条读指令在系统修改后所表现出的实际行为。首先，对于所有的写操作，我们都使用互斥锁对其进行保护，并由此获得所有写操作上的访存依赖 (这一处理类似于我们在第 4.1 节中描述的乐观锁技术)。对于读操作，我们首先执行共享内存读取，并获得相应的访存事件 $e_r=R(t,x,v)$。读操作执行完毕后，使用数值预测缓存对读值进行预测，然后根据缓存的预测结果分情况讨论：

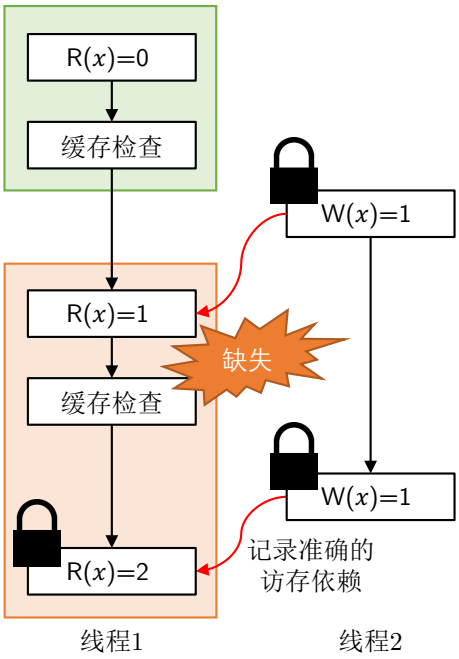

图 5.2 CARE 在数值预测缓存缺失时再次执行读操作并用锁保护以实现访存依赖记录。

1. 缓存命中，即 $V_t(x) = v$。此时，e_r 读出的数值与缓存数值相同，因而可以假设 e_r 和缓存检测是同时、原子发生的，所以无需执行额外操作。图 5.2 中第一次对 x 的读取为这一情况。
2. 缓存缺失，即 $V_t(x) \neq v$，此时需要记录访存依赖。我们与使用乐观锁相同的技巧，将 e_r 所对应读事件的值丢弃，并且在互斥锁的保护下再次进行读操作，以保证能够获得满足一致性的写后读 ($\prec_{w\to r}$) 依赖。图 5.2 中第二次对 x 的读取为这一情况。

由此，我们的执行重放算法在程序运行时记录部分满足一致性的访存依赖 $\prec_d \subseteq \prec_{tot}$，其能够作出如下保证：

1. 对于数值预测缓存命中的读事件，总能在重放时推导它的数值；
2. 对于数值预测缓存缺失的读事件 e_r，对于其实际写入数值的写事件 e_w，我们保证 $e_w \prec_{d^*} e_r$；
3. 对于两个写入同一变量的事件 e_i, e_j ($e_i \asymp e_j$ 且 $e_i.rw = e_j.rw = w$)，其顺序记录在 $\prec_d$ 中，即 $e_i \prec_{d^*} e_j$ 或 $e_j \prec_{d^*} e_i$ 成立。

基于缓存的执行重放技术对程序进行的修改如图 5.3 所示，其读重试 (第 5-8 行) 和写上锁 (第 14-18 行) 的技术类似于基于乐观锁的访存依赖追踪 (在第 4.1 节中讨论)。由于只需写后写 ($\prec_{w\to w}$) 和写后读 ($\prec_{w\to r}$) 两类依赖，因此

```
1  对读指令 x_k^t = R(x) 作出的修改
2  begin
3      e_r : x_k^t = R(x);
4      if V_t(x) ≠ e_r.v then
5          lock(ℓ_x); // 数值预测缓存缺失
6          begin
7              e'_r : x_k^t = R(x); // 在临界区中再次读取 x
8              e_w ← w_x;
9          unlock(ℓ_x);
10         V_t(x) ← e'_r.v;
11         记录访存依赖 e'_w ≺_d e_w;
12 对写指令 W(x, x_k^t) 作出的修改
13 begin
14     lock(ℓ_x);
15     begin
16         e'_w ← w_x; w_x ← e_w;
17         e_w : W(x, x_k^t);
18     unlock(ℓ_x);
19     记录访存依赖 e'_w ≺_d e_w;
20     V_t(x) ← e_w.x;
```

图 5.3 基于缓存的执行重放对读/写指令作出的修改，下划线为被修改的指令。

只要在写事件发生时维护最近一次写入变量 x 的事件 w_x，就可实现访存依赖获取 (第 8, 16 行)。

可见记录的访存依赖 $\prec_d$ 并不满足一致性：它只包含了全部的写后写 ($\prec_{w\to w}$) 和部分的写后读 ($\prec_{w\to r}$) 两类依赖关系，因此并非按照任意 $\prec_{d^*}$ 执行程序都能实现并发程序的执行重放。数值预测缓存体现了我们在执行重放这一问题上对于技术指标作出的权衡：为了实现高效 (运行时开销小)、简化 (记录数量少) 的访存依赖获取，我们适当放松了即时性 (仅追踪部分访存依赖) 和准确性 (不再满足一致性，但仍然可以保证复现并发程序各个线程的执行轨迹)。鉴于这一权衡增加了执行重放的难度，我们设计了专门算法来帮助开发者实现满足顺序一致性的全局调度合成。

基于依赖推导的执行重放. 在缓存缺失时，我们使用同步操作获取准确的访存依赖，即对于缓存缺失的读事件 e_r，我们记录其对应的写者 $e_w \prec_d e_r$。又因所有对 $e_r.x$ 的写操作都被串行化，由此我们可以确定 e_r 在事件序列中的位置：我

们在 E 中找到最早对 $e_r.x$ 的写事件 e'_w ($e'_w.x = e_r.x$ 且 $e'_w.rw = w$) 满足

$$\forall e \in E.\ \big((e.x = e_r.x) \wedge (e.rw = w)\big) \rightarrow e'_w \prec_{d^*} e,$$

即对于每一个缓存缺失的事件，我们都能在所有对 x 的写事件中确定其发生的位置，从而恢复出 e_r 相关的读后写依赖 ($\prec_{r\rightarrow w}$)——若 e'_w 存在，则 $e_r \prec_{r\rightarrow w} e'_w$ 成立。换言之，对于缓存缺失的读事件和所有写事件，我们能通过离线合成的方式得到它们完整的写后写 ($\prec_{w\rightarrow w}$)、写后读 ($\prec_{w\rightarrow r}$) 和读后写 ($\prec_{r\rightarrow w}$) 依赖。

在执行重放时，我们只需按照推导出的完整依赖集合中的事件顺序执行程序，同时维护数值预测缓存，对于执行轨迹中数值预测缓存命中的事件，我们从缓存而非共享内存中读出数值，即可实现并发程序每一线程执行轨迹的重现。这一性质可由数学归纳法证明，已在我们的论文 [78] 中描述，此处不再赘述。

虽然使用上述算法按记录的 $\prec_d$ 执行程序能重现每一线程单独的执行轨迹，但由于从缓存中读出数值破坏了 Java 内存模型的语义，我们并没有获得全局事件排序。因而我们对满足全局顺序一致性的事件序列进行合成。鉴于我们的算法已经获取了相当大部分的访存依赖 (数值预测缓存缺失时的写后读依赖和全部写后写依赖)，对于缓存命中的读事件 $e_r \in E$，令

$$E_w = \{e_w \in E \mid (e_w.rw = w) \wedge \neg(e_w \prec_{d^*} e_r) \wedge \neg(e_r \prec_{d^*} e_w)\}$$

为与 e_r 无法确定顺序写者的集合，只要下列二者之一成立，我们即可确定 e_r 和 E_w 内所有事件的顺序关系：

1. $\forall e_w \in E_w,\ e_w.v \neq e_r.v$，即 E_w 中的所有事件都写入了与 e_r 不同的数值；
2. $\forall e_{w_1}, e_{w_2} \in E_w,\ e_{w_1}.v = e_{w_2}.v = e_r.v$，即 E_w 中的所有事件都写入与 e_r 相同的数值。

在情形 1 中，对于任意的 $e_w \in E_w$，$e_r \prec_{d^*} e_w$ 都必须成立，否则将导致 e_r 读出不同于缓存的数值。在情形 2 中，我们选取任意的 $e_w \in E_w$，令 $e_w \prec_{w\rightarrow r} e_r$ 为写后读依赖，即可满足顺序一致性。我们在此基础上进一步分析，提出启发式算法以在运行时获得满足顺序一致性的执行轨迹：

1. 在上述两个情形均不满足时，一定存在写事件 e_w 满足 $e_w \prec_{d^*} e_r$ 写入了 e_r 的数值 ($e_r.v = e_w.v$)。因此在执行重放时我们仍然按照记录的依赖顺序进行执行。当有多个线程同时满足 $\prec_{d^*}$ 而可供执行时，我们优先执行那些数值预测缓存中数值与共享内存中数值相同的事件 (此时这一事件可以直接从共享内存中读取从而满足顺序一致性)；
2. 另一方面，$|E_w|$ 越小，生成满足顺序一致性的执行概率越大。若能在 $\prec_d$ 中加入额外的顺序关系，则能使得 $|E_w|$ 减小。因此，我们不以二元组的形式

记录 $\prec_d$，而是为每一事件维护其发生时的时间戳。我们可以将不同变量的时间戳进行合并，用原子计数器进行维护。对于事件 e_i, e_j，定义 $e_i \prec_d e_j$ 当且仅当 e_i 发生时的时间戳小于 e_j 发生时的时间戳，由此记录更多的顺序关系，减小 $|E_w|$。

上述启发式算法帮助我们在执行重放时推导出所有实验对象满足顺序一致性的访存依赖。

5.1.1.3 系统实现与实验评估

系统实现. 我们实现了 Java 程序的执行重放技术 CARE (Cache Guided Deterministic Replay)。CARE 的实现基于 DPAC [81] 框架，其使用 JVM Tool Interface [2] 和 ASM 字节码转换工具 [1]，在 Java 类被加载时获得加载类的字节码并进行动态插装，因此可以捕获动态创建类、反射类中的共享内存访问。由于插装在字节码层完成，因而 Java 虚拟机的优化、垃圾回收等机制不会对其正确性产生影响。

CARE 扫描加载类的字节码，并在其中的共享内存访问指令 (数组访问 `*aload`/`*astore`、静态域访问 `getstatic`/`putstatic` 和对象域访问 `putstatic`) 前后插入相应的字节码。对于写指令，在其前后插入同步指令并记录访存依赖；对于读指令，当读出数值与缓存数值相等时直接返回，否则使用同步指令包裹再次执行的同一读指令并记录访存依赖。

在程序运行时，我们使用时间戳记录访存依赖。我们维护线程 t 本地的时间戳 tsc_t (使用 `ThreadLocal` 实现) 和全局的时间戳 tsc。每当线程执行读/写指令时，线程本地时间戳相应更新 ($tsc_t \leftarrow tsc_t + 1$)；当访存依赖被捕获时，其对应事件的全局时间戳相应更新 (使用原子操作实现 $tsc \leftarrow tsc + 1$)。在 CARE 的事件记录中，事件 e 用三元组 $\langle t, tsc_t, tsc\rangle$ 表示，任意两个事件满足 $\langle t, tsc_t, tsc\rangle \prec_d \langle t', tsc'_t, tsc'\rangle$ 当且仅当 $tsc < tsc'$。

在对执行轨迹进行记录时，我们使用与记录时完全相同的方式插装程序，并在访存指令执行前设置断点。指令能执行当且仅当指令所对应的事件 e 之前的事件 ($\{e' \mid e' \prec_{d^*} e\}$) 均已被执行。在运行时，我们维护当前能执行下一个访存事件的线程集合，并且每当事件 e 被执行后，我们都对这一集合进行更新，以实现与 $\prec_{d^*} = tr(\prec_d \cup \prec_{po})$ 一致的执行重放。

实验设计. 我们使用表 5.1 中描述的基准程序对 CARE 进行实验评估。表中的基准程序选自 Dacapo Java 基准程序集和我们对比技术 LEAP 的论文 [69]，包含服务器类应用、科学计算类应用和离散搜索类应用。

表 5.1 用于评估基于缓存 Java 程序执行重放技术 CARE 的实验对象。

实验程序	任务描述
avrora	AVR 微控制器模拟
batik	SVG 图像生成
h2	数据库事务
lusearch	Lucene 搜索
sunflow	数值模拟计算
tomcat	HTTP 服务器
xalan	XML 转换
tsp	旅行商问题求解
moldyn	分子动力学模拟

评判执行重放技术优劣的主要指标是在运行时引起的系统执行速度拖慢和产生记录的数量：执行重放技术需要在获得满足一致性访存依赖的前提下，尽可能地确保高效性 (减少系统速度拖慢) 和简化性 (减少记录数量)。我们选取同样能获取一致访存依赖的技术 LEAP 作为对比。我们在同样的实验平台上实现了 LEAP，用 `monitorenter` 和 `monitorexit` 指令对共享内存访问进行包裹，并记录共享变量在线程间访问的顺序 (LEAP 记录完整的写后写 $\prec_{w\to w}$、写后读 $\prec_{w\to r}$、读后写 $\prec_{r\to w}$ 和读后读 $\prec_{r\to r}$ 依赖，类似于 RWTrace 对比实验中的 MutexLock)。我们使用同样的输入运行 CARE 和 LEAP，并对比它们的运行时间开销和记录数量。鉴于两种技术的运行时间不同，实验结果中每秒的记录数量按照未修改程序的运行时间进行归一化。

所有实验在一台运行 Ubuntu Linux (Kernel 3.2)、拥有 2 个 6 核心 Intel Xeon E5645 (共 12 核心) 处理器和 48GB 内存的服务器上完成，实验数据来自 10 次运行的平均值。

实验结果. 实验结果如表 5.2 所示。表中第 2-4 列包含了 CARE 的实验结果：运行时开销 (运行时间的倍数)、日志数量、无法直接推导出访存依赖的事件数量 ($|E_w| > 0$ 的读事件 e_r 的数量) 和启发式算法能否成功推导出满足顺序一致性的访存依赖。实验结果表明，由于同步局部性的存在和版本号合并技术，对于真实程序绝大部分读事件都满足 $|E_w| = 0$，仅有少量的事件无法准确确定其顺序。对于这些事件，CARE 的启发式算法成功地推导出了满足顺序一致性的访存依赖，从而能够实现正确的执行重放。

相比于 LEAP 的实验结果 (表中第 5-6 列)，CARE 总体而言显示出了中位数 2.6 倍的运行时开销减少和中位数 4.9 倍的记录数量减少。对于内存密集型

表 5.2 CARE 和 LEAP 的运行时开销和日志数量对比。

实验程序	CARE				LEAP	
	时间 (×)	日志 (/s)	未定序	解决	时间 (×)	日志 (/s)
avrora	1.52	2.18MB	23K	✓	9.48	24.3MB
batik	1.49	1.51KB	0	✓	3.77	2.32KB
h2	18.5	24.2MB	0	✓	62.8	27.4MB
lusearch	3.41	6.53MB	0	✓	9.01	46.0MB
sunflow	64.9	886MB	0	✓	389	6.03GB
tomcat	4.76	7.80MB	15	✓	11.9	23.5MB
xalan	7.18	13.6MB	0	✓	12.2	143MB
tsp	2.79	1.84MB	0	✓	111	570MB
moldyn	11.9	24.1MB	0	✓	50.5	303MB

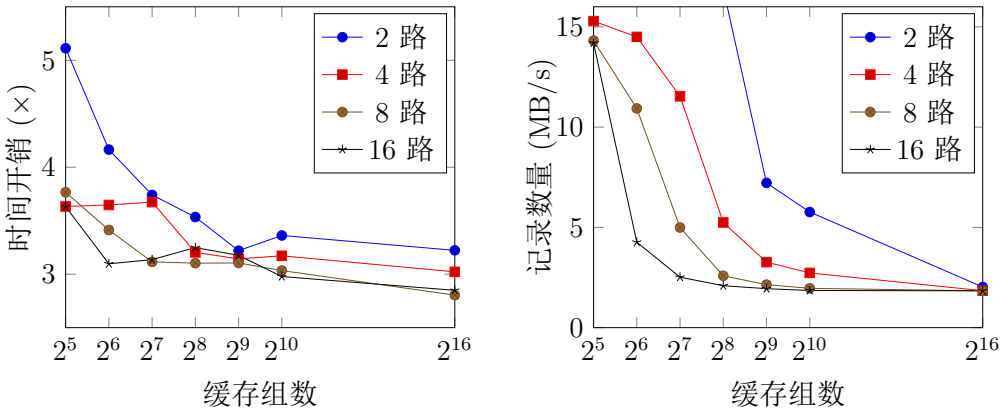

图 5.4 CARE 在不同缓存设置下的时间开销和记录数量对比。

的基准程序 (如 tsp)，CARE 的时间开销和记录数量分别仅为 LEAP 的 1/40 和 1/300。因此，我们认为 CARE 所作出的权衡 (无法像 LEAP 一样在运行时即得到一致的访存依赖，部分情况下需要离线推导，但在数值预测缓存命中时能大幅增加依赖记录的高效性和简化性) 能更高效地对现实程序进行执行重放。

最后，我们在 tsp 基准程序上评估不同缓存设置对运行时开销和记录数量的影响 (tsp 是其中访存最密集的应用程序，因此其对缓存的性能亦最为敏感)，评估结果如图 5.4 所示。可见，在缓存较小 (小于 512 组) 时，随着缓存的增加，CARE 的运行时间和日志数量大幅降低，但当缓存达到一定数量时 (对 tsp 而言是 1024 组)，其性能增加不再明显。因此我们有理由认为一个中等大小的缓存即能支持高效的执行重放。

小结. 本节描述了利用并发程序共享内存访问的线程局部性实现的基于缓存的执行重放技术。这一技术体现了我们在访存依赖获取技术指标之间进行的权衡：为执行重放这一并发程序动态分析问题定制高效、简化 (同时即时性、准确性相对弱化) 的访存依赖获取技术，相比于现有技术能在实现同样目标 (确定性重放并发程序执行) 的前提下大幅减少时间开销和日志数量。

5.1.2 基于二分组协议的动态分析

5.1.2.1 数据竞争检测

问题定义与解决思路. 若存在程序 P 的执行轨迹 τ，其中包含先后发生的冲突访存事件 $e_i \asymp e_j$[1]，则称 P 中的变量 $e_i.x$ 存在数据竞争 (data race)。除直接导致不符合规约的变量数值外 [56]，数据竞争也是引起其他类型并发缺陷 (如原子性违反或顺序违反) 的主要原因 [94]。因此对并发程序中数据竞争的检测得到了研究者的广泛关注。例如可以直接对程序进行静态数据竞争检查 [125] (静态数据竞争检测)，或对执行轨迹中的数据竞争进行检测 (动态数据竞争检测)。

动态数据竞争检测技术开销小、正确性 (soundness) 容易保证，在实践中已得到广泛应用，且检出了知名并发程序中前所未知的数据竞争 [71]。动态数据竞争检测通过考察执行轨迹中的访存事件推断出并发程序中潜在的数据竞争。例如，若存在两个访存事件 $e_i \asymp e_j$ 且 e_i, e_j 所对应的锁集合交集为空，则它们可能形成数据竞争 [115]；若能求出执行轨迹中同步事件发生的因果关系 $\prec_{syn}$，则若存在 $e_i \asymp e_j$ 满足

$$\neg(e_i \prec_{syn} e_j) \wedge \neg(e_j \prec_{syn} e_i),$$

则可以证明并发程序中存在数据竞争或死锁。目前，基于 Lamport happens-before [82] 因果模型的数据竞争检测是研究的热点之一 [25, 55, 127]。

