基础软件性能缺陷检测研究综述

何浩辰[1)] 李姗姗[2)] 贾周阳[2)] 姚懿恒[3),4)] 张元良[2)] 王 戟[2)] 廖湘科[2)]

[1)](卫星信息智能处理与应用技术重点实验室 北京 100080)

[2)](国防科技大学计算机学院 长沙 410073)

[3)](清华大学集成电路学院 北京 100084)

[4)](中国科学院 北京 100864)

摘 要 一直以来,软件性能缺陷给企业造成了巨大的经济损失。在性能缺陷流入生产环境之前,及时检测和修复缺陷可以有效预防性能故障,降低经济损失。然而,不同于一般软件缺陷,性能缺陷更加难以检测,具体表现为触发条件更加苛刻、表现症状更加隐蔽、缺陷类型更加多样等方面。当前已有研究从多个方面提出自动化的性能缺陷检测方法,形成了三个流派:一是基于特定模式的性能缺陷检测,首先调研特定类型性能缺陷的特征,然后设计针对性的模式匹配方法检测缺陷;二是基于性能测试的缺陷检测,从对性能缺陷触发条件以及其症状特征的理解出发,一方面提高触发缺陷的概率,另一方面挖掘有效的性能测试预言;三是基于 Profiling 的传统缺陷检测,首先假设程序执行最慢的代码段可能是性能缺陷,采用各种程序分析技术定位软件性能瓶颈,预测潜在的性能缺陷。本文系统性研究了 104 篇相关高水平论文,对现有研究工作进行归类和分析,总结了现有研究的不足和面临的挑战,归纳出性能检测实践中的一些通用共识,并对未来的研究趋势进行了展望,总结了 7 个未来可能的研究方向,对下一步工作具有一定指导意义。

关键词 软件性能缺陷;软件缺陷检测;综述

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2025.00210

Survey on Performance Bug Detection in System Software

HE Hao-Chen[1)] LI Shan-Shan[2)] JIA Zhou-Yang[2)] YAO Yi-Heng[3),4)]

ZHANG Yuan-Liang[2)] WANG Ji[2)] LIAO Xiang-Ke[2)]

[1)](Key Laboratory of Satellite Information Intelligent Processing and Application Research, Beijing 100080)

[2)](School of Computer Science, National University of Defense Technology, Changsha 410073)

[3)](School of Integrated Circuits, Tsinghua University, Beijing 100084)

[4)](Chinese Academy of Sciences, Beijing 100864)

Abstract As users' requirements for software functionality continue to grow, expectations for software service quality increase, and the external environments for deploying software become more diverse, software systems also face greater challenges. As one of the most notorious of them, software performance bugs have long caused significant economic losses for businesses. Detecting and fixing performance bugs before they make their way into production environments can effectively prevent performance failures and reduce economic losses. However, unlike general software bugs, performance bugs are more challenging to detect. Firstly, unlike general software bugs, the symptoms of performance bugs are more subtle. When a performance bug is triggered,

收稿日期:2024-02-04;在线发布日期:2024-09-12。本课题得到国家自然科学基金(62272473,62202474)、湖南省科技创新计划(2023RC1001)资助。**何浩辰**,博士,助理研究员,主要研究方向为软件性能与可靠性、软件测试、边缘计算。E-mail:hehaochen13@nudt.edu.cn。**李姗姗**(通信作者),博士,教授,博士生导师,主要研究领域为软件可靠性、智能软件工程。E-mail:shanshanli@nudt.edu.cn。**贾周阳**,博士,副研究员,主要研究方向为软件可靠性、通用操作系统。**姚懿恒**(通信作者),博士研究生,主要研究方向为智能软件、工业软件技术。E-mail:yaoyh@cashq.ac.cn。**张元良**,博士,讲师,主要研究方向为软件可靠性、软件演化。**王 戟**,博士,教授,博士生导师,主要研究领域为软件可靠性、形式化验证。**廖湘科**,博士,研究员,博士生导师,主要研究领域为系统软件、通用操作系统。

it does not exhibit explicit characteristics (such as logs or crashes) but only manifests as slow performance. Secondly, the conditions required to trigger performance bugs are more strict. They typically involve higher loads, specific environments, or particular configurations. Thirdly, the root causes of software performance bugs are more complex, leading to longer diagnostic times compared to general bugs. Finally, fixing performance bugs is also more challenging. Consequently, to develop automated tools for detecting performance bugs, researchers often need to conduct preliminary feature research and analysis, address various challenges based on their findings, and then design targeted solutions. Current research has proposed automated performance bug detection methods from multiple perspectives, leading to three main approaches. Pattern-based methods: these approaches usually first investigate the characteristics of specific types of performance bugs, and then design targeted pattern-matching methods to detect these bugs. Testing-based methods, which start from understandings of the triggering conditions and symptom characteristics of performance bugs, these methods aim to increase the probability of triggering bugs, while also uncovering effective performance testing oracles. Profiling-based methods: these methods start with the assumption that the slowest code segments in a program are likely performance bugs. It uses various program analysis techniques to locate software performance bottlenecks and predict potential performance bugs. Currently, there is limited systematic research on software performance bugs. Existing reviews include load testing, but load testing differs from performance testing in that it focuses more on system functionality's availability and stability under extreme loads, and concerns about performance degradation after version updates. In contrast, performance bugs can be triggered under normal load, specifically manifesting as software running significantly slower than expected, complementing the goals of load testing. Also, existing reviews cover performance optimization and bug detection in Android. However, software architecture differences across operating systems result in varied bug characteristics, leading to distinct characteristics and detection methods for performance bugs in Android software compared to performance bugs discussed in this paper. To this end, this paper systematically reviews 104 high-quality papers on the subject, categorizes and analyzes existing research, summarizes the shortcomings and challenges faced, and also synthesizes some general consensus in performance bug detection practices, including selection of evaluation datasets, evaluation of defect detection capability, and handling of performance instability. Finally, we offer an outlook on future research trends, summarizing seven potential future research directions, which could provide guidance for subsequent research on performance bug detection.