现有的数据竞争检测技术需要获取准确的访存依赖 (必须使用变量指派)，并为每一记录的访存依赖 $e_i \prec_d e_j$ 检测其是否满足 $(e_i \prec_{syn} e_j) \vee (e_j \prec_{syn} e_i)$，因此可能带来数百甚至数千倍的运行时开销且需要大量内存来维护元数据。如何降低这些开销是一项重要的研究挑战。

由于二分组协议能在不付出较多运行时开销的前提下大幅降低访存依赖的数量，因而我们很自然地考虑如何使用二分组协议以实现高效的数据竞争检测。但使用二分组协议获取的访存依赖 $e_i \prec_d e_j$ 不能直接用于数据竞争检测：由于对变量进行了分组，即便 $(e_i \prec_{syn} e_j) \vee (e_j \prec_{syn} e_i)$ 不成立，$e_i \prec_d e_j$ 仍可能是不存在数据竞争的假依赖。

[1] $e_i \asymp e_j$ 当且仅当 $e_i.x = e_j.x, e_i.t \neq e_j.t$ 且 $(e_i.rw = w) \vee (e_j.rw = w)$。

考虑以上情形的反面：若对于所有的 $e_i \prec_d e_j$ 都满足 $(e_i \prec_{syn} e_j) \vee (e_j \prec_{syn} e_i)$，则可证明本次执行不存在 $\prec_{syn}$ 意义下的数据竞争。对此，我们提出基于二分组协议的自适应数据竞争检测算法。

基于二分组协议的数据竞争检测. 二分组协议能在小幅增加运行时开销的前提下大幅减少访存依赖的数量。由于因果关系检查通常是较为耗时的操作，基于二分组协议实现高效数据竞争检测技术的直观想法是尽可能地只对 $\prec_d$ 中的事件二元组 $\langle e_i, e_j \rangle \in \prec_d$ 进行因果关系检查。二分组协议能大幅减少 $|\prec_d|$ 的数量，因而可减少因果关系检测的次数，进而降低数据竞争检测的开销。

实现基于二分组协议的高效数据竞争检测立足于如下两点观察：

1. 对于某一变量组 G_k，若对所有其上记录的访存依赖 $e_i \prec_d e_j$ 都满足 $(e_i \prec_{syn} e_j) \vee (e_j \prec_{syn} e_i)$，则对于任意变量 $x \in G_k$，都不存在 $\prec_{syn}$ 意义上的数据竞争；
2. 变量组 G_k 具有空间-线程局部性，即在较长的时间内，要么被一个线程独占访问，要么在多个线程间以只读的方式共享。对于独占访问的情形，其不存在多线程访问故而不存在数据竞争；对于多线程共享的情形，现实程序通常都以互斥锁或条件变量等同步机制实现，通常亦不存在数据竞争。

同时，因并发程序具有同步局部性，我们有理由相信，对于绝大部分的变量组 G_k，其组上的访存依赖不存在 $\prec_{syn}$ 意义下的数据竞争。只要这一性质在程序运行过程中保持不变，我们就只需在变量组的粒度上进行数据竞争检测。

因此，我们使用二分组协议对访存依赖进行获取，并在程序运行时维护因果关系 $\prec_{syn}$ (例如，对于 Lamport happens-before 因果关系，可以使用 FastTrack [55] 算法进行维护)。在获得变量组 G_k 上的访存依赖 $e_i \prec_d e_j$ 后，若 $(e_i \prec_{syn} e_j) \vee (e_j \prec_{syn} e_i)$ 成立，则说明 G_k 所对应的同一变量组到目前为止都没有产生数据竞争。我们称这样的检测为“粗粒度检测”。若粗粒度检测一直保持到程序执行结束，则对于任意的 $e_i \asymp e_j$，我们有 $e_i \prec_{d^*} e_j$。由于所有记录的 $e_i \prec_d e_j$ 都满足 $(e_i \prec_{syn} e_j) \vee (e_j \prec_{syn} e_i)$，而 $\prec_{d^*}$ 为 $\prec_d$ 增加了 $\prec_{po}$ 的传递闭包，因此有 $(e_i \prec_{syn} e_j) \vee (e_j \prec_{syn} e_i)$ 成立，则 E 中的事件不存在 $\prec_{syn}$ 意义下的数据竞争。

但当变量组中的依赖 $e_i \prec_d e_j$ 是 $\prec_{syn}$ 意义上的数据竞争 (称为变量组数据竞争)，即 $\neg(e_i \prec_{syn} e_j) \wedge \neg(e_j \prec_{syn} e_i)$ 时，我们则需要判定其变量组 G_k 中的变量是否存在实际的数据竞争。对于假依赖的情况 (图 4.7)，虽然记录了访存依赖 $e_i \prec_d e_j$，但实际并没有任何线程间数据流，也没有数据竞争，若用变量组的数据竞争代替实际数据竞争，会导致假阳性 (误报) 的发生 [124]。

对于产生了变量组数据竞争的变量组 G_k 和访存依赖 $e_i \prec_d e_j$，需要解决以下两个问题：

1. 由于变量组存在数据竞争，在没有假依赖的前提下，变量组中的变量是可能产生数据竞争的候选，因此需要对 $x \in G_k$ 中的变量进行数据竞争检测。
2. 需要判定是否已经存在变量 $x \in G_k$ 的数据竞争：存在访存事件 e 满足 $e \asymp e_j, e.x = e_j.x$，它发生在 e_i 的同一线程且发生在 e_i 之前 $(e \prec_{po} e_i)$，且 $\neg(e \prec_{syn} e_j) \wedge \neg(e_j \prec_{syn} e)$。

为了解决问题 (1)，对于检测到变量组数据竞争的 G_k，我们对 G_k 中的变量使用细粒度数据竞争检测技术 [55]，以确保 G_k 中变量未来发生的数据竞争能被正确检出。我们为 G_k 中的每一变量追踪准确的访存依赖，并在访存依赖获取后检查是否存在数据竞争。相当于我们对变量组 G_k 中的变量，使用已有的技术进行数据竞争检测。当变量组经过一段时间的细粒度数据竞争检测后，其中所有共享变量访问均在 $\prec_{syn}$ 中被排序，此时 G_k 中的变量再次具有空间-线程局部性，从而应当切换到按变量组 (即粗粒度) 的数据竞争检测检测，以提高性能、减少维护开销。

在细粒度检测时，我们使用队列维护未被 $\prec_{syn}$ 确定顺序的变量 (即可能产生数据竞争的变量)。一旦我们检测到某一变量在某次访存依赖发生后被 $\prec_{syn}$ 覆盖，就将它从队列中删除。当队列为空时，这一组依赖中不存在可能产生数据竞争的变量，即可恢复到粗粒度检测。当二分组发生时，生成的子区间继承父区间的检测粒度。

为了解决问题 (2)，若 $e_i \prec_d e_j$ 包含了真实的数据竞争 $(e \asymp e_j)$，由于原始的二分组协议在 e_j 发生时并没有包含 G_k 变量组对 $e_j.x$ 的访问历史记录，因而我们必须在运行时记录额外信息以保证数据竞争不会被漏检。我们在粗粒度检测时保存每一共享变量的访问日志 (仅需为每一变量维护每一线程最近的读/写的事件，可使用类似 RWTrace 中对 r_x 的维护算法，使用并发散列表实现，且具有 $O(1)$ wait-free 的性质)。当组依赖发生时，组内变量的记录被扫描，并转换为细粒度检测所需的元数据。

上述算法在粗/细粒度检测之间切换。对于执行轨迹中 $\prec_{syn}$ 意义下的数据竞争 $e_i \asymp e_j$，以下情形之一必然成立：

1. $e_i.x = e_j.x$ 对应的变量组 G_k 处于粗粒度检测模式。由于检测到变量组数据竞争后 G_k 会切换到细粒度检测，因而 G_k 处于粗粒度检测模式意味着之前 G_k 变量组上获取的访存依赖在 $\prec_{syn}$ 下不存在数据竞争。据此，冲突的事件 $e_i \asymp e_j$ 一定对应某一获取的访存依赖 $e \prec_d e_j$ 且 $(e = e_i) \vee (e \prec_{po} e_i)$。此时扫描 G_k 相关的变量访问日志即能检测出这一数据竞争；

```
1  struct {
2    int x, y;
3  } global;
4
5  void thread_1() {
6    for (int i = 0; i < 1000000000; i ++)
7      global.x = global.x + 1;
8  }
9
10 void thread_2() {
11   for (int i = 0; i < 1000000000; i ++)
12     global.y = global.y + 1;
13 }
```

图 5.5 存在假共享缺陷的代码片段。

2. $e_i.x = e_j.x$ 对应的变量组 G_k 处于细粒度检测模式。由于每一变量单独维护元数据，细粒度检测等价于已有的数据竞争检测技术，亦能保证数据竞争被检出。

因这一算法在粗/细粒度数据竞争检测之间切换，故称为自适应数据竞争检测算法。其高效性基于两点假设：(1) 变量组依赖在数量级上少于共享内存访问；(2) 绝大部分变量组依赖在 $\prec_{syn}$ 中确定顺序。假设 (1) 已在第 4.3.2 节对二分组协议的实验评估中得到验证；并发程序具有的同步局部性则能说明对于实际程序，假设 (2) 成立。在两点假设均成立的前提下，绝大部分变量组都处于粗粒度检测模式。对于处于粗粒度检测模式变量组中的变量，都只需在 $O(1)$ 时间内进行少量数据结构更新，不涉及 $\prec_{syn}$ 的比较和维护操作，因而可实现高效的数据竞争检测。

5.1.2.2 假共享检测

问题定义与解决思路. 假共享 [123] 是并发程序中常见的一类导致性能问题的缺陷。在通过缓存一致性实现共享内存的多核处理器硬件上，为了利用并发程序空间-线程局部性以减少维护缓存一致性的开销，缓存数值的一致性通常以代表一段连续内存的缓存线 (通常为 64 字节或 128 字节) 为单位。访问缓存线中的任意数据都将导致整个缓存线的一致性维护。在具有空间-线程局部性的程序中，缓存线通常在一段时间内被一个线程独占访问或在多个线程之间共享，因此这一机制能够减少缓存一致性维护的次数，从而减少处理器访存的延迟。

若开发者编写不当 (或编译器对内存地址布局不当) 而使得两个不具有空间-线程局部性的变量被放置在同一缓存线中，则可能导致不必要的缓存一致性维护，从而引起并发程序的性能问题。图 5.5 展示了具有假共享缺陷的代码片

段。其中变量 x 和 y 分别被两个线程独占访问，并不存在线程间的数据依赖。但因其位于同一结构体中 (从而在内存上位于连续的地址)，而结构体又通常被编译器排布在同一缓存线中，两个线程虽然访问不同的内存地址，却由于 x 和 y 处于同一缓存线，将会导致缓存一致性在两个处理器之间频繁维护，大幅降低并发程序的性能。

现有技术能实现假共享 [91, 141] 的自动检测。但因事先无法预知可能出现假共享的内存地址，为了实现假共享检测，就必须为每一缓存线分别维护共享内存的访问记录。假设程序使用 M 的内存，则这一开销也是 $O(M)$ 的。鉴于假共享和二分组协议定义中假依赖定义的相似性，我们使用二分组协议进行假共享检测，能大幅降低元数据维护的开销。

基于二分组协议的假共享检测. 假共享缺陷是由实际不存在数据依赖的缓存一致性维护导致的，因此假共享也可在我们的形式化框架下定义。对于共享内存访问事件 e_i, e_j，若 $(e_i.t \neq e_j.t) \wedge (e_i.x \neq e_j.x)$，则 $e_i \not\asymp e_j$ 不存在数据依赖。在理想缓存的假设下 (缓存线大小为 m 且缓存为容量无穷大的全相联实现)，e_i, e_j 被分配为同一缓存线当且仅当

$$\left\lfloor \frac{e_i.x}{m} \right\rfloor = \left\lfloor \frac{e_j.x}{m} \right\rfloor 。$$

为了检测 e_i, e_j 之间发生的假共享，我们只需按缓存线的粒度 (将地址空间中连续长度为 m 的单元指派为同一锁) 追踪访存依赖，此时不具有实际数据依赖的访存依赖就意味着不必要的缓存一致性维护。这一定义与第 4.2.2 节二分组协议中的“假依赖”定义完全一致：对于记录的访存依赖 $e_i \prec_d e_j$，若在 $e_i.t$ 中上一次对变量组记录依赖到 e_i 发生的时间内，不存在对 $e_j.x$ 的访问 e 满足 $e \asymp e_j$，则 $e_i \prec_d e_j$ 是一个假依赖，同时也是一个潜在的假共享。在缓存线上频繁出现的假依赖/假共享将导致并发程序性能不必要的下降。

因此，二分组协议自然适用于假共享检测。二分组协议进行区间二分的条件是检测到假依赖的累积，而假依赖恰好对应着变量组大小的缓存线上的假共享 (设变量组 $G = [l, r)$，则缓存线包含内存地址 $[l, r)$ 之间的变量)。我们只需设置每一变量组在大小恰好为 m 时不再进行二分组 (但仍统计假依赖)，并当某一大小为 m 的变量组频繁产生假依赖时报告假共享即可。

在第 4.3.2 节的实验评估中我们发现，对不存在假共享的真实程序 (桌面、科学计算和服务器程序)，二分组协议通常仅需极少的二分组次数。据此相对于目前已有的假共享检测技术 [91, 141] 需要 $O(M)$ 的空间，二分组协议可以大幅降低假共享检测的空间开销。此外，使用二分组协议进行假共享检测无需追踪读后写 ($\prec_{r\to w}$) 依赖。对于产生假共享/假依赖的两个事件 e_i, e_j，至少有一个是

写事件。因此对于产生假依赖的共享内存访问，其至少产生一个写后读 ($\prec_{w\to r}$) 或写后写 ($\prec_{w\to w}$) 依赖，故至少有 1/2 的依赖被捕获，而不会造成假依赖的漏检。基于乐观锁的访存依赖追踪技术能在大部分读操作不引入任何同步/原子操作的前提下实现写后读和写后写依赖的追踪，以实现高效的假共享在线检测。

5.2 基于依赖反转的软件测试

访存依赖确定了冲突事件 $e_i \asymp e_j$ 之间的顺序，从而实现了并发程序共享内存不确定性的“捕获”。获得反映真实执行的访存依赖对实现并发程序的动态分析技术 (轨迹分析或并发控制) 是至关重要的。若对于实际发生的 $e_i \prec_{tot} e_j$ 记录 $e_j \prec_d e_i$，则会引发执行重放失败、缺陷预测误报/漏报、并发控制失效等后果。因此并发程序的动态分析技术通常要求访存依赖 $\prec_d$ 满足一定的准确性 (如满足一致性，$e_i \prec_{d^*} e_j$ 当且仅当 $e_i \prec_{tot} e_j$)，即保证访存依赖描述了与 $\prec_{tot}$ 等价的程序执行。

访存依赖的准确性对实现并发程序的动态分析是至关重要的。从另一角度而言，若我们改变获取的访存依赖，就能得到不同的程序执行轨迹。对访存依赖进行系统化、有目标的调整 (如将 $e_i \prec_d e_j$ 反转为 $e_j \prec_d e_j$) 则能实现并发程序调度空间的遍历。

依赖反转技术已广泛应用于并发程序的测试、验证等质量保障技术 [102, 29]。沿着这一研究思路，我们将依赖反转的思想运用于两类新型的软件测试问题：应用程序崩溃一致性的全自动检测和移动应用的并发缺陷暴露。

5.2.1 应用程序崩溃一致性的全自动检测

5.2.1.1 问题定义与解决思路

崩溃一致性的全自动检测问题. 计算机软件必须在罕见异常事件发生时仍然保证重要数据的安全。而这些罕见事件通常在测试中难以被覆盖，会在软件中埋下“定时炸弹”，轻则导致软件崩溃 (例如由移动应用中的异常引发 [138])，重则导致重大的系统故障和损失 (如 2003 年的美国大停电 [39])。

在本节中，我们讨论应用程序的*崩溃一致性* (crash consistency) 检测问题。崩溃一致性指应用程序在持久性存储中保存的应用数据 (如文档、数据、配置等) 于系统发生故障时不会丢失或损坏 [111]。只要应用软件需要保存应用数据，崩溃一致性就是其需要关注的问题，其原因有二：(1) 根据小概率事件大量重复必然发生的事实，随着软件被反复使用，崩溃一致性缺陷必然会被触发; (2) 崩

溃一致性缺陷可能导致文档、数据、配置等重要应用数据丢失或损坏，造成十分严重的后果。

保证应用程序的崩溃一致性需要开发者对软件自身和文件系统的行为都明确掌握。然而，现有的研究表明，开发者对崩溃一致性相关问题的严重性认识不足，即便有经验的开发者在具有强崩溃一致性需求的成熟开源软件 (如数据库系统、版本管理系统等) 中仍存在崩溃一致性缺陷 [111, 142]。

现有用于崩溃一致性检测的半自动技术主要关注测试输入的生成和崩溃现场的模拟 [111, 131, 142]，其具有较为类似的工作流程：(1) 为被测程序生成输入或负载 (workloads)；(2) 使用生成的输入执行程序，并在程序执行时对其执行轨迹 (文件系统访问和 I/O 行为) 予以记录；(3) 根据记录的执行轨迹使用依赖反转技术模拟出可能的崩溃现场 (文件系统快照)，并进行默认的恢复操作 (如文件系统的恢复操作或数据库的自检操作)；(4) 使用人工预先指定的判定程序对崩溃一致性进行检测 (如检测数据库的 ACID 性质)。

尽管这类技术能有效检测具有确定的崩溃一致性语义模型的程序 (如数据库系统、版本管理系统等) 中的缺陷，但步骤 (4) 所需的测试预言对于一般开发者开发的一般应用来说通常是不存在的：开发者不会对文件访问进行显式的建模，从而无法实现对一般应用程序崩溃一致性缺陷的自动检测。

本节解决实现一般应用程序的崩溃一致性检测的主要挑战：构造能根据应用程序崩溃现场 (文件系统快照) 判定其是否存在数据丢失/损坏的测试预言 (test oracle)，以实现一般应用崩溃一致性的全自动检测。

案例分析与研究挑战. 我们首先分析来自真实开源软件中崩溃一致性缺陷的两个案例，并以此揭示崩溃一致性检测的研究挑战。

如图 5.6 所示的代码来自 Ted 文本编辑器，其开发者在保存文件时试图使用临时文件保证崩溃一致性：首先将文件保存到临时文件，然后删除原文件，最后将临时文件重命名。开发者期望在程序执行的任意中间状态，文件系统中总有文件的副本 (原文件或临时文件)，以确保不致数据丢失。

然而，上述代码片段的崩溃一致性依赖于文件系统操作不被乱序执行的假设。实际文件系统的实现并不满足此假设 (POSIX 并未对文件系统的这一行为进行规范)，因此 `TextFileUtils.writeTextFile` 中的 `write` 系统调用和 `deleteItem` 中的 `unlink` 系统调用可能被乱序，导致在程序崩溃时，原文件被删除而临时文件数据又尚未被写入，使得用户数据丢失。这一缺陷能被我们的自动工具检出，其修复已被开发者接受[1]。

[1] https://github.com/xgouchet/Ted/pull/45.

```
1 backupPath = path + ".tmp";
2 TextFileUtils.writeTextFile(backupPath, content);
3 deleteItem(path);
4 renameItem(backupPath, path);
```

图 5.6 文本编辑器 Ted 中具有崩溃一致性缺陷的代码。

因此，如何建模开发者自身意图和实际系统语义之间的鸿沟从而获得崩溃一致性的测试预言，是实现崩溃一致性缺陷自动检测的主要挑战。

另一个例子来自 GNU coreutils sort。`sort` 命令主要在管道和重定向中使用，然而其为了保持兼容性，提供了 in-place 排序的功能：`sort data -o data` 实现对文件内容的排序，这一用法得到了 StackOverflow 上开发者的推荐[1]。

然而，in-place 排序的实现却包含崩溃一致性缺陷：开发者直接使用覆盖 (`O_TRUNC`) 标志打开了文件，将文件内容清空。如果此时系统崩溃或磁盘空间不足，原文件的内容就可能永久丢失。我们的工具检出了这一缺陷。开发者表明在跨平台的软件中正确修复这一问题十分困难，而目前已在文档中标明这一问题可能的影响[2]。

在实际系统中，ext4 文件系统会保证小文件写入的原子性。因此，这一崩溃一致性缺陷仅在文件足够大时才会触发。若不使用特定的测试用例，现有技术中 [111, 131, 142]，仅有 Alice [111] 能报告此类缺陷，但其仍然无法给出一个切实导致崩溃一致性缺陷的用例。

故而，如何减少对特定测试用例的依赖，是实现应用程序崩溃一致性缺陷自动检测的第二个挑战。

解决思路. 给定应用程序及其测试用例，我们捕获其 I/O 事件记录，使用基于依赖反转的搜索算法对崩溃现场进行枚举，实现全自动的崩溃一致性检测。

针对测试预言的挑战，我们设计了用于应用程序崩溃一致性自动检测的测试预言，基于以下两点观察：

1. 开发者在对文件系统进行元数据操作后的文件系统快照，开发者通常认为是正确的 (否则程序中断就会暴露缺陷)；
2. 没有出现数据丢失/损坏的文件系统镜像通常能通过简单的恢复操作得到开发者认为正确的文件系统镜像。

据此，我们提出基于编辑距离的测试预言，首先收集文件系统元数据操作后的快照 (认为是满足崩溃一致性的快照)，然后使用崩溃现场和满足崩溃一致性快照之间的编辑距离 [104] 判定崩溃现场的一致性。

[1] http://stackoverflow.com/questions/9117274。
[2] http://debbugs.gnu.org/cgi/bugreport.cgi?bug=22769.

针对需要特定测试用例才能触发缺陷的挑战，我们提出测试增强 [138] 技术，即在一个简单输入的执行轨迹中人为插入额外的文件系统同步操作来打破文件系统偶然造成的原子性，并在这些同步点进行崩溃一致性的检测，从而使用简单的测试用例检出原先需要特定输入才能触发的崩溃一致性缺陷。

5.2.1.2 崩溃一致性的全自动检测

定义满足一致性的快照. 我们发现，开发者在保存用户数据时，通常都在一定程度上考虑了数据的安全性 (例如保证在进程被杀死或异常退出时不会造成数据丢失)。导致崩溃一致性缺陷的主要原因是开发者对文件系统的理解和文件系统实际实现之间的区别，由于 POSIX 标准未对文件操作的持久性做出明确定义，开发者很可能没有意识到因文件系统缓存、调度等机制的存在，一些按顺序执行的操作在逻辑上被交换了执行顺序。

基于此我们认识到，开发者编写的程序通常在具有强崩溃一致性保障的文件系统实现下是正确的，即在每一个文件操作的系统调用都满足*原子性* (在执行的一瞬间立即完成) 和*持久性* (所做操作立即正确反映到磁盘) 的前提下，应用程序不存在崩溃一致性问题。进而，我们假设开发者在对文件进行元数据操作 (目录操作、文件关闭和 `fsync` 操作) 时考虑到其正确性，在元数据操作后的文件系统状态均满足崩溃一致性[1]。

据此，我们定义期望文件系统快照“Expected Snapshot, ES”为开发者认为正确的文件系统快照，并以此作为判定崩溃一致性的基准。ES 是文件的集合 $\{\langle f_1, c_1\rangle, \ldots \langle f_n, c_n\rangle\}$，其中 f_i 为第 i 个文件的路径 (如 `/mnt/crash/file.txt`)，c_i 为该文件的内容。由于我们主要关注文件内容的持久性，这一定义并不考虑目录的树型结构。文件 f_i 的内容 $c_i = [b_1, b_2, \ldots, b_m]$，其含义为文件大小为 m 字节，且第 j 个字节的数值为 b_j。

运行程序即可收集所有元数据操作后的文件系统镜像 $\mathcal{S} = \{ES_1, ES_2, \ldots\}$。

建模物理磁盘行为. 我们使用虚拟磁盘捕捉文件系统发向磁盘的 I/O 请求，然后使用依赖反转技术生成模拟的崩溃现场。

设 I/O 事件的集合 $E = \{e_1, e_2, \ldots, e_n\}$，由于请求按顺序到达，事件记录满足 $e_i \prec_d e_{i+1}$ $(1 \le i < n)$。但物理磁盘出于性能调度的考虑，并不要求严格按照这一顺序处理 I/O 请求 [8, 111]。当上层系统需要对执行顺序进行控制时，可以发送路障指令 (在 Linux 块设备 I/O 中由 `REQ_FLUSH` 或 `REQ_FUA` 标志指定)。路障指令 e_i 保证所有 $e_j \prec_d e_i$ $(j < i)$ 在 e_i 执行前被保存到磁盘。

[1] 否则，无需系统崩溃即会触发相应的缺陷。

磁盘 D 是从扇区号到其内容的映射 (类似于共享变量 X_s)。对于扇区 $s \in \{1, 2, \ldots\}$，用 $D(s)$ 表示其数据。在运行时，系统维护内部队列 Q 保存在途的请求。我们用 $U(i)$ 和 $V(i)$ 表示第 i 个请求的扇区号和数据 (类似于 $e_i.x$ 和 $e_i.v$)，其初值为空。

因此我们用四元组 (D, Q, U, V) 表示任意时刻磁盘的状态。事件序列 $E = \{e_1, e_2, \ldots, e_n\}$ 由写指令 $W(s, d)$ 和障碍指令 B 组成，其语义如下 (由于读指令不影响磁盘上的内容从而不影响崩溃一致性，故在此模型中不予讨论)：

1. 写指令 $e_i = W(s, d)$ 写入 s 扇区、数据 d。数据并不直接到达磁盘，而是将请求插入队列 Q 中：

$$(D, Q, U, V) \Rightarrow (D, Q \cup \{e_i\}, U[i \mapsto s], V[i \mapsto d])。$$

2. 障碍指令 $e_i = B$ 保证写操作被写入磁盘。当队列 $Q = \{e_p, e_{p+1}, \ldots e_q\}$ 时，令 $D_p = D$, $D_{k+1} = D_k[U(k) \mapsto V(k)]$ $(p \le k \le q)$，执行障碍指令相应更新磁盘的状态

$$(D, Q, U, V) \Rightarrow (D_{q+1}, \varnothing, U, V)。$$

3. 最后，对于任意时刻的磁盘状态 $(D, Q = \{e_i, e_{i+1}, \ldots e_j\}, U, V)$，令 $\mathcal{D}_i = \{D\}$, $\mathcal{D}_{k+1} = \mathcal{D}_k \cup \{D[U(k) \mapsto V(k)] \,|\, D \in \mathcal{D}_k\}$ $(i \le k \le j)$，其可能的崩溃现场 $\mathcal{D}_{\text{crash}} = \mathcal{D}_{j+1}$。

这一崩溃模型准确描述了 Linux 块设备 I/O 层与文件系统层之间的交互行为，其定义了所有可能的乱序 (从磁盘的角度，乱序加系统崩溃的效果相当于丢弃某个 Q 中事件的子集)。此外，这一模型假设物理磁盘是可靠的，即保存的数据不会损坏。若不作此假设，磁盘物理层的损坏将可能导致在任何系统层次均无法解决的崩溃一致性问题 [143]。

基于依赖反转的崩溃现场生成. 尽管 I/O 事件按顺序发生 (即 $e_i \prec_d e_{i+1}$)，对于连续一段不存在障碍的写事件 $E = \{e_i, e_{i+1}, \ldots, e_j\}$，将 E 中事件发生的顺序进行任意的反转均满足磁盘的语义。

基于反转少量 I/O 操作的顺序即可触发数据库系统中的崩溃一致性缺陷的观察 [142]，我们使用使用定界搜索，仅将不超过 k 个事件进行反转，如图 5.7 所示。我们首先枚举崩溃产生的时间点 j (第 2 行)，然后选择一个不超过 k 个事件的集合 (第 4 行) 进行依赖反转 (第 5-8 行)。每个生成的磁盘镜像都被挂载到本地，并用默认的文件系统检查机制对其进行恢复 (第 10-11 行) 以得到崩溃现场的集合 $\mathcal{C}$，其中每一崩溃现场 $S \in \mathcal{C} = \{\langle f_i, c_i \rangle\}$ 中名为 f_i 文件的内容为 c_i。

检测崩溃一致性. 我们的崩溃一致性测试预言基于对崩溃现场 S 的以下分析：

Input: I/O 请求序列 $\{e_1, e_2, \ldots, e_n\}$ 和搜索上界 k
Output: 崩溃文件系统快照集合 $\mathcal{C}$

$\mathcal{C} \leftarrow \varnothing$;
for $j \in \{1, \ldots, n\} \wedge (e_j \neq B)$ **do**
  令 $(D, Q = \{e_i, e_{i+1}, \ldots e_j\}, U, V)$ 为完成 $\{e_1, e_2, \ldots, e_j\}$ 后磁盘的状态;
  for $\ell \in \{0, 1, \ldots, \min\{k, j - i + 1\}\}$ **do**
    for $P \subseteq \{0, 1, \ldots, j - i - 1\} \wedge |P| = j - i - \ell$ **do**
      $D_c \leftarrow D$;
      for $s \in P \cup \{j - i\}$ **do**
        $D_c \leftarrow D_c[U(i + s) \mapsto V(i + s)]$;
      if $D_c \notin \mathcal{C}$ **then**
        $S \leftarrow \mathsf{mount}(D_c)$;
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$;

图 5.7 基于依赖反转的崩溃现场生成算法。

1. 若 S 与 $S' \in \mathcal{S}$ 完全相同，我们的基本假设表明，S 是满足崩溃一致性的崩溃现场。
2. 如果能通过一些简单的手工恢复操作将 S 转换为 $S' \in \mathcal{S}$ (如将损坏的备份文件删除或将原文件替换为有正确数值的备份文件)，则 S 也应该被认为是满足崩溃一致性的。
3. 在前两种条件均不满足时，则必须经过较为复杂 (non-trivial) 的恢复操作才能得到开发者认为正确的文件系统快照。此种情形下，我们认为 S 非常有可能暴露了崩溃一致性的缺陷。

以上分类指导我们设计了基于编辑距离 [104] 的测试预言。形式地说，崩溃现场 $S = \{\langle f_i, c_i\rangle\}$ 满足崩溃一致性，当且仅当存在 $S' = \{\langle f'_i, c'_i\rangle\} \in \mathcal{S}$ 使得 S 能通过较少数量的恢复操作 (定义如下) 转换得到 S' (假设 $\langle f, c\rangle \in S$, $c = [b_1, b_2, \ldots, b_m]$ 并且 $f' \neq f$)：

1. 创建空文件：$S \Rightarrow S \cup \langle f', \varnothing\rangle$；
2. 删除文件：$S \Rightarrow S \setminus \langle f, c\rangle$；
3. 重命名文件：$S \Rightarrow S \setminus \langle f, c\rangle \cup \langle f', c\rangle$；
4. 移动文件中连续内容：$S \Rightarrow S \setminus \langle f, c\rangle \setminus \langle f', c'\rangle \cup \langle f, [b_1, \ldots, b_m, b'_p, \ldots, b'_q]\rangle \cup \langle f', [b'_1, \ldots, b'_{p-1}, b'_{q+1}, \ldots b'_{m'}]\rangle$，其中 $[b'_p, \ldots, b'_q]$ 是 f' 内容的连续子串。

直观地说，崩溃现场 CS (Crash Snapshot，即 $S \in \mathcal{D}_{\text{crash}}$) 满足崩溃一致性当且仅当它能通过较少数量的编辑操作转换到某个 ES。否则，若需大量操作或无法实现此种转换，我们则认为该 CS 反映了崩溃一致性的缺陷。

然而，精确计算编辑距离是 NP-完全问题 (可以将 bin-packing 问题规约到此问题)。因而我们进一步对这一定义进行放松，获得能在线性时间预处理、$O(1)$ 时间内进行崩溃一致性验证的高效算法：我们定义崩溃现场 S 满足崩溃一致性，当且仅当它能通过有限步编辑操作转换为某个满足一致性的 S'。

这一放松的定义等价于存在崩溃现场 S 到 $S' \in \mathcal{S}$ 的满射 (injective mapping)，即存在 $S' \in \mathcal{S}$ 满足对于每一个字节数值 $\sigma \in \Sigma$，

$$\sum_{\langle f,c\rangle \in S'} \Big|\{j \mid c_j = \sigma\}\Big| \le \sum_{\langle f,c\rangle \in S} \Big|\{j \mid c_j = \sigma\}\Big|。$$

进而，我们只需为每一文件 $\langle f, c\rangle$ 在线性时间计算出每一 σ 出现的次数，即可在 $O(|\Sigma|) = O(1)$ 时间内判定崩溃现场的一致性。

测试用例增强. 文件系统可能由于其内部实现在无意中保证了某些操作的原子性和持久性而导致各类检测算法的漏报。我们使用测试用例增强技术打破这些偶然发生的原子性和持久性，从而能够在不使用特殊输入 (例如足够大的文件) 的情况下检测这些缺陷。

鉴于此类漏报产生的原因是文件系统实现时偶然导致的原子性和持久性，因此只需在这些操作之间插入全系统的同步操作 (`sync`)，即能暴露此类缺陷。这样的同步操作对程序执行的正确性没有影响 (程序应当能和任意的 `sync` 并发执行)，却可生成更多可能的崩溃现场，以识别程序中潜在的崩溃一致性缺陷。

测试增强在单独的一次程序执行中完成，称为“测试增强执行”。在测试增强执行中，我们劫持可能导致数据丢失的系统调用 (`ftruncate`, `open`，且这些系统调用通常包含在库函数中，从而导致开发者不知其可能引发崩溃一致性缺陷)，然后插入 `sync` 操作。对测试增强执行对应的 I/O 记录用图 5.7 所示的算法生成崩溃现场并用测试预言进行崩溃一致性检测。

5.2.1.3 系统实现与实验评估

我们实现了上述崩溃一致性自动检测算法，并将工具 C^3 开源发布[1]。由于我们对程序/系统的修改位于系统调用和文件系统层，无需对运行时系统进行适配，因而 C^3 能用于验证各类应用程序的崩溃一致性。我们希望 C^3 工具能对开发者提升软件质量有所帮助。

[1]开源工具主页：http://jiangyy.github.io/c3/。

测试输入. 给定程序的输入，C^3 会多次使用这一输入运行程序。测试输入由 Python 编写的测试脚本指定，提供：(1) 一个初始的文件系统镜像、(2) 执行程序的方法、(3) 程序运行时给定的输入 (输入包含文件或键盘/鼠标事件)。

崩溃现场生成. 类似于 `eXplode` [131]，C^3 使用虚拟磁盘收集 I/O 操作。所有应用程序的读写操作都必须发生在该虚拟磁盘上。测试时，我们首先在指定的目录下挂载虚拟磁盘，然后在该目录执行测试脚本。测试脚本结束后，C^3 通过对虚拟磁盘的 `ioctl` 获得 I/O 操作记录，然后根据这些记录恢复出文件系统镜像、挂载后进行快照得到 CS。

由于 C^3 只验证虚拟磁盘上文件的崩溃一致性，因而测试脚本必须确保被测程序的文件操作都发生在虚拟磁盘上。现实中，程序可能修改某些由全局路径指定的文件 (如`/install/path/.config`)，若开发者意图验证这类文件操作的崩溃一致性，只需将这类文件建立在虚拟磁盘上，并用符号链接替代全局路径指定的文件。

崩溃一致性缺陷检测. C^3 使用 `ptrace` 对应用程序运行时的系统调用进行劫持。因文件系统并不保证文件操作的原子性和持久性，ES 无法用虚拟磁盘获得。又因文件系统在内存中维护满足文件操作原子性的镜像 (其需要对上层表示出文件操作顺序执行的语义)，因而我们在劫持元数据系统调用后暂停被测程序，同时对文件系统进行扫描以实现对 ES 的收集。

所有 CS 和 ES 生成完毕后，我们对其进行比较。当我们发现多个被认为存在崩溃一致性缺陷的 CS 时，我们计算 CS 到某个 ES 的最小编辑距离 d (在我们放松的定义中，即为能使编辑操作实现需要添加的最少字节数)。在所有的 CS 中我们找到 d 的最大值，并将这一镜像报告给开发者：这一 CS“丢失”了最多数量的信息，从而更倾向于是一个真正的崩溃一致性缺陷。

最后，为了避免来自程序运行环境不确定性的影响，我们仅报告 $d \geq 32$ 的 CS。例如，程序运行时可能将一些特定的运行时信息 (如时间戳等) 写入文件。这类信息每次执行均不相同，因而导致少量的文件内容无法用编辑操作恢复。若某一 CS 确实存在崩溃一致性缺陷，其通常丢失了元数据或整个扇区的数据，因 32 字节仅占一个扇区大小的 6.25%，所以这一处理既有效避免了误报，同时也具有较低的漏报风险。

实验设计. 我们使用两类 25 个知名开源应用程序对 C^3 进行了实验评估：功能程序包括在命令行中使用的对文件数据进行处理的程序 (包含 make, gzip, sed, indent 等 10 个程序)，生产力工具包括编辑用户数据的终端或图形界面交互程序 (包含 TeXstudio, Atom, Libreoffice 等 15 个程序)。

表 5.3 C^3 在 14 个应用中检出的崩溃一致性缺陷。粗体为开发者前所未知的缺陷。

类型	应用名称	版本	缺陷报告	增强	后果	d (字节)
功能程序	Gmake	4.1	46193 (savannah)		编译失败	7.33K
	Gzip	1.6	**22770** (debbugs)		数据丢失	5.04K
	bzip2	1.0.6	**N/A** (email)		数据丢失	8.56K
	Coreutils sort	8.21	**22769** (debbugs)	✓	数据丢失	23.9K
	Perl	5.22	**127663** (perlbug)		数据丢失	17.4K
	Shelve	2.7.11	**25442** (bug tracker)		数据损坏	907
生产力工具	Gimp	2.8.14	763124 (bugzilla)	✓	数据丢失	188K
	CuteMarkEd	0.11.2	**285** (github)	✓	数据丢失	5.61K
	TEXmaker	4.5	**1553361** (launchpad)	✓	数据丢失	1.61K
	TEXstudio	2.10.8	**1693** (sourceforge)	✓	数据丢失	1.61K
	Ted	1.0	**45** (github)		数据丢失	4.10K
	jEdit	5.1.0	**3952** (sourceforge)		数据丢失	1.61K
	GitHub Atom	1.5.3	10609 (github)	✓	数据丢失	1.61K
	Adobe Brackets	1.5.0	**12103** (github)	✓	数据丢失	1.61K