Keywords　software performance bug; performance bug detection; survey

1　引　言

基础软件是国家信息产业发展和信息化建设的重要基础和支撑[1]，已经广泛应用于航天、金融、民生等各个领域，使得互联网产业逐步走向“软件定义一切”的时代。基础软件主要包括数据库、中间件、编译器等各类在应用软件与硬件资源之间提供基础服务的软件。

随着用户对软件功能需求的不断增长、对软件服务质量要求的不断升高、部署软件的外部环境多样变化，基础软件系统的发展也面临更多挑战。一方面，使用软件的用户数量不断增加，软件所承载的数据量也飞速增长。另一方面，用户对于软件的要求也越来越高，功能从单一到多样，需求从简单到定制化，安全要求从一般到严格。使得软件实现看似的简单功能却要耗费大量资源，一旦资源运用不合理，则可能会出现性能故障。

因此，有效预防性能缺陷流入生产环境导致性能故障，是软件各方参与者关注的焦点。在对 148 家企业的调查中，92%的企业将提高软件性能列为最重要的任务之一[2]。近年来，软件性能故障导致巨大

的经济和客户损失[3]。美国 Amazon 公司指出,网页每增加 0.1s 延迟,将直接导致 1%的销售损失[4];因性能缺陷导致用户弃用 Firefox 浏览器的概率是功能缺陷(Functional bug)的 8 倍[5]。同时,修复性能问题耗费高昂成本。Mozilla 开发者每月需修复5~60 个性能缺陷[5],61%的性能缺陷修复需要 5 周以上时间[6]。可以看出,性能缺陷一旦流入生产环境,将会带来大量的用户、流量损失以及高昂的开发者维护成本。

为预防损失,软件开发和测试人员在软件发布前,采用多种手段检测性能缺陷。然而,性能缺陷的检测面临大量挑战:首先,不同于一般的软件缺陷,软件性能缺陷症状更加隐蔽,缺陷被触发时不具备显式特征(如日志、崩溃等),仅表现为运行缓慢;其次,性能缺陷的触发条件更加苛刻,通常需要较大负载、特定环境、特定配置等条件;然后,软件性能缺陷的根因更加复杂[5,7-9],诊断时间相较一般缺陷更长;最后,性能缺陷的修复也更加困难。因此,为研发性能缺陷自动检测工具,研究人员通常需要开展先导性特征调研分析,基于发现破解各类难题,然后进行针对性设计。基于这一基本途径,软件性能缺陷检测工作形成了三大流派。

(1)基于特定模式的性能缺陷检测。研究人员首先调研特定类型性能缺陷的特征,然后设计针对性的模式匹配方法检测缺陷。

(2)基于性能测试的缺陷检测。从对性能缺陷触发条件以及其症状特征的理解出发,一方面提高触发缺陷的概率;另一方面挖掘有效的性能测试预言(Test oracle)。

(3)基于 Profiling 的传统缺陷检测。首先假设程序执行最慢的代码段可能是性能缺陷,采用各种程序分析技术定位软件性能瓶颈,预测潜在的性能缺陷。

目前,存在少量工作对软件性能缺陷相关研究进行过系统性归纳。Jiang 等人[10]对负载测试(Load testing)相关研究展开综述,从负载测试的设计、执行和评估三个角度总结了一系列最佳实践,为负载测试的应用提供全面参考。负载测试和性能测试相比具有以下差异:一是更加侧重极限负载下系统功能的可用性和稳定性;二是更加关注软件性能在版本更新后不能变差。然而性能缺陷则是在常规负载下即可触发,具体表现为软件运行速度显著低于预期,与负载测试为关注的目标为互补关系。Hort 等人[11]对安卓软件中的性能优化和性能缺陷检测展开综述。然而,不同操作系统下软件架构差异明显,缺陷具有不同特征。例如,安卓软件具有以下特征(不限于):大多带有图形化界面;且图形计算和算数计算共享同一片内存;需要关注图形和算数计算不合理使用一类的性能缺陷[12];需要频繁重启;重点关注能耗而非运行效率等等[13]。Linux 操作系统中的基础软件通常无图形计算,也不会频繁重启,且对低能耗的要求显著低于安卓软件。这些差异导致安卓软件的性能缺陷与本文关注的性能缺陷具有不同特征和不同检测方式。

基于上述现状,本文针对软件性能缺陷相关研究的现状和进展进行综述。首先阐述本文的综述研究方法;其次,针对软件性能缺陷的概念进行梳理和定义,确定本文的研究目标和调研对象;然后,分别对基于特定模式的性能缺陷检测、基于 Profiling 的传统缺陷检测、基于性能测试的性能缺陷检测三类主要方法进行系统性的脉络梳理,并关注研究的进展和不足;之后,对性能缺陷检测相关研究中的常见的共性难点问题和处理措施进行总结,为以后研究提供参考;最后,对性能缺陷检测相关研究的未来方向进行展望。

2 研究方法

本文遵循系统文献综述方法(Systematic Literature Review,SLR)[14]展开研究。该方法已被大量综述类研究工作采用,具有较强的权威性。本研究分为论文筛选和综述实施 2 个步骤。

首先,以“performance bug”、“performance problem”、“configuration”及对应中文为关键词在 Google scholar 论文搜索引擎和 ACM Digital Library、IEEE Xplore Digital Library、Elsevier ScienceDirect、Springer Link online Library、CNKI 等在线数据库中全面搜索近 10 余年(2013~2024)性能缺陷相关的期刊及会议论文,并重点关注以下方面:

(1)论文出处。性能缺陷相关研究数量众多,本文旨在对本领域高水平研究成果进行总结分析,从而得出更具代表性的分析结果。因此本文仅研究高水平论文,即经同行审阅评议,正式发表在 CCF-A 类(含中文)会议或期刊论文;

(2)论文类型。研究(research-track)类或工业界(industrial-track)论文长文,正文 8 页及以上;

(3) 论文主题。性能缺陷相关研究。区别于为优化软件或系统性能提出新的设计、配置性能调优、性能回归问题、超时缺陷问题。

为防止关键词筛选遗漏关键论文，同时避免遗漏部分发表在 CCF-B 类、C 类或不属于推荐列表，但却受到高度认可的论文，补充以下规则：

(1) 获得较高引用(年均 20 次以上)的 CCF-B 或 C 类论文或短文(正文 7 页及以下)；

(2) 对所选论文的引用(特别是相关工作章节涉及论文)进行人工筛选，补充上述方法遗漏的研究工作；

(3) 至少两名作者通过人工阅读文献内容并讨论，共同判断所筛选的文献是否为软件性能缺陷相关的研究工作。

最终，本文筛选出 104 篇相关论文，分布如图 1 所示。数据显示，发表在 ICSE 上的论文数量最多，且论文发表数量最多的会议均集中于软件工程领域(71 篇，68.3%)，而发表在 SOSP、OSDI 等计算机系统会议中的论文数量较少，仅 9 篇(8.7%)。发表在计算机体系结构、程序设计语言领域的论文共 17 篇(16.3%)。可以看出，领域针对软件性能缺陷的研究主要多从软件本身入手。

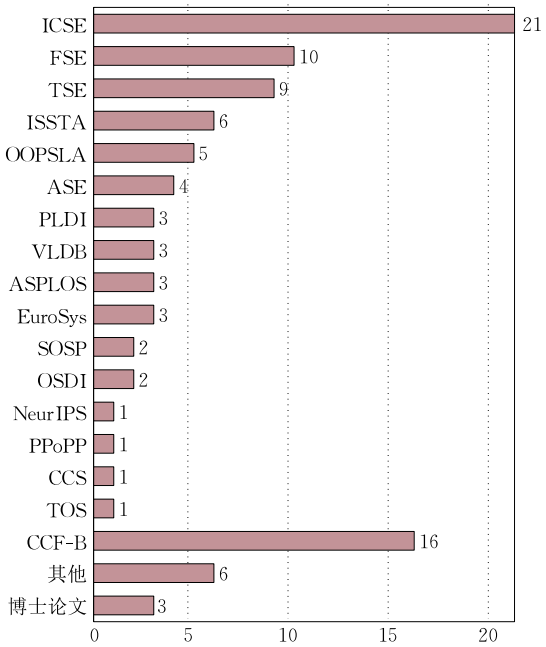

图 1　本文研究的软件性能缺陷相关论文数量统计

3　软件性能缺陷的内涵与研究路线

性能缺陷(Performance bug)这一概念定义为：因软件错误的代码逻辑导致软件运行缓慢[5]。随着研究的不断深入，形成了更为完善的共识[15-20]。从表象上看，性能缺陷是指：不影响软件正确运行，但会使得用户感知到程序运行的效率显著低于预期的代码缺陷；从内在原因上看，性能缺陷是指消耗了计算存储等资源，但却没有产生有用的结果的代码。

相比于传统的功能缺陷，性能缺陷具有显著差异。一般地，功能缺陷可能导致软件崩溃、数据丢失、宕机、输出结果错误(fail-stop)等故障；而性能缺陷可能并不会表现出上述显示特征，原因在于，程序运行缓慢(non-fail-stop)的判别更为主观，这使得性能缺陷的自动化检测更具挑战。

3.1　软件性能缺陷实例

图 2 展示了两个实际性能缺陷。在缺陷 GCC#27733[21] 中，结构 hash_entry。t 用于保存哈希值，在递归函数 mult_alg 中，else 分支包含大量计算，而通过哈希值记忆以往迭代计算结果可大大减少进入 else 分支的次数，从而避免冗余的递归。但因开发者疏忽，使用了错误的哈希值数据类型(unsigned int)，导致 if 条件常常无法被满足，引起大量冗余递归调用，最终致使一个本来仅需不到 1 s 即可完成编译的文件消耗 17 min 才完成。

缺陷实例 1：GCC#27733

```
  struct hash_entry {
-       unsigned int t;
+       HOST_WIDE_UINT t;
  };
  void mult_alg(… HOST_WIDE_UINT t, …) {
    hash_index=hash(t);
    if (alg_hash[hash_index]。t==t){
      //复用上次递归结果
    }else{
      //重新进行计算，消耗大量资源
    mult_alg (…);//递归
    }
  }
```

缺陷实例 2：MySQL#21727

```
//该段代码在每次数据库进行子查询(sub-select)时被执行
- sort_keys=my_malloc
            (sort_buffer_size, …);
+ if (!table_sort。sort_keys)
+    sort_keys=my_malloc
            (sort_buffer_size, …);
  …
- x_free(sort_keys);
+ if (!subselect)
+      x_free(sort_keys);
```

图 2　性能缺陷实例

在缺陷 MySQL#21727[22] 中，MySQL 会在执行子查询时分配一个缓冲区用于存放临时数据。在缺陷版本中，每次迭代子查询都会重新分配该缓冲

区空间，造成了大量的系统开销，当用户增大该缓冲区的大小时，额外的开销变得更加显著。正确的修复方式是复用缓冲区，防止重复内存分配。

相较于一般的功能缺陷，在上述两个性能缺陷实例中，既没有发生崩溃、拒绝服务、数据丢失等严重后果，软件也没有给出日志、告警等辅助信息，大大增加了性能缺陷的检测难度。传统的功能缺陷检测手段如单元测试、模糊测试等无法发挥作用。