对于每个程序，我们使用一个典型测试用例对其进行测试，并用 C^3 进行崩溃一致性检测，再人工检查 C^3 的缺陷报告结果以评估其有效性。有效性评估包含两方面内容：(1) 能否检出真实程序中的崩溃一致性缺陷；(2) 测试增强技术是否能有效找到更多崩溃一致性缺陷。

所有实验都在典型的开发环境中进行：运行 Ubuntu Linux 14.04 (Kernel 4.2) 的虚拟 CPU、2GB 内存的虚拟机中。实验在目前主流 Linux 发行版默认的 ext4 文件系统下进行。

实验结果. 针对 C^3 有效性的实验结果如表 5.3 所示。对于 25 个实验程序，在仅使用一个典型简单测试用例的情形下，C^3 总共报告了 14 个不一致的崩溃现场 (所有崩溃现场的 d 均远超过 32 这一阈值)。我们分析了所有的缺陷报告，其均确认了数据损坏或丢失的存在。我们向开发者提交了所有 14 个缺陷报告 (表中第 4 列)，其中 8 个缺陷被开发者确认为前所未知，3 个为已知缺陷 (例如已在上游开发版本中被修复)。在 14 个缺陷中，7 个需要测试增强才能触发 (表中第 5 列)。这些实验结果证实了 C^3 工具的有效性。

所有缺陷均会导致严重的数据丢失或损坏。12/14 个缺陷 (gzip, bzip2, sort, perl 和所有生产力工具) 会导致用户数据在系统崩溃后完全丢失，甚至其中的有些缺陷已被用户触发 (如 GitHub Atom 开发者已经收到由同一问题导致的缺陷

报告)。Python Shelve 的崩溃一致性缺陷会导致无法解析的损坏数据库，其中 Shelve 的后端之一 GDBM 已于过去的研究中发现崩溃一致性缺陷 [111]。而我们进一步发现 Shelve 的其他数据库后端也同样存在类似缺陷。对于 GNU make，崩溃一致性缺陷可能导致损坏的编译目标和更新的时间戳，从而最终造成不正确的编译结果。

小结. 本节提出了基于依赖反转的崩溃现场生成、基于编辑距离的崩溃一致性测试预言和基于原子性破坏的测试增强技术，实现了应用程序崩溃一致性的全自动检测，在知名开源应用程序中发现了前所未知的崩溃一致性缺陷。更详细的算法描述和实验评估在我们的论文 [77] 中讨论。

5.2.2 移动应用的并发缺陷暴露

5.2.2.1 问题定义与解决思路

移动应用并发缺陷的暴露. 移动计算设备 (如手机) 已经成为现代人日常生活不可或缺的一部分。为了提高移动应用的性能，降低其响应时间，在移动应用中引入并发性已成为其开发的主流趋势 [4]。本节讨论如何有效地在测试环境中使用较少的代价暴露移动应用中的并发缺陷。

为防止引入并发缺陷，移动应用所运行的系统和框架使用了简化的并发模型。例如，Android 系统 (本节后续讨论均基于 Android 系统展开，但相关技术可推广至其他类似的移动系统) 采用了基于事件的机制，所有 UI 事件必须在主线程中串行处理，耗时的操作则必须通过异步事件进行管理。尽管这一机制减少了并发移动应用的不确定性 (事件在主线程被原子地执行，因此彻底消除了 UI 组件之间的数据竞争)，但随着应用功能日益复杂，实际应用中仍存在多种多样的并发缺陷，并威胁其用户体验。

为了检测这类缺陷，已有技术 [23, 68, 98] 扩展了数据竞争 (data race) 的定义，提出“事件竞争”的概念及其检测方法。然而，使用此类技术检测移动应用中的并发缺陷需要高质量的执行轨迹作为输入，而一般开发者通常难以提供可能触发并发缺陷的执行轨迹；此外，此类技术只能确定两个事件是否冲突 (访问共享资源且至少有一个是资源的修改事件，类似于 $e_i \asymp e_j$)，但存在冲突不代表存在并发缺陷，误报率较高。如何降低其误报率是非常具有挑战性的研究问题 [23]。

本节提出能有效暴露移动应用并发缺陷的主动测试技术 AATT，其利用移动应用事件驱动的特征，同时生成移动应用的输入和并发事件的调度，以实现对应用输入空间和调度空间的同时探索。

```
item.setOnClockListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
    PopupMenu popup = new PopupMenu(context, item);
    popup.inflate(R.menu.mission);
    Menu menu = popup.getMenu(); // 不加检查地创建菜单，导致重复创建
    MenuItem del = menu.findItem(R.id.del);
    del.setVisible(true);

    popup.setOnMenuItemClickListener(
      new PopupMenu.OnMenuItemClickListener() {
        @Override
        public boolean onMenuItemClick(MenuItem item) {
          if (item.getItemId() == R.id.del) {
            manager.deleteItem(downloadItem.pos); // 被执行两次导致崩溃
            return true;
          }
          return false;
        }
      });
    popup.show();
  }
});
```

图 5.8 GigaGet 中包含并发缺陷的代码。

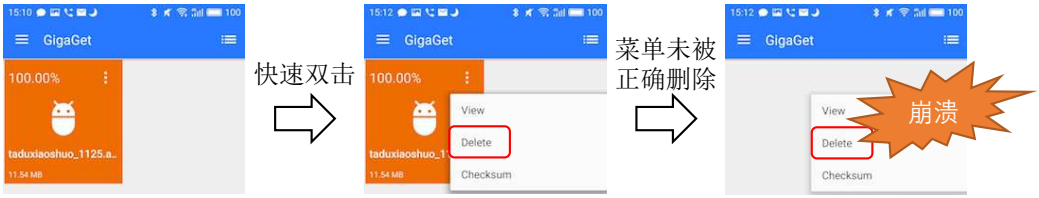

图 5.9 GigaGet 中并发缺陷的暴露。

案例分析与研究挑战. 图 5.8 包含了来自 Gigaget 应用的原子性违反并发缺陷的代码片段，图 5.9 展示了缺陷的触发方法。GigaGet 以磁贴的方式显示多个从互联网下载的文件，单击磁贴右上角的按钮触发 `item.onClick` 事件显示新创建的弹出菜单，在 `onClick` 执行后磁贴上的按钮将不再有效。由于 Android 并发机制并不保证点击事件和它所对应的 `onClick` 事件之间的原子性，若系统在短时间内连续收到两个点击事件，则 `onClick` 会被执行两次而创建出两个弹出菜单。在两个弹出菜单分别点击“删除”将导致应用崩溃。

这一并发缺陷难以被现有的技术暴露。基于状态空间遍历的移动应用测试技术 [16] 不考虑并发事件，无法检出此类缺陷；基于事件竞争 [23, 68, 98] 的技术需要包含快速点击同一按钮的执行轨迹才能检出事件竞争，但普通开发者难

以提供此类执行轨迹；尽管随机测试 [5] 有可能触发这一并发缺陷，但在有限的测试资源内生成触发缺陷的事件序列概率极低。

解决思路. 根据上述分析，移动应用并发缺陷的暴露需要三个条件：(1) 移动应用处于特定的状态；(2) 某个特定的事件序列被触发；(3) 事件序列及其相关的并发事件按照某种特定的调度顺序发生。

为了同时达成这三个目标以触发并发缺陷，我们使用二阶段分析。第一阶段对移动应用的状态空间进行系统地遍历。此时，我们不考虑移动应用并发事件的调度，但对其执行轨迹进行记录。我们对执行轨迹进行分析，找出其中访问冲突资源事件所对应的访问点 (access point, AP)。基于这些分析结果，第二阶段再次对移动应用的状态空间进行遍历，但在此次遍历时用依赖反转技术系统化地枚举 AP 的调度，以暴露移动应用中的并发缺陷。

5.2.2.2 暴露移动应用并发缺陷的二阶段分析

获得存在潜在冲突的输入事件. 我们扩展第 2.1 节描述的系统模型使之能支持移动应用。首先，系统中的共享变量 X_s 不仅包含共享内存，也包含各类其他共享资源 (文件、数据库等)。访问 X_s 中变量的指令称为访问点 (access point, AP)。对资源的访问也可分为读/写操作，因此事件 $e \in E$ 既可能是对共享内存的访问，也可能是对共享资源的访问。其次，移动应用的运行是输入事件 (如点击/滑动屏幕或系统事件) 驱动的，因此每一共享资源访问均能对应某个输入事件。对于输入事件 i，令由 i 产生的事件中的共享资源访问事件为 $E_i \subseteq E$。

我们执行移动应用，获得其执行轨迹 τ 中的事件集合 E 及事件发生的顺序关系 $\prec_d$。对于两个输入事件 i_1 和 i_2 (i_1 可以等于 i_2)，如果存在 $e_p \in E_{i_1}$，$e_q \in E_{i_2}$ 满足 $e_p \asymp e_q$，则称输入事件 i_1 和 i_2 是冲突的。我们求出移动应用每一状态 s 上可能导致冲突的输入事件组合 $\mathcal{C}_s = \{(i_1, i_2)\}$，然后在第二阶段通过依赖反转技术对冲突的输入事件组合及其 AP 进行调度以暴露并发缺陷。

基于依赖反转的调度生成. 我们提出面向调度的深度优先状态空间搜索算法 (SO-DFS) 来实现移动应用的并发缺陷暴露 (图 5.10)。算法主体是对应用状态空间的深度优先遍历 (第 11-17 行)，在移动应用到达每一不同状态后尝试其所有可能的输入事件，并在到达之前未访问的状态后进行递归处理。

在到达每一状态 s 后，算法第 6-10 行对事件调度进行系统枚举。我们尝试所有 s 上可能发生输入事件的 k-组合 (在实际中为了取得最佳缺陷暴露有效性和运行时开销的权衡，我们选用 $k = 2$)。对于输入事件序列 $i_1, i_2, \ldots, i_k$，我们首先在状态 s 执行这些事件 (第 7 行)，并记录执行轨迹 τ。若 τ 中存在冲突的 AP 访问 e_p, e_q，则分两次探索冲突 AP 的不同调度，其中一次满足 $e_p \prec_{tot} e_q$ (第 9

```
   Input: 搜索组合上限 k
1  S ← ∅;
2  Function SO-DFS(s, π)
3  begin
4      S ← S ∪ {s};
5      I ← {i | ∃i_1.(i, i_1) ∈ C_s};
6      for (i_1, i_2, ... i_k) ∈ I 且 (i_j, i_{j+1}) ∈ C_s do
7          τ ← execute(i_1, i_2, ..., i_k);
8          for 有冲突的 AP 访问 e_p, e_q ∈ τ do
9              τ ← execute(i_1, i_2, ..., i_k)，同时满足 e_p ≺_d e_q;
10             τ ← execute(i_1, i_2, ..., i_k)，同时满足 e_q ≺_d e_q;
11     for 输入事件 i ∈ I do
12         sendEvent(i);
13         s' ← getCurrentState();
14         if s' ∉ S then
15             SO-DFS(s', π :: ⟨i⟩);
16         if s' ≠ s then
17             restore(s, π);
```

图 5.10 实现移动应用并发缺陷暴露的 SO-DFS 算法。

行)，另一次对依赖进行反转，满足 $e_q \prec_{tot} e_p$ (第 10 行)，从而实现系统化的移动应用并发缺陷暴露。

5.2.2.3 系统实现与实验评估

系统实现. 我们实现了 AATT 工具用于移动应用并发缺陷的自动暴露。我们使用 E-GreenDroid [126] 实现事件输入的生成和对应用状态空间的遍历；使用 `Uiautomator` 发送输入事件；使用修改过的 Android `ART` 虚拟机收集移动应用的执行轨迹。我们收集包含 Java 对象的静态/对象域访问和方法调用序列的执行轨迹，再通过方法调用的参数识别出冲突资源 (如数据库) 的访问。

我们使用基于 ATT 框架 [99] 的程序插装技术实现 SO-DFS 算法。首先，我们使用类似 A^3E 技术 [16] 深度优先策略遍历移动应用的所有状态 (具有相同 UI layout 的状态被视为等价的状态，遍历所有等价状态的每条出边后终止)，然后在并发事件的开头处插入信号量 P/V 操作以实现依赖的反转。为了避免无意义的状态空间搜索，每一 AP 在执行过程中至多只能反转一次。

表 5.4 评估 AATT 所用的两组实验对象：有已知并发缺陷的应用、随机选取的应用。

有已知并发缺陷的应用		
应用名称	类别	代码行数
vlillechecker	旅行	5,330
AnyMemo	教育	29,871
OIFileManager	效率	3,484
Tomdroid	效率	12,233
SunShine	天气	26,472
MyTrack	通讯	43,372
ChatSecure	通讯	68,681
Feedex	新闻	10,151
sgtpuzzles	游戏	5,780
K-9 Mail	通讯	95,098
todowidget	效率	689
AAT	旅行	43,725
Douya	社交	34,725
weiciyuan	社交	78,936
FBReader	参考	16,578

随机选取的应用		
应用名称	类别	代码行数
2buntu	新闻	963
aarddict	参考	2,077
aNarXiv	参考	3,357
andiodine	工具	1,502
Down	工具	2,045
DroidWeight	健康	5,078
externalIP	工具	2,416
falling blocks	游戏	1,763
GigaGet	工具	3,123
HostIsDown	工具	631
KindMind	生活	5,510
LilyDroid	社交	10,471
MultiPing	工具	547

实验设计. 我们使用两组开源 Android 应用对 AATT 的有效性和性能进行评估。实验对象分为两组：

1. 从 Github 上通过关键字搜索选取的 15 个具有已知并发缺陷的 Android 应用。将这些应用作为基准程序用于测试 AATT 是否能成功暴露已知缺陷，以度量 AATT 暴露并发缺陷的有效性。
2. 在 F-droid 等开源应用发布网站随机选取的 13 个流行开源应用 (存在并发性从而可能包含并发缺陷)。运行 AATT 对其进行并发缺陷暴露。

实验对象如表5.4 所示。对于每个实验对象，我们对比 AATT 和其他两类移动应用的测试技术：(1) 不关注并发事件调度的 DFS [16] 技术 (我们增强了 DFS，使之能快速连续触发相同事件)；(2) 工业级随机测试工具 Monkey。在实验中，AATT 和 DFS 均运行至暴露第一个并发缺陷或状态空间搜索结束 (所有状态均被探索过) 为止，Monkey 则配置为 AATT 2 倍的时限。所有实验均在具有 4GB 内存和 Intel i5 处理器的 PC 机及一台 Google Nexus 5 设备上完成。

表 5.5 AATT 在两组实验对象上的评估结果。

应用名称	缺陷类型	AATT		DFS		Monkey	
		时间	缺陷	时间	缺陷	时间	缺陷
vlillechecker	AV	1,425	✓	2,835	-	2,850	-
AnyMemo	AV	872	✓	758	-	1,744	-
OIFileManager	AV	980	✓	2,258	-	1,960	-
Tomdroid	OV	2,375	-	2,036	-	4,750	-
SunShine	OV	635	✓	1,926	-	1,270	-
MyTrack	AV	456	✓	778	-	912	-
ChatSecure	OV	1,358	✓	2,996	-	2,716	-
Feedex	AV	1,028	✓	3,768	-	2,056	-
sgtpuzzles	AV	1,256	-	947	-	2,512	-
K-9 Mail	AV	1,263	✓	1,274	-	2,516	-
todowidget	OV	685	✓	568	-	1,370	✓
AAT	OV	1,768	-	1,528	-	3,536	-
Douya	AV	1,029	-	1,128	-	2,058	-
weiciyuan	OV	637	-	537	-	1,274	-
FBReader	OV	858	✓	1,433	-	1,716	-
2buntu	-	1,876	-	875	-	2,832	-
aarddict	AV	275	✓	1,252	-	316	-
aNarXiv	AV	376	✓	956	-	406	-
andiodine	AV	2,450	✓	2,258	-	4,494	-
Down	AV	98	✓	281	-	26	✓
DroidWeight	-	6,247	-	2,320	-	12,470	-
externalIP	-	450	-	223	-	899	-
falling blocks	AV	226	✓	158	-	430	-
GigaGet	AV	643	✓	271	✓	1,310	-
HostIsDown	AV	237	✓	819	-	462	-
KindMind	-	2,679	-	1,478	-	5,280	-
LilyDroid	AV	1,023	✓	11,909	-	1,590	-
MultiPing	OV	425	✓	440	-	750	-

实验结果. 实验评估结果如表 5.5 所示。首先，AATT 检测出了 10/15 个存在已知缺陷的应用和 9/13 个随机应用中的并发缺陷，效果远好于 DFS (1/28) 和 Monkey (2/28)，其能够有效地暴露移动应用中的并发缺陷。对于随机挑选的应用，经人工确认，AATT 暴露的并发缺陷均为真实存在的。我们将缺陷报告反馈给开发者，已经得到 3 个确认反馈。此外，尽管增加了调度探索的部分，实际应用中 AATT 的耗时与 DFS 仍处于同一量级，因此在实际中有较好的应用前景。

此外，我们还对 28 个应用中存在的 19 个并发缺陷进行了人工分类整理，分类结果见表 5.5 第 2 列，其中 AV 代表原子性违反 (Atomicity Violation)、OV 代表顺序违反 (Order Violation)。可见，导致移动应用并发缺陷的主因仍然符合 Lu 等人提出的并发缺陷分类框架 [94]。

小结. 本节在移动应用的测试场景下，针对现有移动应用并发缺陷检测技术需要特定执行轨迹的不足，通过同时生成移动应用的输入和并发事件的调度，实现移动应用并发缺陷的有效暴露技术 AATT。AATT 成功检出了流行开源应用中前所未知的并发缺陷。

5.3 小结

本章讨论了我们基于访存依赖展开的一系列软件测试、分析方面的研究：

1. 基于访存依赖的动态分析。基于缓存的执行重放技术 CARE 利用并发程序的线程局部性，实现部分访存依赖的获取 (只需要记录写后写和部分写后读依赖)，相比于互斥锁有多达数十倍的性能提升、依赖数减少为数百分之一；基于二分组协议的数据竞争检测和假共享检测利用约减后的访存依赖，能够大幅减少检测调用的数量和元数据维护开销；
2. 基于依赖反转的软件测试技术。C^3 实现了软件中由于文件系统操作顺序反转导致的原子性违反缺陷的全自动检测，检出了知名开源软件中前所未知的崩溃一致性缺陷；AATT 实现了移动应用输入空间和调度空间的同时探索，进而实现了高效的并发缺陷暴露，亦检出了流行开源移动应用中前所未知的并发缺陷。

第六章　访存依赖获取复杂性的讨论

我们已经知道，若只在线程本地记录共享内存访问的事件 (不记录其全局顺序)，即便记录了每次读写操作的数值以及各个事件在线程内发生的顺序，要想得到满足顺序一致性的访存依赖仍然是 NP-完全问题 [57]。

二十多年来对访存依赖获取技术的研究在减少记录的数量和开销方面取得了一定的进展。这些研究工作可被分为两类：

1. 在程序插装或运行系统修改中引入额外同步操作 (如互斥锁、原子操作) 的技术。目前，仅有这类技术可以在多项式时间内获得满足顺序一致性的访存依赖 [28]。
2. 在程序插装或运行系统修改中不引入额外同步操作的技术。目前已知的这类技术推导访存依赖的时间复杂性都是 NP-难的，通常需要借助搜索算法或约束求解器 [74, 87]。

例如，Stride [144] 能在读操作不引入任何同步的情况下，在多项式时间内合成访存依赖，而为了保证算法的正确性，写操作必须被同步；Octet [28] 和 CARE [78] 等技术使用乐观锁，实际程序大部分的事件都不需要使用同步，但总存在需要同步的 slow-path。相反，那些不引入任何额外同步的技术，要么不能保证获得一致的访存依赖 [109]，要么需要求解 NP-难问题 [87]。

这一现象自然引发了我们对访存依赖获取复杂性问题的兴趣。我们能否做到只对并发程序作出 wait-free 的修改，同时又能在多项式时间内获取满足顺序一致性的访存依赖？

本章对这一问题进行了探讨，并据此给出了一个否定的猜想—没有免费 (只进行 wait-free 的修改) 的午餐 (在多项式时间内获得满足顺序一致性的访存依赖)。首先提出程序修改的模型，然后形式化地提出 No-Free-Lunch (NFL) 猜想。基于这一框架，我们给出顺序一致性判定 (VSC) 问题的一个新证明，并据此证明 NFL 猜想的一个特殊情形。最后，我们从复杂性结论出发进行了对访存依赖获取问题的一些深入思考。

6.1 No-Free-Lunch (NFL) 猜想

6.1.1 问题定义

本章旨在讨论获取访存依赖的理论复杂性，试图证明在一定的条件下，获取访存依赖的时间复杂性具有 NP-完全性的特征。由于我们的理论框架和技术

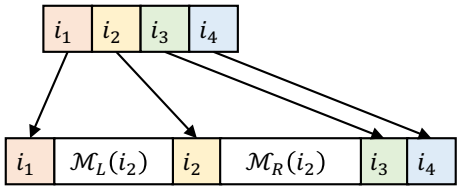

图 6.1 复杂性讨论中对并发程序的修改示意图。

框架力图包容各类访存依赖获取技术，对“访存依赖获取”问题采取了广义的定义：只需获得的访存依赖 $\prec_d$ 确定所有 $e_i \asymp e_j$ 的顺序即可，允许与实际执行的顺序 $\prec_{tot}$ 不一致、不等价，从而将离线合成等技术纳入框架中统一处理。为了研究访存依赖获取的理论复杂性，我们首先对本章讨论的“访存依赖获取”问题范围进行精细化限定，以便能严格证明理论方面的结论。

具体而言，我们限定本章讨论的访存依赖获取问题为“通过对并发程序进行插装或对其运行时系统进行修改，并在线程本地记录读/写事件，从而获得能满足顺序一致性访存依赖的问题”。我们首先形式化地定义“并发程序的修改”为一类具有代表性的程序插装或对运行时系统进行修改的操作 (第 6.1.1.1 节定义对并发程序修改的限制，第 6.1.1.2 节定义执行并发程序所能获取记录的形式)，在此基础上进一步定义狭义的“访存依赖获取问题”(第 6.1.1.3 节)。

6.1.1.1 对并发程序修改的限制

获取访存依赖是为了分析并发程序的执行轨迹。因此，在获取访存依赖的过程中，对并发程序修改 (即对程序进行的插装或对运行系统造成的修改) 不应破坏其原有的语义。例如，对并发程序的修改不应删除原程序中的任一指令，否则会导致原本正确的程序在运行时崩溃。本节首先提出一个合理的 (能代表目前研究工作对并发系统作出的修改) 对并发程序进行修改的模型，其定义了所有允许对并发程序作出的修改。

我们限制对并发程序的修改只能在访存指令前后插入一定数量的、不修改原有程序状态 (共享变量 X_s 和线程局部变量 X_t 均不能修改) 且能独立运行的指令序列，其直观描述见图 6.1 所示。

形式化地说，我们定义对并发程序 P 的修改为四元组 $\langle X_s^+, X_t^+, \mathcal{M}_L, \mathcal{M}_R \rangle$：

1. X_s^+ 为增加的全局共享变量集合，用于在运行时维护全局的动态信息。
2. X_t^+ 为增加的局部变量的集合，每一线程均可增加其局部变量。修改后的程序可以使用这些局部变量进行线程本地的运算。
3. $\mathcal{M}_L, \mathcal{M}_R$ 为对指令序列进行实际的修改，其含义为在程序 P 中每条指令 $i \in I$ 的前、后分别插入指令序列 $\mathcal{M}_L(i)$、$\mathcal{M}_R(i)$。

为了防止 P 的语义被破坏，我们对 P 的修改作出以下限制：

1. 仅能对读/写共享内存指令进行修改。即对于仅有线程本地效果的分支指令和局部运算指令 i，有 $\mathcal{M}_L(i) = \mathcal{M}_R(i) = \varnothing$。
2. 插入的指令 ($\mathcal{M}_L(i)$ 和 $\mathcal{M}_R(i)$) 对共享变量的访问仅限于 X_s^+、对线程局部变量的访问仅限于 X_t^+。
3. $\mathcal{M}_L(i)$ 中跳转的目标指令只能来自 $\mathcal{M}_L(i)$；$\mathcal{M}_R(i)$ 中跳转的目标指令只能来自 $\mathcal{M}_R(i)$。

给定程序 $P = \langle T, X, I\rangle$ ($I = \{i_1, i_2, \ldots, i_n\}$),对 P 应用修改 $\langle X_s^+, X_t^+, \mathcal{M}_L, \mathcal{M}_R\rangle$ 将得到修改后的程序 $P' = \langle T, X', I'\rangle$，满足

$$\begin{aligned} X' &= X \cup X_s^+ \cup X_t^+, \\ I' &= \mathcal{M}_L(i_1) :: \{i_1\} :: \mathcal{M}_R(i_1) :: \mathcal{M}_L(i_2) :: \ldots :: \{i_n\} :: \mathcal{M}_R(i_n), \end{aligned}$$

其中 :: 为序列的拼接操作。

由于我们限制程序的修改只能在 P 的访存指令前后添加，因此对于每一 P 中的指令 $i \in I$，都存在 $i^+ \in I^+$ 与之对应。由于在 I^+ 中插入的指令不会影响 X 中变量的状态 (由限制 2、3 共同保证)，故若执行 P' 得到执行轨迹

$$\tau' = \left[\sigma'_0 \xrightarrow{t'_1} \sigma'_1 \xrightarrow{t'_2} \ldots \xrightarrow{t'_n} \sigma'_n\right],$$

若在 τ' 中删除所有执行插入的指令 (即 $\mathcal{M}_L(i)$, $\mathcal{M}_R(i)$ 中的指令) 后产生的状态，并对于每一个 $\sigma'_i = \langle V'_i, PC'_i, E'_i\rangle$，将 V'_i 中 X_s^+, X_t^+ 变量的值删除、将 E'_i 中与 X_s^+ 有关的事件删除，则可以得到 P 的执行轨迹

$$\tau = \left[\sigma_0 \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \ldots \xrightarrow{t_m} \sigma_m\right].$$

换言之，在我们的定义下，对任意一次执行修改后程序 P' 的执行轨迹，都对应了一条 P 的执行轨迹，即修改保证不破坏程序的语义。在一次访存操作之前，我们允许执行任意数量的局部变量或全局变量访问和运算，因而对并发程序作出修改的这一限制有着很强的表达能力。例如，我们可以将一次访存事件用自旋锁包裹，以序列化所有的访存事件。事实上，综述框架中的绝大部分技术都能在满足这一限制的前提下被实现，如 LEAP [69]、Stride [144]、Octet [28] 等。

6.1.1.2 执行并发程序获得记录的限制

同理，我们也对并发程序运行时获取访存依赖能够得到的记录进行限制。如果我们允许在并发系统运行时严格记录它的行为 (即每一个 $\sigma_i \to \sigma_{i+1}$ 的状态

迁移)，就能得到执行轨迹中事件发生的全序 $\prec_{tot}$，从而使访存依赖获取问题变成平凡 (trivial) 的，但这样的记录在现实系统中是无法实现的。

根据目前计算机系统的设计，每个线程通常只能在本地对自身的状态予以记录，而不存在所有指令执行的全局视角。因此我们限制访存依赖获取的记录只能发生在线程本地，且满足：

1. 执行读指令 $x_k^t = R(x_{k'})$ 只能获得共享变量 $x_{k'}$ 的地址 k' 和读出的值 (保存在局部变量 x_k^t 中)。
2. 执行写指令 $W(x_k, x_{k'}^t)$ 时只能获得共享 x_k 的地址 k 和写入的数值 (保存在局部变量 $x_{k'}^t$ 中)。

这些信息恰好对应了并发程序执行后访存事件的集合 E：对于事件 e，它要么是读事件 $R(t, x, v)$，要么是写事件 $W(t, x, v)$，这些信息均能在指令被执行时获取。此外，由于记录发生在线程本地，因此每一线程能记录访存事件发生的顺序，即拥有程序顺序 $\prec_{po}$。

因此，我们限制并发系统执行程序 P，记录仅能得到执行轨迹 τ 中的 $\langle E, \prec_{po}\rangle$。$\langle E, \prec_{po}\rangle$ 已包含足够的信息用以重现每个线程 t 自身的执行轨迹：我们从初始状态 σ_0 开始执行线程 t (其他线程均不被执行)，并根据当前指令 i 的类型分别考虑：

1. 若 i 是本地计算指令、跳转指令或共享内存写指令，则直接根据指令语义执行，更新相应的局部/共享变量的值。
2. 若 i 是读指令 $x_k^t = R(x_{k'})$，对应事件为 $R(t, x_{k'}, v)$，则令局部变量 x_k^t 的数值为 v，并取出下一条指令继续执行。

由于仅有 E 和 $\prec_{po}$ 并不包含访存依赖 (无法确定一对冲突事件 $e_i \asymp e_j$ 的先后关系)，故我们在此基础上定义访存依赖获取问题。

6.1.1.3 访存依赖获取问题

给定并发程序 P，访存依赖的获取定义为以下过程：

1. 对 P 进行修改 (满足第 6.1.1.1 节中的限制) 得到 P'，这对应了对并发程序进行插装或对其运行的并发系统进行修改。
2. 通过执行 P' 获得修改后程序的事件集合及程序顺序 $\langle E', \prec'_{po}\rangle$。由于 P' 的执行轨迹 τ' 对应了唯一的 P 的执行轨迹 τ，可求出 τ 对应的事件集合 E。

3. 通过 $\langle E', \prec'_{po}\rangle$ 求解 E 上的一个全序 $\prec_{tot}$，使 E 中的事件按此排序满足顺序一致性，即对任意读事件 e_r，都存在一个写事件 e_w 满足

$$(e_r.x = e_w.x) \wedge (e_r.v = e_w.v) \wedge \\ \Big[\forall e \in E.((e.x = e_r.x) \wedge (e.rw = w)) \rightarrow ((e_r \prec_{tot} e) \vee (e \prec_{tot} e_w))\Big]。$$

这一问题定义清晰地反映了获取访存依赖复杂性中的矛盾：一方面，为了实现高效的访存依赖获取，我们希望程序的修改对其造成的影响 (时间开销、语义改变等) 越小越好。另一方面，$\langle E', \prec'_{po}\rangle$ 没有包含任何关于冲突事件发生的顺序关系，只有对程序进行修改来获得更多的信息，才有助于确定这些事件之间的顺序，从而得到满足顺序一致性的事件排序。正如我们在技术框架中所总结的 (图 3.5)，现有的研究工作对程序造成的修改和记录越小，获取访存依赖所需要的代价就越大。例如，如果我们完全不修改并发程序 ($X_s^+ = X_t^+ = \varnothing$, $\mathcal{M}_L(i) = \mathcal{M}_R(i) = \varnothing$)，那么即便获得完整的 $\langle E, \prec_{po}\rangle$，获取满足顺序一致性的访存依赖的时间复杂性也是 NP-完全的 [57]。这自然地引发了我们对复杂性的一系列讨论。

6.1.2 NFL 猜想

在明确了访存依赖获取问题的定义后，我们就能着手从理论上研究“对程序造成的修改和记录越小，获取访存依赖所需要的代价就越大”这一权衡。其中最关键的问题即是：究竟需要对程序进行多大的修改、记录多少信息，才能实现高效的访存依赖获取？NFL 猜想 (及其被证明的特殊情况) 即是对解答这一问题做出的尝试。

6.1.2.1 VSC 问题的 NP-完全性

在并发程序测试与分析研究的早期，Gibbons 和 Korach 即注意到根据共享内存访问事件求解执行轨迹的相关理论 [57]，并提出了 VSC 问题的如下定义 (VSC 问题的直观描述如图 6.2 所示)。

定义 6.1. *(VSC, Verifying Sequential Consistency 问题)* 给定并发程序 P 执行轨迹中事件的集合 E *(*对于 $e \in E$，可知其类型 $e.rw$、发生的线程 $e.t$、访问的变量 $e.x$ 和读写的数值 $e.v$*)*，及 E 中事件的程序顺序 $\prec_{po} \subseteq E \times E$。*VSC* 问题判定是否存在一个 E 的一个全序 $\prec_{tot}$，使其满足顺序一致性，即对于任意的读

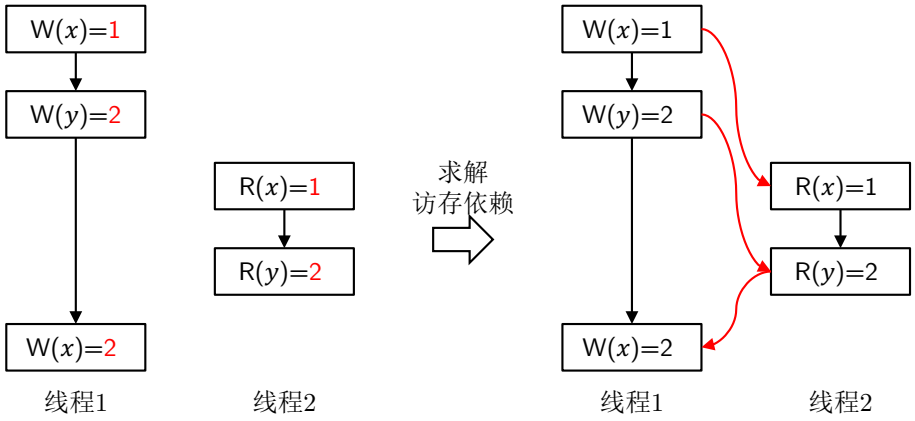

图 6.2 VSC 问题：根据 $\langle E, \prec_{po}\rangle$ 恢复满足顺序一致性的访存依赖。

事件 $e_r \in E$，都存在写事件 $e_w \in E$ 满足

$$(e_r.x = e_w.x) \wedge (e_r.v = e_w.v) \wedge \Big[\forall e \in E.\,((e.x = e_r.x) \wedge (e.rw = w)) \rightarrow ((e_r \prec_{tot} e) \vee (e \prec_{tot} e_w))\Big]。$$

直观上，VSC 问题给定若干线程以及这些线程内按次序发生的共享内存读/写事件 (每一事件包含其访问的地址和数值)，要求求解一个全局合法的调度满足顺序一致性，即每次对 x 的读事件总是得到最近一次对 x 写事件写入的值。这一问题等价于确定访存依赖 $\prec_d$，使得所有冲突的事件 $e_i \asymp e_j$ 都被确定顺序，且 $\prec_{d^*}$ 的任意线性序均满足顺序一致性。

定理 6.1. *VSC 是 NP-完全问题。*

证明. 文献 [57] 证明了对于任意一个 3-SAT 问题的实例 C，都能构造事件的集合 E 及其程序顺序 $\prec_{po}$，使得 C 有解当且仅当 $\langle E, \prec_{po}\rangle$ 存在一个满足顺序一致性的调度，此处不再赘述。 □

VSC 问题对应了在不修改并发程序 P (即 $X_s^+ = X_t^+ = \varnothing$, $\mathcal{M}_L(i) = \mathcal{M}_R(i) = \varnothing$) 的前提下获取访存依赖的情况：执行 P 将会得到其执行轨迹中访存事件的集合 E 及 E 上的程序顺序 $\prec_{po}$。定理 6.1 表明，在仅给出 $\langle E, \prec_{po}\rangle$ 的前提下，获得满足顺序一致性的访存依赖是 NP-完全问题。VSC 问题给出了“没有免费午餐”中最基本的情况：没有完全免费 (不用对程序进行任何修改) 的午餐 (在多项式时间内求解访存依赖的算法，假设 P $\neq$ NP 成立)。

6.1.2.2 NFL 猜想

VSC 问题的 NP-完全性结论对访存依赖获取技术有着深远的影响。一方面，仅使用线程局部信息实现访存依赖获取的技术都必须使用搜索算法求解访存依

赖，例如文献 [74, 87, 86] 都使用了 SMT 求解器。另一方面，在对访存依赖获取这一问题的研究中，研究者普遍认为减少同步操作的使用对提高效率是至关重要的。然而，在二十多年的研究中，研究者尚未找到一种不使用同步且在多项式时间内获取访存依赖的方法。Stride 在对读操作的修改完全避免了使用锁操作 [144]；Octet [28] 和我们的工作 RWTrace [79] 都实现了在具有局部性的访存事件上避免同步操作。但目前这些工作仍然无法完全消除同步操作的使用，总存在 slow-path 需要使用同步操作对共享内存访问指令进行包裹，以实现准确或一致的访存依赖获取，以此在多项式时间内推导出其他依赖。

这些结果使我们作出“不引入同步操作”和“多项式时间内获取访存依赖”之间具有矛盾的猜想：

猜想 6.1. *(NFL 猜想) 存在并发程序 P，满足无论对其作出怎样的修改 $\langle X_s^+, X_t^+, \mathcal{M}_L, \mathcal{M}_R\rangle$ 得到的程序 P'，只要插入的代码 ($\mathcal{M}_L$, $\mathcal{M}_R$ 中的指令序列) 是 wait-free 的 (即对于 P' 的任意调度，一个读写指令前后插入的一段代码总在有限步内结束)，那么总存在 P' 的执行轨迹 τ'，根据执行轨迹 τ' 中的事件 E' 和程序顺序 $\prec'_{po}$ 推导出 P 所对应的事件 E 满足顺序一致性的全序 $\prec_{tot}$ 是 NP-完全的。*

首先，VSC 问题是 NFL 猜想中描述最简单的特例：VSC 问题等价于对程序作出 $X_s^+ = X_t^+ = \varnothing$, $\mathcal{M}_L(i) = \mathcal{M}_R(i) = \varnothing$ 的修改，其显然满足 wait-free 性质。在此情况下，定理 6.1 已经证明了其 NP-完全性。