3.2 性能缺陷相关的其他类型缺陷

大量研究工作关注软件的性能问题，如配置性能调优[23]通过调节软件配置优化软件性能(Performance tuning)，但此类方法并不能检测性能缺陷；用户配置错误(Misconfiguration)可能会导致软件资源运用不合理从而引发性能下降[24]，但代码中并无性能缺陷；面向性能的架构设计[25]相关研究一般针对负载特征或硬件特性重新设计数据结构和软件架构，从而优化性能，不在本文综述范围之内；由于硬件问题导致的软件性能下降[26]通常无法被软件开发者修复，而是需要内核提供相应支持，故亦不在本文研究范围。性能回归问题(Performance regression)[27-28]是一种典型的性能缺陷检测方式，是指通过对比测试，检测软件持续开发过程中，某次代码更新时引入的性能缺陷。性能回归问题具有较为特定的场景和检测、诊断框架，相关研究针对框架中的不同环节展开了大量研究。而本文面向一般性场景，旨在梳理针对一般性能缺陷的解决方案，与性能回归问题为互补关系。超时缺陷(Timeout bug)[29]也是一类非功能缺陷，但此类缺陷会导致软件彻底卡死，与本文关注的导致软件运行缓慢的性能缺陷不同。此外，在分布式软件中的性能缺陷产生的原因也有较大差异[30-33]，这类缺陷通常与分布式系统的特性(如一致性、协同计算)相关，其表现通常为某一节点的异常或崩溃等导致整个系统效率下降，也被称为“Gray Failure”，不在本文综述范围之内。

3.3 软件性能缺陷检测的研究路线

通过广泛调研相关研究工作，本文首先将性能缺陷检测相关研究按照是否依据缺陷根因模式(Root Cause Pattern)分为两大类：基于特定根因模式的检测方法和无根因模式的检测方法，根因模式是指性能缺陷发生时，程序暴露出的特定模式(如某一系统调用序列)。后者还可根据是否有表征特征(如吞吐量特征、延迟特征等)分为基于性能测试的方法和基于 Profiling 的方法，如图 3 所示。每类技术路线基本遵循“先理解、后检测”的研究模式。

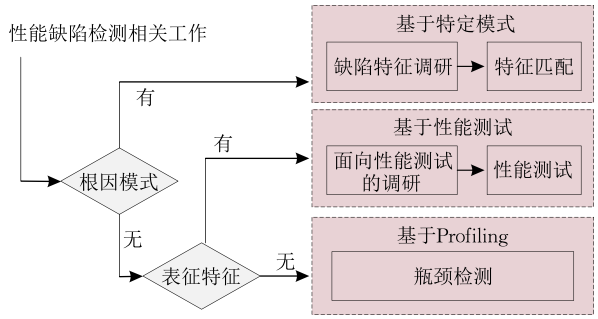

图 3 性能缺陷相关研究工作分类

基于特定特征模式的检测技术通常需要总结归纳历史性能缺陷的根因特征模式，然后针对性设计各种监测手段，从而匹配模式，相关工作主要从针对编译优化的性能缺陷检测和基于动态模式的性能缺陷检测 2 个角度开展研究。由于依赖特定模式，此类方法的通用性相对不足。基于性能测试的方法则无需依赖特定模式，但需要理解性能缺陷被触发时，性能表征表现出的特征(即测试预言)，相关研究通常从测试样例生成和测试预言构建 2 个角度出发研究。当特定模式和性能表征特征均不存在特征时，研究人员通常假定程序运行相对缓慢的代码段可能代表性能缺陷。基于这一假设，采用基于 Profiling 的方法，分别从快速搜索使程序执行缓慢的输入，和综合分析推荐除最慢代码段外的可疑代码以提高检测精度 2 个方面入手展开研究。

图 4 展示了各类研究工作(共计 73 篇)的数量分布。可以看出，在三种流派的性能缺陷检测方法中，基于特定模式的检测技术研究占比最高(42%)，而基于性能测试的方法和基于 Profiling 的技术则相对较少(分别占比 23%和 16%)。本文将在第 4 节至第 6 节分别对 3 种流派的方法进行梳理和分析。

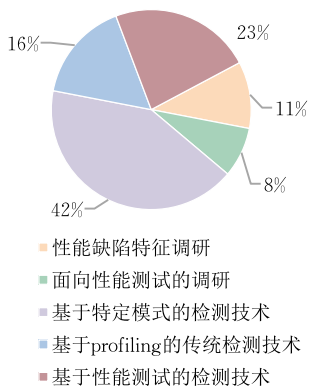

图 4 性能缺陷相关研究工作数量分布

4　基于特定模式的性能缺陷检测

尽管性能缺陷被触发时通常不会产生显式信息，但大量调研结果显示，性能缺陷具有各种特征模式。因此，研究人员设定各种规则，采用各类运行监控手段（如代码插装、内核监控等）获取关键信息，匹配特定模式，检测缺陷。此类方法高度依赖于模式。研究人员需要挖掘大量历史缺陷，或者分析典型缺陷，构建特征规则库，而后采用动静态程序分析、代码静态插装、动态监控等技术匹配模式，如图 5 所示。

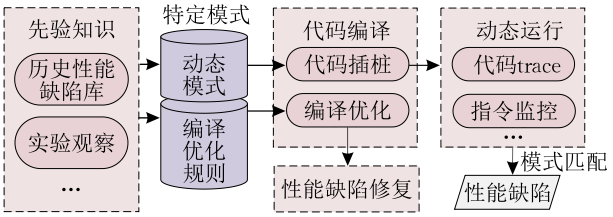

图 5　基于特定模式的缺陷检测一般方法

4.1　性能缺陷特征理解

性能缺陷特征理解（Understanding Performance Bug）是指在性能缺陷被引入、暴露、定位和修复的整个过程中，通过人工调研的方式，归纳性能缺陷在不同环节表现出的特征的过程。这些特征可以有效帮助开发者在相应环节设计更加具有针对性的自动化工具。图 6 展示了性能缺陷特征理解相关工作所关注的环节及相应的特征：开发人员编写软件时引入性能缺陷，缺陷在测试环境或生产环境中被触发（触发条件特征），测试员或用户观测到软件运行缓慢（观测手段特征），进而诊断症结（根本原因特征），最终修复缺陷（修复方式特征）。

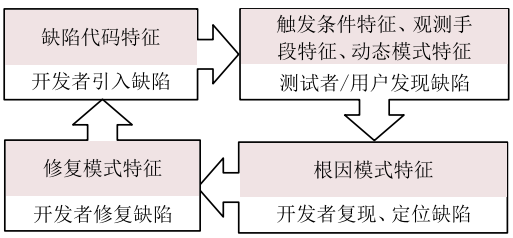

图 6　性能缺陷特征理解相关工作关注的环节及相应的特征

表 1 展示了这些性能缺陷特征理解工作覆盖的方面。其中部分研究除了在特征理解方面给出了洞察，还基于洞察针对其中典型的缺陷提出了解决方案，并进行了评估。由于这些解决方案通常仅针对某一小类典型缺陷设计，本文侧重关注这些研究工作在缺陷特征理解方面的贡献。

表 1　软件性能缺陷特征调研相关工作

研究工作	软件类型	触发条件	根本原因	观测手段	修复方式
PLDI'12[5]	服务端	√	√	—	√
ESEM'16[7]	服务端	※	√	—	√
ASE'18[34]	服务端	※	—	—	—
TSE'21[35]	通用库等	—	√	—	√
ISSRE'19[36]	基础软件	—	√	—	√
FSE'22[37]	人工智能库	√	√	—	√
ICSE'16[38]	JavaScript	—	√	√	※
OOPSLA'14[39]	基础软件	—	—	※	√

注："√"指给出洞察，"※"指给出洞察并提出可能解决途径；"—"指由于作者对性能缺陷的关注点差异，通常会着重在某些方面给出洞察，而可能不会关注其他方面。

Jin 等人[5]对软件性能缺陷展开系统性的调研，认为制约软件发挥性能的因素不再是硬件和编译器，而是软件本身，并首次给出性能缺陷的较为权威的定义：修改少量代码可大幅提高软件运行速度，同时保证功能不变（"Defects where relatively simple source-code changes can significantly speed up software, while preserving functionality."）。该调研涵盖了五款服务器端大型开源软件的 109 个性能缺陷。总结出性能缺陷的故障症状、触发条件、根本原因等特征，调研指出，相比于一般的功能缺陷，性能缺陷隐藏更深，具体表现在，近半数性能缺陷同时需要较大的特定负载才能被触发。但同时性能缺陷也存在一定的共性模式，进而归纳出 25 条共性模式，粗略筛选潜在性能缺陷。诊断与修复方面，尽管性能缺陷报告（But report）中的信息通常更充分，但其修复的时间反而更长、需要的人力更多、造成用户流失的概率更高。原因在于，性能缺陷的根因理解往往需要更加系统的专业知识。Jin 等人[5]的研究为后续软件性能缺陷相关工作奠定了重要基础，使得性能缺陷进入了研究人员的视野，提出了应通过自动化方法系统地检测性能缺陷，并指出了其中的主要科学问题：性能缺陷自动检测需要构建有效的测试预言。

Han 等人[7]发现，软件配置（Software configuration）作为控制资源和软件运行策略的接口，易导致软件缺陷。在服务端软件（如 MySQL、Apache-Httpd）中，配置相关的性能缺陷占比超过半数，且这些配置相关性能缺陷的根因常与资源（如 CPU、内存、I/O、同步）使用低效相关。研究还发现，绝大多数的配置相关性能缺陷都需要修复相关一代码而非简单调整配置取值。在上述研究基础之上，Han 等人[34]进一步发现触发配置相关性能缺陷的配置项具有时间局部性，为后续配置相关性能缺陷的检

测提供了指导。

Zhao 等人[35]对来自不同领域的 13 个开源通用软件中 570 个历史性能问题进行调研。归纳出 8 类根因类型及对应的修复策略。该团队还发现 27%的性能问题都需要从软件设计层面进行优化才能解决,而非简单修改几行代码。同时,归纳出 4 种典型的设计级优化模式,包括经典设计模式、更改传播、优化克隆和并行优化。该团队发现,软件测试集中,极少测试用例关注性能缺陷,即便开发者为某一修复好的性能缺陷编写了额外的测试用例,但通常也仅用于测试该修复是否影响功能。这预示着面向性能的测试仍较为初级。

Chen 等人[36]从演化(code commit)的角度,调研了 13 个软件总计 733 个修复性能缺陷的代码补丁,发现这些缺陷补丁存在 8 种固定模式,包括如低效锁、值记忆(Memorization)、循环优化等等。还对不同类型的缺陷修复模式、修复难度进行了定量分析。尽管工作较为扎实,但缺陷的分类仍较为笼统,例如占比最高类别"fast-path"是指缺陷补丁使程序执行了更快的一条代码路径,该信息无法对研究人员进一步研究提供有效支撑。

Cao 等人[37]针对使用了 TensorFlow 和 Keras 框架的软件的性能缺陷展开特征分析,调研了 210 个历史性能缺陷,归纳出 58 个低效模式,例如调用"batch"函数前调用"map"会导致低效等等,为研究人员理解深度学习软件中可能存在的性能缺陷给出了范例。