另一方面，如果允许对程序作出超越 wait-free 限制的修改 (如允许修改插入的代码在特定的线程调度下不会终止)，则我们可以对共享内存访问指令作出如下修改：

$$\begin{aligned} X_s^+ &= \{\ell, tsc\}, \\ X_t^+ &= \varnothing, \\ \mathcal{M}_L(i) &= \{\texttt{lock}(\ell),\ tsc = tsc + 1\}, \\ \mathcal{M}_R(i) &= \{\texttt{unlock}(\ell)\}, \end{aligned}$$

其中 ℓ 为全局的锁变量、tsc 为全局时钟的时间戳、`lock`/`unlock` 操作为基于过滤锁算法实现的自旋互斥锁 [64] (可以用共享变量的读写、线程本地计算和跳转指令实现)。这一修改保证了所有共享内存访问指令的原子性，并在临界区中记录下该访存事件的时间戳，且每一访存事件 $e \in E$ 都能对应到唯一的时间戳，从而推出与实际访存指令执行顺序完全一致的全序 $\prec_{tot}$。因此，猜想中限制对程序的修改只能插入 wait-free 的代码是一个充分 (但可能不必要) 的条件。

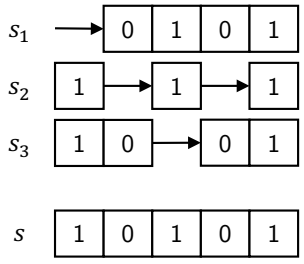

图 6.3 最短超序列问题 (SCS)：给定 01 字符串 $s_i \in S$ 和 k，判定是否存在长度为 k 的字符串 s 满足 s_i 是 s 的子序列。

直观上，这一猜想试图说明在 P $\neq$ NP 的假设下，要么需在程序修改时引入同步操作，要么必须付出超过多项式时间的代价才能获得满足顺序一致性的访存依赖。满足顺序一致性是一个“合法”执行的必要条件，满足一致性的访存依赖必然满足顺序一致性，而理解不满足顺序一致性的执行对开发者来说通常是困难的。

最后，完整证明这一猜想是非常具有挑战性的。这一猜想可以用如下逻辑公式表达：

$$\exists P.\, \forall M.\, ((\forall S.\, \text{wait-free}(P, M, S)) \rightarrow \exists \tau.\, \text{NPC}(\tau)),$$

其中 P 是程序、M 是对 P 的修改、S 是对修改后程序的调度、τ 是修改后程序的执行轨迹。这个公式有四层全称量词/存在量词的嵌套，且每一层嵌套都定义在非常复杂的问题空间 (涉及程序执行的语义、对程序修改的限制等)。我们尚未发现能够对其进行处理的数学工具。但我们仍然可以找到其中一类有代表性的特殊情形，并对其正确性予以证明。

6.2 NFL 猜想特殊情形的证明

6.2.1 VSC 问题的新证明

在证明 NFL 猜想的特殊情形之前，我们首先给出 VSC 问题的一个新的证明思路，并证明即便在并发系统仅有一个变量的前提下，判定一个事件集合是否满足顺序一致性仍然是 NP-完全的。我们的证明使用了另一个 NP-完全问题：最短超序列问题 (SCS) 作为归约的基础。SCS 的例子如图 6.3 所示。

定义 6.2. *(SCS, Shortest Common Supersequence 问题)* 给定 *01* 字符串集合 $S = \{s_1, \ldots, s_n\}$ 和整数 k，判定是否存在长度为 k 的 *01* 字符串 s，满足对于任意的 $s_i \in S$，s_i 都是 s 的子序列 *(subsequence)*。a 是 b 的子序列当且仅当能通过从 b 中删除一定数量的字符得到字符串 b'，使得 $a = b'$ 被满足。

Räihä 等人证明了 SCS 是 NP-完全问题 [114]，描述为如下定理：

定理 6.2. *SCS 是 NP-完全问题。*

同理，我们可以定义另一判定版本的 SCS 问题，LSCS 问题如下：

定义 6.3. *(LSCS, Limited Shortest Common Supersequence 问题) 给定 01 字符串集合 $S = \{s_1, \ldots, s_n\}$ 和整数 k, p，判定是否存在长度为 k 的 01 字符串 s，且其中恰好有 p 个 1，并满足对于任意的 $s_i \in S$，s_i 都是 s 的子序列 (subsequence)。*

推论 6.1. *LSCS 是 NP-完全问题。*

证明. 将 SCS 问题的任意实例 (S, k) 在多项式时间内归约到 LSCS 问题，分两种情况讨论：

$k \geq \sum_{i=1}^{n} |s_i|$。这一情形是平凡的，通过将 S 中的字符串拼接后插入 $k - \sum_{s_i \in S} |s_i|$ 数量的任意字符，即可得到 SCS 问题的一个合法解。

$k < \sum_{i=1}^{n} |s_i|$。易见 SCS (S, k) 有解当且仅当存在 $m \in \{0, 1, 2, \ldots, p\}$ 满足 (S, k, p) 是 LSCS 问题的解。此时 $k = O(\sum_{i=1}^{n} |s_i|)$，即 k 是关于输入大小的多项式，因此 m 可以在多项式时间内被枚举，即在多项式时间内将 SCS 问题在多项式时间内归约到 LSCS。

由定理 6.2 可知，LSCS 是 NP-完全的。 □

基于 LSCS 的 NP-完全性，我们实现了定理 6.1 结论的扩展，即证明在 P 满足更强限制条件下 VSC 问题的 NP-完全性：

引理 6.1. *在并发程序 P 仅包含一个共享变量 ($|X_s| = 1$，此时事件集合 E 中所有事件 $e.x \in X_s$ 都相同) 的前提下，VSC 问题仍是 NP-完全的。*

证明. 给定任一 LSCS 问题的实例 $S = \{s_1, \ldots, s_n\}$ 和整数 k, p，假设并发程序执行只访问一个共享变量 x (即 $X_s = \{x\}$)，其初值 $x = -1$，现按如下方式构造并发程序执行的事件集合 $E = W_0 \cup W_1 \cup W_2 \cup R_1 \cup \ldots R_n$ 及程序顺序 $\prec_{po}$ (构造的 $\langle E, \prec_{po} \rangle$ 如图 6.4 所示)：

1. W_0 包含 $k - p$ 个写事件 $w_i.rw = w$，每个写事件都发生在一个新线程 ($w_i.t$ 为一个新线程，这个线程仅执行 w_i 操作) 上，且 $w_i.x = x$, $w_i.v = 0$。
2. W_1 包含 p 个写事件 w_i，每个写事件都发生在一个新线程上，且 $w_i.x = x$, $w_i.v = 1$。
3. W_2 包含 k 个写事件 w_i，每个写事件都发生在一个新线程上，且 $w_i.x = x$, $w_i.v = 2$。

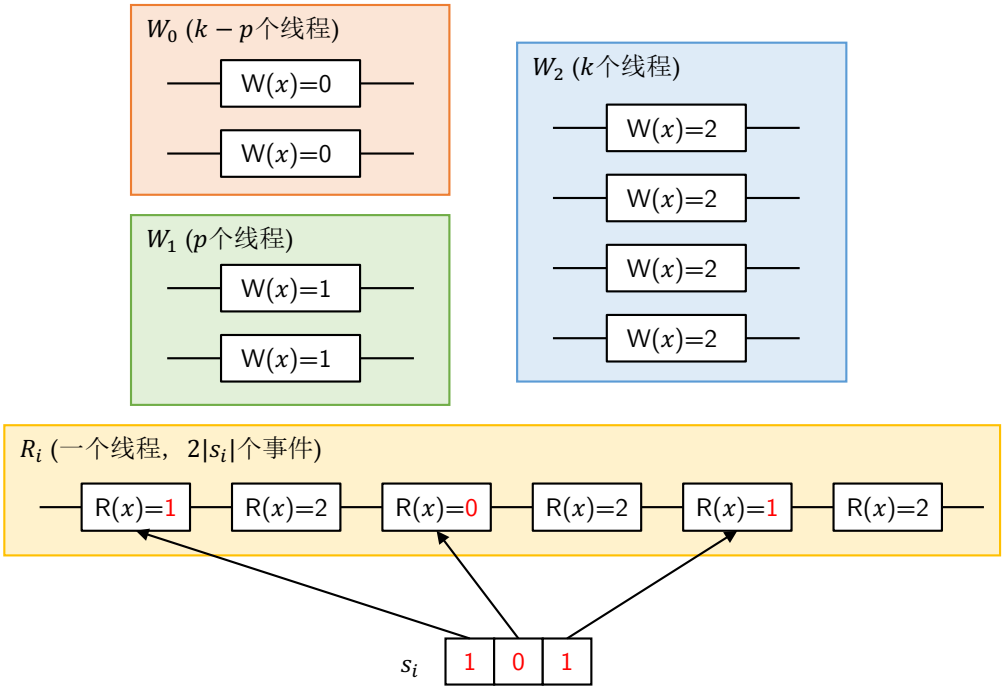

图 6.4 将 LSCS 实例归约到 VSC 所构造的 $\langle E, \prec_{po}\rangle$。

4. 对于每一字符串 $s_i \in S$，设其长度 $|s_i| = m$，构造发生在同一线程 t_i 上的读事件的集合 $R_i = \{e_1, e_2, \ldots, e_{2m}\}$, $e_1 \prec_{po} e_2 \prec_{po} \ldots \prec_{po} e_{2m}$, $\forall e \in R_i.(e.t = t_i) \wedge (e.x = x)$。对于 $1 \leq j \leq m$，设字符串 s_i 在 j 位置的字符 $s_i(j) \in \{0, 1\}$，构造 $e_{2j-1}.v = s_i(j)$, $e_{2j}.v = 3$。直观地说，我们创建了一个线程 t_i，其按顺序从 x 中读出字符串 s_i 的所有字符，并且每读出一个字符后都需要读到 2 进行确认。

接下来我们证明 LSCS 问题实例 $\langle S, k, m\rangle$ 有解当且仅当 $\langle E, \prec_{po}\rangle$ 存在满足顺序一致性的调度。由 LSCS 问题的 NP-完全性可证 VSC 的 NP-完全性。

(必要性) LSCS 问题实例 $\langle S, k, p\rangle$ 有解 $\Rightarrow \langle E, \prec_{po}\rangle$ 存在顺序一致性的调度。

直观地说，若存在长度为 k 的、包含 m 个 1 的字符串 s 满足对于任意 $s_i \in S$, s_i 是 s 的子序列，我们就按照 s 中所有字符出现的顺序执行写操作，并在每次写操作之后执行一个写入 2 的操作。对于任意 $s_i \in S$，由于 s_i 是 s 的子串，因而总可以找到写操作的一个子序列，使得 s_i 中的字符被按顺序写入，得到满足顺序一致性的事件排序。

形式地说，我们首先将 E 中所有写事件 $W_0 \cup W_1 \cup W_2$ 在 $<_w$ 中排序为 $e_1^w <_w e_2^w <_w \ldots <_w e_{2k}^w$，使得对于任意 $1 \leq i \leq k$，$e_{2i-1}^w.v = s(i)$, $e_{2i}^w.v = 2$。由于 W_1 中恰好有 $k-p$ 个写入 0 的事件、W_2 中有 p 个写入 1 的事件，与 s 中 01 的数量保持一致，又由于 W_3 中有 k 个写入 2 的事件，这样的排序总是存在。

对于发生在线程 t_i 上的读事件 $R_i = \{e_1, e_2, \ldots, e_{2m}\}$，由于 s_i 是 s 的子串，故存在序列 $l_1, l_2, \ldots, l_m$，满足对于 $1 \le j \le m$, $s_i(j) = s(l_j)$。按照如下方式确定 R_i 中事件的全局顺序：对于 $1 \le j \le m$，令 $e^w_{2l_j-1} <_r e_{2j-1} <_r e^w_{2l_j}$, $e^w_{2l_j} <_r e_{2j} <_r e^w_{2l_j+1}$，即为每一个读事件确定了其对应的写事件 (所有写事件在 $<_w$ 中排序)，又因为 $s_i(j) = s(l_j)$，由 $tr(<_r \cup <_w \cup \prec_{po})$ 确定的偏序关系的任意线性排序均满足顺序一致性。

(充分性) $\langle E, \prec_{po}\rangle$ 存在顺序一致性的调度 $\Rightarrow$ LSCS 问题实例 $\langle S, k, p\rangle$ 有解。

对于满足顺序一致性的调度 $\prec_{tot}$，我们按 $\prec_{tot}$ 顺序将 $W_0 \cup W_1$ 中的事件排序。根据每一事件 e 写入的数值 $e.v$，排序后的事件确定了一个长度为 k、恰好有 p 个 1 和 $k-p$ 个 0 的字符串 s。现证明对于任意 $s_i \in S$, s_i 均是 s 的子序列。

对于 $s_i \in S$, 其对应的读事件集合 $R_i = \{e_1, e_2, \ldots, e_{2m}\}$ 在 $\prec_{tot}$ 中被排序。根据顺序一致性的定义，对于任意 $e_j \in R_i$，都存在唯一的读事件 e'_j 满足 $e'_j.v = e_j$，且不存在写事件 e 满足 $e'_j \prec_{tot} e \prec_{tot} e_j$。

对于任意的 $1 \le j < 2m$，都有 $e_j.v \ne e_{j+1}.v$ (我们的构造保证了 $e.v \in \{0,1\}$ 和 $e.v = 2$ 的事件交替出现)，因此 $e'_j.v \ne e'_{j+1}.v$，即 $e'_j \ne e'_{j+1}$，故 $e'_j \prec_{tot} e'_{j+1}$。用数学归纳法即可证明 $W_0 \cup W_1 \cup W_2$ 存在一个子集，按照 $\prec_{tot}$ 的顺序排序后，依次写入 R_i 中所有读事件的数值，满足 s_i 是 s 的子序列。 □

6.2.2 NFL 猜想的特殊情形及其证明

尽管我们尚无手段证明 NFL 猜想，但引理 6.1 的框架帮助我们证明 NFL 猜想的一个特殊情形：要求所有对共享变量 x 都进行同样的修改，并且对变量的修改能写成一个只读的前缀和只写的后缀。这一特殊情形可描述为如下定理：

定理 6.3. *假设对并发程序 P 的修改 $\langle X^+_s, X^+_t, \mathcal{M}_L, \mathcal{M}_R\rangle$ 额外满足：*

1. *对于所有访问共享变量 x 的读指令 $i \in IR_x$ 或写指令 $i \in IW_x$，必须作出相同的修改 (插入相同的指令序列)，即*

$$\forall i_1, i_2 \in I.\ \big((\{i_1, i_2\} \subseteq IR_x) \vee (\{i_1, i_2\} \subseteq IW_x)\big) \\ \rightarrow \big((\mathcal{M}_L(i_1) = \mathcal{M}_L(i_2)) \wedge (\mathcal{M}_R(i_1) = \mathcal{M}_R(i_2))\big)。$$

2. *$\mathcal{M}_L$ 和 $\mathcal{M}_R$ 中不包含跳转指令，且首先执行一个只读前缀指令序列，然后执行一个只写后缀指令序列。即 $\mathcal{M}_L$ 能写成 $\mathcal{M}_L = \mathcal{M}_{L_r} :: \mathcal{M}_{L_w}$ 的形式，其中 $\mathcal{M}_{L_r}$ 中访问共享内存的指令只能是读指令、$\mathcal{M}_{L_w}$ 中访问共享内存的指令只能是写指令。同理 $\mathcal{M}_R$ 能写成 $\mathcal{M}_R = \mathcal{M}_{R_r} :: \mathcal{M}_{R_w}$ 为只读前缀和只写后缀的拼接。*

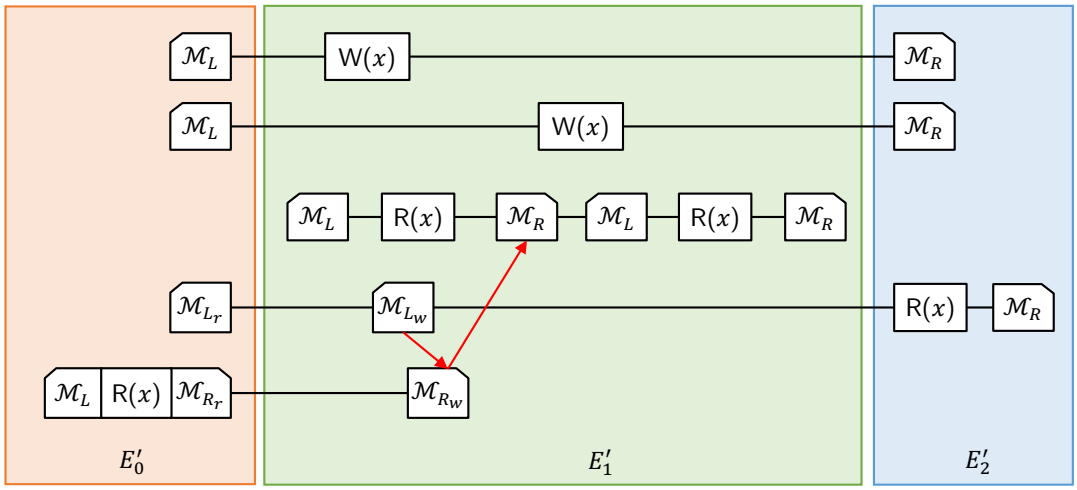

图 6.5 实现对 $\mathcal{M}_L, \mathcal{M}_R$ “信息屏蔽”的调度。

在此情形下，NFL 猜想成立，即存在程序 P，使得对于任何满足上述条件的修改，运行修改后程序 P' (得到事件集合 E' 及程序顺序 $\prec'_{po}$)，推导 $\langle E, \prec_{po}\rangle$ 满足顺序一致性的 $\prec_{tot}$ 时间复杂性是 NP-完全的。

证明. 我们使用引理 6.1 中的框架构造这一情形的 NP-完全性证明。给定 LSCS 的任意实例 $\langle S, k, p\rangle$，我们按照类似的方式构造并发程序 P，其线程集合

$$T = T_0 \cup T_1 \cup T_{w_0} \cup T_{w_1} \cup T_{w_2} \cup \{t_{r_1}, t_{r_2}, \ldots, t_{r_n}\},$$

共享变量集合 $X_s = \{x\}$，其中：

1. T_{w_0} 包含 $k-p$ 个线程，每个线程执行一条对 x 写 0 的指令；T_{w_1} 包含 p 个线程，每个线程执行一条对 x 写 1 的指令；T_{w_2} 包含 k 个线程，每个线程执行一条对 x 写 2 的指令。这些线程用于生成引理 6.1 中的写事件。
2. t_{r_i} 包含 $2|s_i|$ 条对 x 的读指令。这些线程用于生成引理 6.1 中对字符串 s_i 中的字符进行顺序读出所对应的事件。
3. T_0, T_1 各包含 $2\sum_{s_i \in S} |s_i|$ 个线程，每个线程只包含一条对 x 的读指令，用于辅助证明本定理。

易见，若 $\langle S, k, p\rangle$ 存在长度为 k 且恰好有 p 个 1 的超序列 s，则存在一次执行，使得 t_{r_i} 产生的第 $2j-1$ 个事件的读值恰好为 $s_i(j)$，且第 $2j$ 个事件的读值为 2。引理 6.1 证明，满足顺序一致性的事件排序恰好一一对应于 LSCS 问题的解，因此在不对 P 做任何修改的前提下，获取满足顺序一致性的访存依赖是 NP-完全问题。