Selakovic 等人[38]发现,在 JavaScript 软件中,大部分(52%)的性能问题是由于用户使用了次优的 JavaScript API 导致,例如遍历某个列表时,最优方式应是首先调用 list。keys()然后依次用键访问,而非直接采用 for-in 方式。研究还发现,修复 JavaScript 程序通常只需要修改少量代码,并且基本不会影响软件的可维护性和可理解性。尽管部分性能缺陷存在特定模式,但大部分无法用静态方法自动检测。

Song 等人[39]通过调研基础软件中的 65 个历史性能故障诊断的流程,发现超过 80%的故障都是用户通过对比的方式发现,即提供一组导致性能故障的测试样例和一组正常的测试样例,而两组测试样例本身的差异较小。该发现为基于蜕变测试的性能缺陷检测提供了启发。

小结。性能缺陷特征理解对性能缺陷检测起支撑作用,相比一般的功能缺陷,性能缺陷具有以下特点:

(1)症状更隐蔽。性能缺陷发生时,系统功能正常,通常没有日志等显著信息。一般表现为运行缓慢,但缓慢的判断较为主观,不存在通用性的阈值。

(2)触发更困难。触发功能缺陷需要执行到特定路径,而性能缺陷代码被执行时常常没有明显特征,需要较大负载、特定配置和特定环境才能显著显现。

(3)根因差异显著。性能缺陷是系统在功能正常前提下所表现的问题,因此性能缺陷的根因和功能缺陷的根因通常不同。使得功能缺陷检测和诊断手段无法直接迁移到性能缺陷中。

尽管大量研究对传统基础软件的性能缺陷进行了深入调研,但随着人工智能的发展和普及,仅有较少研究对深度学习框架的性能问题进行系统性总结梳理,未来研究应关注人工智能库本身、使用库的软件的性能问题,降低智能程序发生性能缺陷的概率。

4.2 针对编译优化的性能缺陷检测

编译优化(Code Optimization)是检测、消除软件性能缺陷的首要环节,主要通过静态规则对代码进行优化。然而代码逻辑复杂,静态规则无法完成所有优化,导致编译出的软件可能存在性能问题。部分研究针对现有规则的缺陷展开研究,还有研究通过对缺陷的调研总结特征对规则库进行补充。

(1)遗漏编译优化检测

此类研究的思路是通过为代码插入"理应被静态规则优化的代码"(如 Dead Code),识别编译器是否能如预期将其优化。如不能,则认为这些遗漏编译优化(Missed Optimization)会导致软件出现性能故障。

Gong 等人[40]从循环结构入手,通过制造大量语法不同但语义相同的嵌套循环结构,分析不同 C 语言编译器对嵌套循环的编译优化,通过对比,识别优化效果较差的编译器,从而检测性能缺陷。Theodoridis 等人[41]指出,前人方法大多针对某个特定编译优化环节的缺陷展开研究,覆盖面较窄。而编译优化中,无用代码消除(Dead Code Elimination)涉及整个优化流程的大量环节,具有较好的利用价值。因此作者通过在源码中插入无用代码,测试不同 C 语言编译器对无用代码的优化情况来检测性能缺陷。

(2)编译优化规则扩展

编译优化规则缺乏性能缺陷中静态代码模式的指导,因此 Jin 等人[5]通过对 109 个缺陷的调研,总结出若干条静态规则检测缺陷。例如,在多线程程序

中，应避免直接使用系统的 random()函数；在循环中，doTransact()应被替代为 aggregateTransact()函数等等。相比之下，Olivo 等人[20]聚焦某一类型的性能缺陷。作者发现，Java 基础库软件中普遍存在重复计算性能缺陷，即循环反复遍历某一数据结构，但结构中数据却不会发生变化。基于此，作者通过静态分析数据结构的修改和遍历，检测重复计算缺陷，避免了动态方法依赖于动态负载的问题。

上述方法通常可将静态模式以编译优化（例如在编译时增加一个额外 PASS）的方式直接实现缺陷的检测和修复。但众所周知，基于纯静态方法所能检测的缺陷（尤其是性能缺陷）仍十分有限，大量工作基于动态方法展开研究。

4.3　基于动态模式的检测

基于 Profiling 的传统性能缺陷检测方法的核心是通过遍历测试输入，搜索在特定输入下运行最慢的代码段。然而，执行最慢并不代表存在缺陷；且另一方面，即使测试输入执行到了性能缺陷代码，性能瓶颈检测工具也可能无法捕捉。这是因为当输入负载较大时，执行正常代码所需的时间可能会超过执行缺陷代码的时间，导致该类方法出现误报或者漏报。此外，触发性能缺陷常常需要特定负载，然而 Profiling 技术需采用较为通用常见的负载。基于上述局限，大量工作通过匹配特定的代码或运行时状态的低效模式特征检测性能缺陷。

4.3.1　低效内存访问

低效的内存访问模式是最为常见的一种低效模式，如图 7 所示，按照软件发布前检测和在生产环境检测划分，可分为在线检测和离线检测；离线检测按照模式的级别可分为代码级和指令级低效模式检测，其中指令又分为低效读和低效写指令。

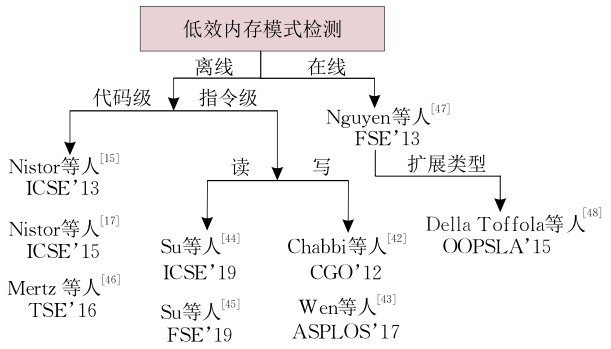

图 7　基于低效内存模式的缺陷检测研究脉络

（1）指令级离线检测。Chabbi 等人[42]发现，软件中存在大量连续写（Dead-write，即两次写入操作之间没有读操作）冗余操作，且受限于程序规模、指针别名等因素，无法被传统编译器优化。基于此，该团队设计实现重复写动态监测工具 DeadSpy，通过修复这些缺陷，使软件性能平均提 14%。与之类似地，Wen 等人[43]设计实现工具 REDSPY，检测软件中的重复写（即某次读前后的两次写的值总是相等）性能缺陷。

上述工作[42-43]研究了冗余的写操作导致的性能问题。相对地，Su 等人[44]发现在 Java 程序中，重复的读指令（即多次从同一地址读取相同值，或连续从相邻地址读取相同值）也常是各种性能故障的共同特征，并基于此设计实现 LOADSPY，利用动态技术直接在 Java 二进制层面开展性能缺陷检测。Su 等人[45]指出，前人方法大多依赖插装以获得低效内存访问特征信息，然而插装方法存在两点局限：一是额外开销巨大，无法在正式发布的软件版本中应用，无法真实服务于实际系统；二是无法捕获更加底层的低效模式，例如，代码“data[j]=data[i]−k; data[i]+=k;”在 i 与 j 永不相等的情况下，编译成机器码后会对“data[i]”重复读取内存 2 次。类似地，该团队总结共四类低效模式，然后基于硬件性能检测单元（Performance Monitoring Unit）以较低代价对运行中的软件进行在线监控，大大降低了额外开销并，同时还测出了前人无法检测的缺陷。

（2）代码级离线检测。上述工作[42-45]从底层读写指令层面分析性能缺陷模式，部分工作从代码语义层面展开研究。Nistor 等人[15]通过广泛调研发现，超过半数的严重性能故障都是由代码中的循环结构导致的，因此聚焦循环结构，全面总结了四类低效的内存访问模式，如 3.1 节中图 2 缺陷实例 1 所示，程序重复地执行某个循环，且产生的结果在以前循环中已经计算完毕，造成了性能缺陷。该研究提出了一种基于插桩的性能缺陷检测技术 Toddler，检测软件运行时的因低效内存访问模式导致的性能缺陷。例如，对形如“for(…;…;…){write(…);}”的代码段，工具将分析每次循环结束是否产生有效结果（即发生写操作），若绝大多数情况下均无结果，将判定为性能缺陷。原因在于，可以通过施加判断条件，仅针对会产生有效结果的情况才进入循环，否则直接继续（continue）循环。Song 等人[16]完善了模式的类型。具体的，该研究组[17]进一步调研了性能缺陷本身的影响和其补丁带来的副作用间的平衡问题，并发现开发者修复概率最高的性能缺陷存在某一特定模式：循环结构中，如果某一条件的满足后，后续循环迭代都无需进行时，应当及时跳出（break）循环。基于此，该团队调研发现了四类上述低效模式，检测程序循环结构中的性能缺陷。

缓存是一种典型的提高内存访问效率的方式，历史上已有大量研究针对缓存效率低下问题展开研究。Mertz 等人[46]从什么时机(when)、如何缓存(how)、哪些(what)数据、到什么地方(where)四个角度，详细调研了 Web 软件中，缓存使用的常见误区和最佳实现。

(3)在线检测。离线(软件发布前)检测缺陷的一大局限在于测试环境中构造的负载较为有限，而性能缺陷通常需要特定负载触发。因此，Nguyen 等人[47]提出应在运行时进行检测，即检测是否存在大量计算总是产生相同结果(Memory Boat)。为此，该团队分别从指令层、数据结构层、函数调用层分别监控软件运行时的重复计算，同时定位具体的低效代码。Della Toffola 等人[48]进一步在函数层面扩展了可复用优化的类型，认为可以通过记录程序的输入输出对，使第二次相同的输入可以快速返回结果。

4.3.2 低效锁

上述工作关注内存中的低效模式导致的性能缺陷，还有部分工作聚焦因低效的锁和同步机制导致的性能缺陷。这类问题可进一步归纳为两种思路：一是降低程序执行某个独立任务的绝对时间(on-CPU Time)，可以通过分析关键路径(Critical Path)，找出最影响程序执行总时间的代码段集合，并寻找优化可能。另一种是降低程序的等待时间(off-CPU Time)，可以通过分析程序中等待的事件所消耗的时间，尽可能解耦一些长时间等待事件。

第一类方法的思路比较直观，一般需要首先定义关键路径，然后进行优化分析。Jin 等人[49]指出，在多线程程序中，低效锁、数据通信、数据依赖等均可能导致性能故障，但通常需要逐一排查。正是受这一特点启发，该文献提出了一种性能缺陷的图表征方式，首先执行大量负载收集程序执行过程中的性能情况，然后将多线程之间的多种依赖关系统一建立为 flowGraph，找到其中关键路径，并指导开发者在关键路径代码上进行性能优化。

第二类方法更为常见，代表性的有：Yu 等人[50-51]指出，已有工作大多针对多线程程序的正确性展开研究，而对其性能问题缺少探索。这些性能问题主要是由于线程间低效的等待导致，然而检测多线程程序的性能缺陷并非易事：需要特定输入触发特定线程间的执行次序，还缺乏有效准则判定缺陷。基于此，该团队首次提出，若增大软件测试负载，软件执行时间增加，但 CPU 的利用率下降，则判定为存在缺陷。上述方法是通过表征(软件执行时间、CPU 的利用率)特征来检测低效锁，Zhou 等人[52]进一步深入探索，构建程序执行过程中，所有等待事件构成的等待图(Wait Graph)，然后找到其中最长等待链条，从而找到程序的竞争瓶颈，此种方法相较于 Yu 等人[50-51]的方法，能够更加精准地检测缺陷代码段，开发者优化后，平均性能提升达 4.83 倍。