我们现在考虑对 P 作出修改后的程序 P' 的执行。对于一次特定的执行，由于 P' 中引入了额外的共享内存访问操作，根据这些访存操作之间的依赖关系

(若一对读写事件读入/写入了全局唯一的数值，则可以在多项式时间内确定它们发生的先后关系) 可能在多项式时间内推导出 E 中事件满足顺序一致性的全序。因此，我们证明存在 P' 的一次执行，其生成的事件为 E'，已知 E' 发生的全序 $\prec'_{tot}$，该全序保证不能推导出任何 E 中事件的任何信息。

为此，我们构造如图 6.5 所示的调度。首先，假设 E 中的事件存在满足顺序一致性的 $\prec_{tot}$，且 t_{r_i} 产生的第 $2j-1$ 个事件读的值恰好为 $s_i(j)$，这保证了求解 E 中顺序一致性的调度能对应到 LSCS 相应实例的解。

由于 P' 对 P 产生了修改，E 中的每一事件 e 所对应的指令 i 都在之前插入了指令序列 $\mathcal{M}_L = \mathcal{M}_{L_r} :: \mathcal{M}_{L_w}$、在之后插入了指令序列 $\mathcal{M}_R = \mathcal{M}_{R_r} :: \mathcal{M}_{R_w}$。令 $EL(e) = EL_r(e) \cup EL_w(e)$ 表示 e 之前插入的指令序列所生成的事件、$ER(e) = ER_r(e) \cup ER_w(e)$ 表示 e 之后插入的指令序列所生成的事件，其中 EL_r/ER_r 代表读前缀产生的事件，EL_w/ER_w 代表写后缀产生的事件。考虑 E' 对应的 P 的执行事件记录 $\langle E, \prec_{po}\rangle$，令 E_w 为其中写事件的集合；E_r 为其中 $\{t_{r_i}\}$ 线程产生读事件的集合，E_0 为其中 T_0 中线程产生读事件的集合，E_1 为其中 T_1 中线程产生读事件的集合。我们将 E' 分为三组：

1. E'_0 为以下事件集合的并集：(1) E_w 中事件 e 的 $EL(e)$; (2) E_0 中事件 e 的 $EL_r(e)$; (3) E_1 中事件 e 的 $EL(e) \cup \{e\} \cup ER_r(e)$。
2. E'_1 为以下事件集合的并集：(1) E_w; (2) E_r 中事件 e 的 $EL(e) \cup \{e\} \cup ER(e)$; (3) E_0 中事件 e 的 $EL_w(e)$; (4) E_1 中事件 e 的 $ER_w(e)$；
3. E'_2 为以下事件集合的并集：(1) E_w 中事件 e 的 $ER(e)$; (2) E_0 中事件 e 的 $\{e\} \cup ER(e)$。

在我们构造的调度中，令 $E'_0 \prec'_{tot} E'_1 \prec'_{tot} E'_2$ (集合上的偏序关系 $P \prec_{tot} Q$ 定义为 $\forall p \in P, q \in Q. p \prec_{tot} q$)。此外，对于一个读事件 $e \in E_r$，我们各从 T_0 和 T_1 中选取一个对 x 的读事件 e_0, e_1 (对每个 e 选取不同的 e_0 和 e_1)，令 $EL_w(e_0) \prec_{tot} ER_w(e_1) \prec_{tot} EL(e)$，且这些事件在 $\prec_{tot}$ 中连续发生 (如图 6.5 中箭头所示)。由于每个 e_0, e_1 只被选中一次，易见这样的顺序是一定可以被满足的。

然后我们考虑 $EL_r(e)$ 或 $ER_r(e)$ 中的一个读事件 e_r。由于 $e_r \in E'_2$, $\forall e \in E'_1$, $e \prec'_{tot} e_r$ 成立。此外，由于读事件 e_r 是修改程序产生的，因而 $e_r.x \in X^+$。由于所有向 x 的读指插入的指令序列均相同，我们对 e_r 进行分类讨论：

1. 若 $e_r.x \in \{e.x \mid e \in EL_w(e_r) \cup ER_w(e_r)\}$，即 $e_r.x$ 会被读操作的修改改写，由于 $EL_w(e_0) \prec_{tot} ER_w(e_1) \prec_{tot} EL(e)$ 连续发生，$e_r.v$ 可在 E'_0 结束后确定。
2. 若 $e_r.x \in \{e.x \mid e \in EL_w(e_w)\}$，$e_w$ 为任一对 x 的读事件。由于 $EL(e_w) \subseteq E'_0$，$e_r.v$ 也可在 E'_0 结束后确定。

3. 否则，不存在任何 $e \prec_{tot} e_r$ 满足 $e.rw = w \wedge e.x = e_r.x$，此时对 $e_r.x$ 的读将读到其初始值。

因此在 $\prec'_{tot}$ 中，对于任意 $e \in E_r$，$EL(e)$ 和 $ER(e)$ 中共享内存读事件的值都在 E'_0 中确定。换言之，对调度 $\prec'_{tot}$ 而言，对 P 的修改等价于插入了一些在 P 运行前确定的常数。因此，运行 P' 所得到的记录 $\langle E', \prec'_{po}\rangle$ 不包含任何关于 E 中事件的信息。根据 VSC 问题的 NP-完全性，求解 E 满足顺序一致性的访存依赖是 NP-完全问题。 □

据我们所知，这一定理是迄今为止首个“允许对程序进行修改”这一模型下，与共享内存访问顺序相关的复杂性证明。我们希望以此为起点，对这一问题进行更深入的研究。

6.3 复杂性结果的启示

NFL 猜想无论是被证明还是被证否，都有其重要意义。若猜想被证明，则为了获取并发程序的执行轨迹，要么需要在运行时加入额外的同步，要么需要付出求解 NP-完全的代价。目前我们已知其两种特殊情形：VSC 问题是 NP-完全问题；本章证明的具有读前缀-写后缀特性的 wait-free 程序修改，获取访存依赖的事件复杂性也是 NP-完全的。这两种特殊情形已经表明，“免费的午餐”是很难获得的，至少在对并发程序不做修改或仅做简单修改的前提下，获取访存依赖是非常具有挑战性的问题。

另一方面，若猜想被证否，就意味着给定一个程序 P，存在一个对其的 wait-free 修改，使得我们能在多项式时间内根据修改后程序的执行推导出 P 的执行轨迹。这无疑是访存依赖获取技术的一大技术突破，能够填补二十多年来研究工作的空白 (目前已知最好的结论仅能保证对所有读操作作出 wait-free 的修改 [144])。我们认为，NFL 猜想仍存在被证否的可能：分布式理论表明，可以在 wait-free 的情况下获取原子性快照 (atomic snapshot) [10]，即仅使用读/写指令能够实现获得全局一致状态的共享对象。虽然这并不意味着我们能立即在多项式时间内获得共享内存访问的顺序，但类似的技术能否用于实现 wait-free 修改下多项式时间的访存依赖获取，仍然是一个开放性问题。此外，即便满足顺序一致性的执行轨迹能使用原子性快照技术在多项式时间内获取，由于原子性快照的时间复杂性具有 $\Omega(|T|)$ 的下界，如何将这类算法应用到实践中，也是值得进一步研究的问题。

NFL 猜想与分布式理论之间的关系也是一个值得探讨的问题。分布式理论中的一个重要研究问题是能否使用某些并发对象实现其他并发对象 [64]。在此

表 6.1 同步操作的一致性范数层次结构。

一致性范数	可实现的并发对象
1	满足原子性的变量访问
2	test-and-set、并发队列、并发栈
...	...
$2m-2$	m 寄存器原子赋值 ($m>1$)
...	...
∞	compare-and-set、链接加载/条件存储、有界互斥

过程中，提出了一致性范数 (consensus number) 的概念，并且证明了在一致性范数的层次结构中，一致性范数小的并发对象不能在 wait-free 的前提下实现一致性范数大的并发对象 [62]。一致性范数的层次结构与常见的并发对象在表 6.1 中描述。后续的研究进一步表明，原子性的变量访问无法在 wait-free 的前提下实现互斥 [15]。这一“不可能性”结论与 NFL 猜想中“复杂性”的相关的结论具有一定的相似之处：分布式理论表明使用 wait-free 的变量读/写无法实现互斥；NFL 猜想表明使用 wait-free 的变量读/写无法在多项式时间内确定并发程序访问共享内存的全局顺序。这两个形式上相似的结论是否有更深层次的内在联系？是否有可能使用分布式理论中的相关技术对 NFL 猜想进行研究？

本章的理论探讨还揭示了对访存依赖获取既“困难”又“容易”的有趣性质：一方面，NFL 猜想及已经证明的部分表明，总存在一个“复杂”的程序，仅通过线程局部的记录获取访存依赖这一问题是有本质的复杂性困难的。另一方面，尽管导致 NP-完全时间复杂性的最坏情况总是存在，但我们提出的局部性理论又表明，现实世界中并发程序的共享内存访问通常遵循特定的模式，即共享内存访问具有线程、空间-线程和同步局部性。对于具有局部性的程序，获取其访存依赖是相对简单的。例如若能证明并发程序不存在数据竞争，我们只需对同步事件发生的顺序予以记录即可，此时我们对每个内存访问指令只需添加 $O(1)$ 的读/写或线程局部计算操作；即使可能存在数据竞争，在运行时高效地检测局部性也能提高访存依赖获取的效率 [28]。

我们已经看到一些利用访存依赖获取亦难亦易这一特性的研究工作：这些技术虽然在理论上需要求解 NP-完全问题，但使用启发式优化和约束求解器却可以使这些研究工作能有效地得到实际程序的访存依赖 [74, 87]。

当我们再次考虑 NFL 猜想的逻辑结构：

$$\exists P.\,\forall M.\,((\forall S.\,\text{wait-free}(P,M,S)) \to \exists \tau.\,\text{NPC}(\tau)),$$

其中 P 是程序、M 是对 P 的修改、S 是对修改后程序的调度、τ 是修改后程序的执行轨迹。对这一逻辑结构进行拆解，就能发现这一猜想不仅蕴含了引入同步和多项式时间获取访存依赖的矛盾，更揭示了一个更广阔的研究空间：

1. NFL 猜想中的 $\exists P$ 表明了它和局部性理论是不矛盾的。对于现实中的并发程序 P，局部性理论描述了其具有的一些特征，进而促生了一批基于局部性的访存依赖获取技术。因此扩展局部性理论，更深入地研究现实中并发程序共享内存访问具有的特征，并利用这些特征实现访存依赖的高效获取(甚至有可能通过 wait-free 的插装就能得到一致的访存依赖)，是未来研究工作的重点。
2. 对于一个并非最坏情况的 P，很可能 $\exists M$ 满足 $\forall S.\,\text{wait-free}(P, M, S)$，且根据 M 修改程序后总能在多项式时间内得到满足顺序一致性的访存依赖。这启发我们应该对每个 P 做定制化的修改。现有技术通常对所有的 P 都采取完全相同的修改，只有少量研究 [85] 对 P 进行了定制化修改的初步尝试。实际上，对于程序 P，很可能只要作出极少的修改，就足以帮助我们在多项式时间内获取访存依赖。例如我们在尝试扩展文献 [57] 对 VSC 问题 NP-完全性的证明，并以此证明 NFL 猜想的特殊情形时发现，只要对原程序增加极少量的读/写操作，就能在多项式时间内恢复满足顺序一致性的执行 (因此我们放弃了对原证明的扩展，并提出了全新的证明)。对于任意给定的并发程序，能否根据其访存的特点，用较少的修改获得访存依赖，也是未来研究的方向。

因此，无论最终是否真的存在“免费的午餐”，本章从访存依赖获取复杂性的理论探讨出发，最终联系实际并发程序所具有的局部性理论，从一个全新的视角展示了现有工作的空白，帮助我们更深入地理解了访存依赖获取这一问题，对未来可能的研究方向提供了启示。

第七章　总结与展望

访存依赖是实现并发程序动态分析 (及与之相关的并发程序质量保障技术) 的基石。然而目前访存依赖的相关研究仍存在两点局限：(1) 缺乏对访存依赖获取问题的理论认识和对“访存依赖”这一事物的量化研究；(2) 缺乏统一的技术框架以研究现有技术的优势与不足。这两点局限制约了目前访存依赖获取技术的发展，进而影响了基于访存依赖的动态分析技术的发展，导致动态分析应用在真实并发程序中仍面临挑战。

本文针对现有研究的局限，提出了相应的理论框架、技术框架，并在此基础上对访存依赖的获取技术、基于访存依赖的动态分析进行了研究。本章对全文工作进行了总结，并对未来研究方向予以展望。

7.1 工作总结

针对现有研究工作缺乏理论认识、缺乏系统总结的两点局限，我们首先完成了以下框架性的工作：

1. 提出了“访存依赖获取”这一问题的理论框架。首先，理论框架对问题进行了全面的形式化建模，包括并发系统的定义、访存依赖的定义和性质。其次，为了指导访存依赖获取，我们提出了反映现实程序访问共享内存不均匀 (non-uniform) 特征的局部性理论，完成了首次对“局部性”的形式化定义和量化研究，在一组现实程序上证实了线程局部性、空间-线程局部性、同步局部性的存在。
2. 在理论框架的基础上，建立了以“四个评价指标、两类获取技术、两类应用”三要素为主线的技术框架，认识到访存依赖获取技术必须在即时性、准确性、高效性、简化性这些指标之间进行权衡，并在这一技术框架下完成了对横跨体系结构、计算机系统、程序设计语言和软件工程四个领域中访存依赖获取的两类技术 (在线追踪、离线合成) 和两类应用 (轨迹分析、并发控制) 的综述。

在理论框架和技术框架的共同指导下，对技术框架中已有工作的改进空间和现有技术的空白完成了高效访存依赖获取技术的研究工作：

1. 利用线程局部性实现的高效访存依赖获取技术 RWTrace。相比于朴素的互斥锁，RWTrace 在访存密集型的程序上能实现多达数百倍的性能提升；

2. 利用空间-线程局部性实现的高效访存依赖约减的二分组 (bisectional coordination) 协议。相比于高度优化的 RWTrace，在仅付出 0–54.7% 运行时开销的前提下，实现多达 97% 的访存依赖数量约减。

进而，本文还开展了基于访存依赖的动态分析技术研究：

1. 基于访存依赖的并发程序动态分析：基于缓存的执行重放技术 CARE 相比于使用互斥锁的技术，减少多达数十倍的运行时开销和数百倍的访存依赖数量；基于二分组协议的数据竞争检测和假共享检测技术，能够大幅减少检测调用的数量和元数据维护开销；
2. 基于依赖反转的软件测试：应用程序崩溃一致性检测技术 C^3 提出基于编辑距离的测试预言，实现了应用程序崩溃一致性的全自动检测；移动应用的并发测试技术 AATT 能同时探索移动应用的输入空间和调度空间，暴露其中的并发缺陷。两项测试技术在知名开源软件中检出前所未知的缺陷。

最后，我们发现至今没有研究工作能在不引入同步的前提下在多项式时间内获取满足顺序一致性访存依赖。基于这一事实及一些已有的复杂性结论，我们提出了“没有免费午餐”(No-Free-Lunch, NFL) 猜想：在只对程序进行 wait-free 修改的前提下，获得满足顺序一致性访存依赖的时间复杂性是 NP-完全的。我们给出了判定执行轨迹顺序一致性 (VSC) 问题在一个更受限情况下的证明，并用此技术证明了 NFL 猜想的一个特殊情形：在仅能对同一共享内存地址作出同样 wait-free 的修改并且修改能写成一个只读前缀和只写后缀的情况下，无论作出怎样的修改，推导出满足顺序一致性访存依赖的时间复杂性均是 NP-完全的。这一结果可以视作对现有工作的理论总结，并且其分析过程启发了未来可能的研究方向。

7.2 研究展望

总结本文开展的针对访存依赖理论框架、技术框架、访存依赖获取技术、基于访存依赖的动态分析和获取访存依赖复杂性的研究，我们发现仍有很多未来可能的研究机遇，现总结如下：

1. *已有技术向其他领域的渗透*。由于访存依赖获取问题是并发程序动态分析的基础，其研究分布于多个研究领域。针对某一应用场景所提出的访存依赖获取技术能否扩展到其他领域的其他应用，是一个值得研究的问题。
   我们已经看到一些领域之间融合的例子，例如路径合成技术 [74] 基于硬件数值合成技术中的约束可满足性问题 [87]、使用硬件事务内存能实现数据竞争检测的加速 [139]。融合计算机硬件和计算机软件领域的技术是这类

研究的关注点之一。现有的一些使用修改硬件缓存一致性协议的技术 (如 BulkSC [31]) 的纯软件实现是一大挑战；反之，软件中使用的技术如何促进硬件相关技术的发展，也是未来可能的研究方向。例如在硬件层实现本文提出的动态锁指派技术 [80] 并将若干连续的缓存线进行绑定作为原子处理。这一融合不仅能够降低访存依赖获取的代价，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。

2. *基于并发程序自身特性的访存依赖获取*。在 NFL 猜想及其特殊情形证明中，我们已经发现访存依赖获取是既“困难”又“容易”的。NFL 猜想与并发程序共享内存访问具有的局部性并不矛盾，因此利用并发程序自身的特性，通过巧妙地记录额外信息以实现访存依赖的高效获取是未来可能的研究方向。

   目前，有一些研究工作在这一方向上进行了初步的探索：Chen 等人利用了时钟信息减少依赖的数量 [34]；Liu 等人使用了程序中的同步操作来降低约束求解的难度 [90]。然而，目前对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还未有系统性认识。对这一问题的深入理解势必将促进更高效的访存依赖获取技术。

3. *建模并发程序的执行*。本文提出的局部性理论只是对并发程序建模的初步尝试，一些技术使用的启发式算法的有效性 (如 CLAP [74] 进行约束求解的复杂性) 尚不能被局部性理论完美解释。如能更准确地建模真实世界中的并发程序执行及其特征，则能更好地理解这类技术的优势与不足。基于建立的模型，已有技术的有效性、优缺点均可直接在理论模型上予以分析；提出的新技术在进行实证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)；抑或在现实并发程序的模型上证否 NFL 猜想，从而最终说明获取访存依赖对于现实并发程序来说确实是“容易”的。

参考文献

[1] ASM toolkit for bytecode manipulation. http://asm.ow2.org/.

[2] JVM tool interface. http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html.

[3] The LLVM compiler infrastructure. http://llvm.org/.

[4] Processes and threads. http://developer.android.com/guide/components/processes-and-threads.html.

[5] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[6] The Single UNIX Specification. http://www.unix.org/version3, 2002.

[7] Java Specification Request (JSR) 133. Java Memory Model and Thread Specification. http://jcp.org/jsr/detail/133.jsp, 2004.

[8] Linux Kernel block driver docs, 2005.

[9] The Scala Programming Language. http://www.scala-lang.org, 2016.

[10] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of The ACM*, 40(4):873–890, 1993.

[11] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[12] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 193–206, 2009.

[13] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the International Symposium on High Performance Computer Architecture*, HPCA, pages 59–69, 2005.

[14] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[15] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 487–498, 2011.

[16] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 641–660, 2013.

[17] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, PADD, pages 194–206, 1991.

[18] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, second edition, 2006.

[19] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 53–64, 2010.

[20] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Conference on Operating Systems Design and Implementation*, OSDI, pages 177–191, 2010.

[21] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 81–96, 2009.

[22] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE, pages 154–163, 2006.

[23] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for Android applications. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 332–348, 2015.

[24] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[25] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, software-only region conflict exceptions. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 241–259, 2015.

[26] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of The ACM*, 13(7):422–426, 1970.

[27] Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. Efficient deterministic replay of multithreaded executions in a managed language virtual machine. In *Proceedings of the Principles and Practices of Programming on the Java Platform*, PPPJ, pages 90–101, 2015.

[28] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. OCTET: Capturing and controlling cross-thread dependences efficiently. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 693–712, 2013.

[29] Yan Cai and Lingwei Cao. Effective and precise dynamic detection of hidden races for Java programs. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 450–461, 2015.

[30] Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. Drinking from both glasses: Combining pessimistic and optimistic tracking of cross-thread dependences. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 20:1–20:13, 2016.

[31] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 278–289, 2007.

[32] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 227–238, 2006.

[33] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 207–218, 2013.

[34] Yunji Chen, Weiwu Hu, Tianshi Chen, and Ruiyang Wu. LReplay: A pending period based deterministic replay scheme. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 187–197, 2010.

[35] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic replay: A survey. *ACM Computing Surveys*, 48:17:1–17:47, 2015.

[36] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 304–315, 2008.

[37] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 1–19, 1999.

[38] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[39] North American Electric Reliability Council. Technical Analysis of the August 14, 2003, Blackout: What Happened, Why, and What Did We Learn?, 2013.

[40] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical

runtime for deterministic, stable, and reliable threads. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 388–405, 2013.

[41] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 337–351, 2011.

[42] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Conference on Operating Systems Design and Implementation*, OSDI, pages 207–221, 2010.

[43] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 67–78, 2010.

[44] Peter J. Denning. The locality principle. *Communications of The ACM*, 48(7):19–24, 2005.

[45] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 85–96, 2009.

[46] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: A relaxed consistency deterministic computer. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 67–78, 2011.

[47] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO, pages 21–33, 2007.

[48] Dave Dice and Nir Shavit. TLRW: Return of the read-write lock. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 284–293, 2010.

[49] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 1–10, 1990.

[50] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34:4:1–4:48, 2012.

[51] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Conference on Operating Systems Design and Implementation*, OSDI, pages 211–224, 2002.

[52] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE, pages 121–130, 2008.

[53] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 245–255, 2007.

[54] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the Workshop on Dynamic Analysis*, pages 24–27, 2003.

[55] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 121–133, 2009.

[56] Cormac Flanagan and Stephen N. Freund. Adversarial memory for detecting destructive races. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 244–254, 2010.

[57] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26:1208–1244, 1997.

[58] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the International International Computer-Aided Verification*, CAV, pages 176–185, 1990.

[59] Michelle L. Goodstein, Evangelos Vlachos, Shimin Chen, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 257–270, 2010.

[60] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27:8–29, 2007.

[61] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 14–25, 2006.

[62] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Language and Systems*, 13(1):124–149, 1991.

[63] Maurice Herlihy and J. Eliot Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 289–300, 1993.

[64] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Elsvevier, 2012.

[65] Peter H. Hofstee. Future microprocessors and off-chip SOP interconnect. *IEEE Transactions on Advanced Packaging*, 27(2):301–303, 2004.

[66] Nima Honarmand and Josep Torrellas. RelaxReplay: Record and replay for relaxed-consistency multiprocessors. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 223–238, 2014.

[67] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 265–276, 2008.

[68] Chun-Hung Hsiao Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 326–336, 2014.

[69] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proceedings of the Symposium on the Foundations of Software Engineering*, FSE, pages 385–386, 2010.

[70] Jeff Huang, Qingzhou Luo, and Grigore Rosu. GPredict: Generic predictive concurrency analysis. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 847–857, 2015.

[71] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 337–348, 2014.

[72] Jeff Huang and Charles Zhang. An efficient static trace simplification technique for debugging concurrent programs. In *Proceedings of the International Static Analysis Symposium*, SAS, pages 163–179, 2011.

[73] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 144–154, 2011.

[74] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: Recording local executions to reproduce concurrency failures. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 141–152, 2013.

[75] SPARC International Inc. The SPARC Architecture Manual V9, 1994.

[76] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the Symposium on the Foundations of Software Engineering*, FSE, pages 57–66, 2010.

[77] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. Crash consistency validation made easy. In *Proceedings of the Symposium on the Foundations of Software Engineering*, FSE, pages 133–143, 2016.

[78] Yanyan Jiang, Tianxiao Gu, Chang Xu, Xiaoxing Ma, and Jian Lu. CARE: Cache guided deterministic replay for concurrent Java programs. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 457–467, 2014.

[79] Yanyan Jiang, Du Li, Chang Xu, Xiaoxing Ma, and Jian Lu. Optimistic shared memory dependence tracing. In *Proceedings of the International Conference on Automated Software Enginnering*, ASE, pages 524–534, 2015.

[80] Yanyan Jiang, Chang Xu, Du Li, Xiaoxing Ma, and Jian Lu. Online shared memory dependence reduction via bisectional coordination. In *Proceedings of the Symposium on the Foundations of Software Engineering*, FSE, pages 822–832, 2016.

[81] Yanyan Jiang, Chang Xu, and Xiaoxing Ma. DPAC: An infrastructure for dynamic program analysis of concurrency Java programs. In *Proceedings of the International Conference on Middleware*, Middleware Doctoral Symposium, pages 2:1–2:6, 2013.

[82] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of The ACM*, 21:558–565, 1978.

[83] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[84] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36:471–482, 1987.

[85] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 463–474, 2012.

[86] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, and Zijiang Yang. Offline symbolic analysis to infer Total Store Order. In *Proceedings of the International Symposium on High Performance Computer Architecture*, HPCA, pages 357–358, 2011.

[87] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the International Symposium on Microarchitecture*, MICRO, pages 564–575, 2009.

[88] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[89] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 138–147, 1996.

[90] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. Light: Replay via tightly bounded recording. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 55–64, 2015.

[91] Tongping Liu and Emery D. Berger. SHERIFF: Precise detection and automatic mitigation of false sharing. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 3–18, 2011.

[92] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 327–336, 2011.

[93] Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-sized nonblocking hash tables. In *Proceedings of the Symposium on Principles of distributed computing*, PODC, pages 242–251, 2014.

[94] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 329–339, 2008.

[95] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 37–48, 2006.

[96] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 210–221, 2010.

[97] Wayne A. Madison and Alan P. Batson. Characteristics of program localities. *Communications of The ACM*, 19(5):285–294, 1976.

[98] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race dectection for Android applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 316–325, 2014.

[99] Zhanshuai Meng, Yanyan Jiang, and Chang Xu. Facilitating reusable and scalable automated testing and analysis for Android apps. In *Proceedings of the Asia-Pacific Symposium on Internetware*, Internetware, pages 166–175, 2015.

[100] Sang L. Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 235–244, 1991.

[101] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 289–300, 2008.

[102] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the Conference on Operating Systems Design and Implementation*, OSDI, pages 267–280, 2008.

[103] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 229–240, 2006.

[104] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[105] Robert H. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, PADD, pages 1–11, 1993.

[106] Robert H. Netzer and Barton P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the International Conference on Parallel Processing*, ICPP, pages 93–97, 1990.

[107] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 348–354, 1984.

[108] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 25–36, 2009.

[109] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 177–192, 2009.

[110] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO, pages 2–11, 2010.

[111] Thanumalayan S. Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the Conference on Operating Systems Design and Implementation*, OSDI, pages 433–448, 2014.

[112] Michiel Ronsse and Koen De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17:133–152, 1999.

[113] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 263–272, 2006.

[114] Kari-Jouko Räihä and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is NP-Complete. *Theoretical Computer Science*, 16:187–198, 1981.

[115] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15:391–411, 1997.

[116] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI, pages 11–21, 2008.

[117] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. Hybrid static-dynamic analysis for statically bounded region serializability. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 561–575, 2015.

[118] Traian F. Serbanuta, Feng Chen, and Grigore Rosu. Maximal causal models for sequentially consistent systems. In *Proceedings of the International Conference on Runtime Verification*, RV, pages 136–150, 2012.

[119] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of The ACM*, 53(7):89–97, 2010.

[120] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC, pages 204–213, 1995.

[121] Joao M. Silva, Jose Simao, and Luis Veiga. Ditto–Deterministic execution Replayability-as-a-Service for Java VM on multiprocessors. In *Proceedings of the International Middleware Conference*, Middleware, pages 405–424, 2013.

[122] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 387–400, 2012.

[123] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43:651–663, 1994.

[124] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 70–82, 2001.

[125] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 205–214, 2007.

[126] Jue Wang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. E-GreenDroid: Effective energy inefficiency analysis for Android applications. In *Proceedings of the Asia-Pacific Symposium on Internetware*, Internetware, pages 71–80, 2016.

[127] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection. In *Proceedings of the International Conference on Automated Software Enginnering*, ASE, pages 155–165, 2015.

[128] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the Annual International Symposium on Computer Architecture*, ISCA, pages 24–36, June 1995.

[129] Min Xu, Rastislav Bodik, and Mark D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture*, ISCA, pages 122–135, 2003.

[130] Min Xu, Mark D. Hill, and Rastislav Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *Proceedings of the*

Architectural Support for Programming Languages and Operating Systems, ASPLOS, pages 49–60, 2006.

[131] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the Conference on Operating Systems Design and Implementation*, OSDI, pages 131–146, 2006.

[132] Zheming Yang, Min Yang, Binyu Zang, Lvcai Xu, and Haibo Chen. ORDER: Object centRic DEterministic Replay for Java. In *Proceedings of the USENIX Annual Technical Conference*, ATC, pages 1–14, 2011.

[133] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel(R) Transactional Synchronization Extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis*, SC, pages 1–11, 2013.

[134] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 221–234, 2005.

[135] Xiang Yuan, Chenggang Wu, Zhenjiang Wang, Jianjun Li, Pen-Chung Yew, Jeff Huang, Xiaobing Feng, Yanyan Lan, Yunji Chen, and Yong Guan. ReCBuLC: Reproducing concurrency bugs using local clocks. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 824–834, 2015.

[136] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems*, EuroSys, pages 321–334, 2010.

[137] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. Low-overhead software transactional memory with progress guarantees and strong semantics. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 97–108, 2015.

[138] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 595–605, 2012.

[139] Tong Zhang, Dongyoon Lee, and Changhee Jung. TxRace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 159–173, 2016.

[140] Ying Zhang, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Ping Yu. ABC: Accelerated building of C/C++ projects. In *Proceedings of the Asia-Pacific Software Engineering Conference*, APSEC, pages 182–189, 2015.

[141] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE, pages 27–38, 2011.

[142] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI, pages 449–464, 2014.

[143] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of SSDS under power fault. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST, pages 271–284, 2013.

[144] Jinguo Zhou, Xiao Xiao, and Charles Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 892–902, 2012.

简历与科研成果

基本信息

蒋炎岩，男，汉族，1988 年 7 月出生，江苏省南京市人。

教育背景

2011 年 9 月 — 2017 年 12 月	南京大学计算机科学与技术系	博士研究生
2015 年 9 月 — 2016 年 3 月	美国俄亥俄州立大学计算机科学与工程系	访问学者
2007 年 9 月 — 2011 年 6 月	南京大学计算机科学与技术系	本科

学位论文相关的主要学术成果

1. **Yanyan Jiang**, Tianxiao Gu, Chang Xu, Xiaoxing Ma, and Jian Lu, CARE: Cache guided deterministic replay for concurrent Java programs. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE), 457–467, 2014. [CCF-A]
2. **Yanyan Jiang**, Du Li, Chang Xu, Xiaoxing Ma, and Jian Lu, Optimistic shared memory dependence tracing. In *Proceedings of the 30th International Conference on Automated Software Engineering* (ASE), 524–534, 2015. [CCF-A]
3. **Yanyan Jiang**, Chang Xu, Du Li, Xiaoxing Ma, and Jian Lu, Online shared memory dependence reduction via bisectional coordination. In *Proceedings of the 24th International Symposium on the Foundations of Software Engineering* (FSE), pages 822–832, 2016. [CCF-A]
4. **Yanyan Jiang**, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu, Crash consistency validation made easy. In *Proceedings of the 24th International Symposium on the Foundations of Software Engineering* (FSE), pages 133–143, 2016. [CCF-A]
5. **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, 28(4):747–763, 2017.

攻读博士学位期间完成的其他学术成果

6. Jue Wang, **Yanyan Jiang**, Chang Xu, Qiwei Li, Tianxiao Gu, Jun Ma, Xiaoxing Ma, and Jian Lu, AATT+: Effectively manifesting concurrency bugs in Android apps. *Science of Computer Programming* (SCP), to appear, 2017. [CCF-B]
7. Shengtao Yue, Weizan Feng, Jun Ma, **Yanyan Jiang**, Xianping Tao, Chang Xu, and Jian Lu, RepDroid: An automated tool for Android application repackaging detection. In *Proceedings of the 25th International Conference on Program Comprehension* (ICPC), 132–142, 2017. [CCF-B]
8. Tianxiao Gu, Xiaoxing Ma, Chang Xu, **Yanyan Jiang**, Chun Cao, and Jian Lu, Synthesizing object transformation for dynamic software updating. In *Proceedings of the 39th International Conference on Software Engineering*, (ICSE Poster Track), 2017. [CCF-A, Poster Track]
9. Qiwei Li, **Yanyan Jiang**, Tianxiao Gu, Chang Xu, Jun Ma, Xiaoxing Ma, and Jian Lu, Effectively manifesting concurrency bugs in Android apps. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference* (APSEC), 209–216, 2016. [CCF-C]
10. Xiangyu Wu, **Yanyan Jiang**, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu, Testing Android apps via guided gesture event generation. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference* (APSEC), 201–208, 2016. [CCF-C]
11. Jun Sui, Chang Xu, S.C. Cheung, Wang Xi, **Yanyan Jiang**, Chun Cao, Xiaoxing Ma, and Jian Lu, Hybrid CPU-GPU constraint checking: Towards efficient context consistency. *Information and Software Technology* (IST), 230–242, 2016. [CCF-B]
12. Hao Jin, **Yanyan Jiang**, Na Liu, Chang Xu, Xiaoxing Ma, and Jian Lu, Concolic metamorphic debugging. In *Proceedings of the 39th Computer Society International Conference on Computers, Software and Applications* (COMPSAC), 2015. [CCF-C]
13. Ying Zhang, **Yanyan Jiang**, Chang Xu, Xiaoxing Ma, and Ping Yu, ABC: Accelerated building of C/C++ projects. In *Proceedings of the 22st Asia-Pacific Software Engineering Conference* (APSEC), 182–189, 2015. [CCF-C]

14. Zhanshuai Meng, **Yanyan Jiang**, and Chang Xu, Facilitating reusable and scalable automated testing and analysis for Android apps, in *Proceedings of the 7th Asia-Pacific Symposium on Internetware* (Internetware), 2015.
15. Xiangyu Wu, Chang Xu, Ziling Lu, **Yanyan Jiang**, Chun Cao, Xiaoxing Ma, and Jian Lu, CoseDroid: Effective computation- and sensing-offloading for Android apps. In *Proceedings of the 39th Computer Society International Conference on Computers, Software and Applications* (COMPSAC), 2015. [CCF-C]
16. Xiujiang Li, **Yanyan Jiang**, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu, User guided automation for testing mobile apps. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference* (APSEC), 27–34, 2014. [CCF-C]
17. Jun Sui, Chang Xu, Wang Xi, **Yanyan Jiang**, Chun Cao, Xiaoxing Ma, and Jian Lu, GAIN: GPU-based constraint checking for context consistency. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference* (APSEC), 319–326, 2014. [CCF-C]
18. **Yanyan Jiang**, Chang Xu, and Xiaoxing Ma, DPAC: An infrastructure for dynamic program analysis of concurrency Java programs. In *Proceedings of the 2013 Middleware Doctoral Symposium*, 2:1–2:6, 2013. [CCF-B, Doctoral Symposium Track]
19. 许畅, 马晓星, 吕建, 李其玮, **蒋炎岩**. 一种安卓应用并发漏洞检测系统: 中国, 201610952304.7 (授权), 2016.
20. 许畅, 马晓星, 吕建, 武翔宇, **蒋炎岩**. 一种安卓应用的相关手势投放测试框架: 中国, 201610952301.3 (申请), 2016.
21. 马晓星, 许畅, 吕建, 李秀江, **蒋炎岩**. 一种基于用户执行踪迹重放的移动应用测试方法: 中国, 201410364808.8 (授权), 2014.

攻读博士学位期间参与的科研课题

1. 国家自然科学基金重大研究计划集成项目“可信软件理论、方法集成与综合试验平台”，No. 91318301；
2. 国家重点基础研究发展计划 (973 计划) 项目课题“持续演进的自适应网构软件模型、方法及服务质量保障”，No. 2015CB352202；
3. 国家自然科学基金重大项目课题“面向演化的群智化软件建模与构造方法”，No. 61690204。

攻读博士学位期间获得的主要奖励与荣誉

1. 奖学金：
   (a) 2016 年南京大学优秀博士研究生创新能力提升 A 计划；
   (b) 2015 年国家奖学金、国家公派留学资助；
   (c) 2014 年 Microsoft Research Asia Fellowship Award (微软学者奖学金)；
   (d) 2012 年南京大学研究生优秀奖学金。
2. 主要荣誉：
   (a) 2017 年“龙芯杯”全国大学生系统能力培养竞赛优秀指导教师 (团队位列全国第 2 名)；
   (b) 2016 年南京大学学生年度人物；
   (c) 2014 年“华为杯”苏-鲁程序设计竞赛冠军；
   (d) 2012 年腾讯编程马拉松竞赛亚军、南京大学优秀研究生。

致　　谢

感谢我的导师吕建教授。吕老师严谨的治学态度、开阔的视野、敏锐的直觉、科学家的风度都让人景仰。吕老师总是能站在更高的视角上审视我所做的研究工作，并在方法学层面给予富有智慧的指导意见。

感谢许畅教授和马晓星教授。感谢两位老师对我研究工作的指导和生活上无微不至的关怀，用宽松又严格的方式培养了我：被改得面目全非的第一次投稿；自信满满的幻灯片却被从头推翻；两位老师无私地帮助我成长和进步，这些痛苦仿佛都变成了最美好的回忆。

感谢俄亥俄州立大学的秦锋副教授在我赴美访问期间的悉心指导，同时也感谢秦老师实验室的徐尔茨和陈海骋，陪伴我顺利完成了访学的任务。感谢我在美国的室友杨琛和安心，让我在异国他乡却始终有家的感觉。

感谢香港科技大学的 S. C. Cheung 教授和 Charles Zhang 副教授、德克萨斯州 A&M 大学的 Jeff Huang 助理教授、南京大学软件研究所的陶先平教授、徐锋教授、黄宇教授、胡昊副教授、曹春副教授、余萍副教授、马骏老师、汪亮老师、姚远老师、魏恒峰老师、张建莹老师、王金路女士等在科研和生活中给我的关心和帮助。感谢南京大学计算机系的宋方敏教授、袁春风教授、尹一通教授给我的指导和启迪。

感谢南京大学计算机软件研究所的同学们，陪伴我度过读博的时光。

感谢我的父母和爱人，没有你们就没有我今天的成绩。