在多线程程序中，除了多个存在等待、锁和竞争关系的线程产生的性能缺陷，还存在无此类关系的线程因对资源的抢占使用，造成资源在不同线程中分配不合理导致的缺陷。Hu 等人[53]针对这一类问题，将系统中的资源隔离机制上浮到应用软件层，对不同线程进行动态的资源合理分配，虽然取得了很好的效果，但需要开发者在编写代码时，通过作者提供的接口设置动态优化的目标和策略。

4.3.3 特定软件的特定模式

(1)UI 程序。部分工作设计工具检测特定软件或特定场景下的性能缺陷。Pradel 等人[54]指出，传统的测试技术通常以代码覆盖率为目标，而在用户界面(UI)程序中，不同用户操作对应不同的处理代码，因此触发性能缺陷的关键在于构造某一特性顺序的用户操作，如果某一特定操作在操作序列的特定位置时，程序执行变慢，则该操作序列可能触发缺陷。基于此，该团队设计实现 EventBreak，基于性能变化反馈生成测试输入，检测用户界面软件中的性能缺陷。

(2)机器学习程序。Tizpaz-Niari 等人[18]发现，机器学习库中，输入相等规模的数据或相似的库函数参数，机器学习任务运行的时间可能产生巨大差异，从而产生性能缺陷。基于此，作者使用模糊测试的思想，迭代生成测试输入，不断搜索使得机器学习库函数运行时间差异变大，但输入本身规模差距较小的输入。该思想与 PerfFuzz[55]、SlowFuzz[56]接近，但更加适配机器学习库特征。Cao 等人[37]通过对 210 个 StackOverflow 中使用了深度学习框架软件的性能问题进行调研，总结出 58 个此类软件中的低效模式，并设计了静态检测工具，检测出大量历史未知的缺陷。然而该研究总结的模式相对较为固定，例如没有使用批处理调用深度学习库等等，未来工作仍有较大研究空间。

(3)ORM 程序。Chen 等人[57-58]、Shao 等人[59]针对基于对象关系映射(Object Relational Mapping，ORM)框架实现的软件中的性能问题展开研究，根据先验知识匹配“$N+1$ 查询”低效模式、“更新元素某个字段时将元素全部字段读出”低效模式等检测缺陷。Yang 等人[60]针对 ORM 软件，系统地扩展了模式的类型，调研了 12 个软件的 200 余个真实性能

缺陷，归纳出 9 种性能低效模式，检测此类软件中的性能缺陷。Chen 等人[61]在前人工作基础上，进一步调研、扩展了 17 类低效模式，并实现了一种编程语言无关的自动检测方法。

(4) SaaS 程序。Wang[62]针对 SaaS 软件的性能缺陷检测和诊断展开研究，从多个层次生成并抓取丰富的日志信息，为软件性能缺陷的识别提供判别数据；同时提出了一种基于受限玻尔兹曼机的缺陷诊断技术。该研究聚焦在线检测性能缺陷，需要在软件运行时额外生成和收集信息，额外开销较大。

(5) MarkDown 程序。Li 等人[63]发现，近年来，Markdown 编译器的性能故障报告持续增加，因此对 49 个历史性能缺陷展开调研，发现发生故障的根因在于，Markdown 编译器经常使用回溯算法实现其语言功能，而回溯可能会被精心设计的输入利用，从而导致性能缺陷。尽管限制回溯步数可以防止此类问题，但也会导致功能受损。基于此设计工具 MDPERFFUZZ，使模糊测试适配 Markdown 语法结构，以 CPU 满载为测试预言，检测出了 175 个历史未知的性能缺陷。

(6) 虚拟现实(VR)程序。随着 VR 程序的兴起，其性能问题也愈发受到关注。Nusrat 等人[64]通过对 45 个 VR 程序的 183 个性能缺陷补丁的分析，归纳出了一系列发现，主要包括优化 VR 程序性能需要对图形特效在适当时机进行简化、更好地利用好底层库的优化机制进行渲染、尽量减少对堆栈(Heap)的使用等。

4.3.4　基于记忆系统的检测方法

性能缺陷可能会重复出现，记忆已发生过的性能缺陷特征，可检测软件在后续演化中的类似缺陷。

Wert 等人[65]首先对造成软件性能故障的常见原因进行细粒度分类，并将已知故障根因特征建模。然后自动展开一系列性能测试，采用决策树算法匹配当前性能故障与已知性能原因特征进行匹配，辅助开发者理解故障可能的出错原因。该方法的根因分析粒度较粗，仅能针对 2 大类、11 小类常见根因进行分类，且需要执行大量测试，可用性有限。He 等人[66]指出，性能故障常常重复出现，例如基于同一库开发的不同软件可能都会因为库中某个性能缺陷而引发类似的性能故障。而已有的性能缺陷分类方法[65,67-68]通常基于有限的运行信息，因此无法捕捉重复出现的性能故障。因此该团队提出在运行时记录系统性能指标(如 CPU 利用率)、软件运行日志和函数调用 trace，然后挖掘三类信息源分别在出现异常事件前后的因果关系，从而为性能故障生成准确的标签。同时，还可挖掘出可疑的函数，定位缺陷代码。

4.4　本章小结

当前工作基于静态和动态模式从不同角度(内存、锁)针对不同类型软件检测性能缺陷。此外角度上，相比于内存和锁，I/O 和网络同样是制约软件性能的关键因素，然而当前少有研究针对二者导致的性能缺陷进行特征分析，设计解决方案；软件类型选择上，随着人工智能技术的发展，当前对当下流行的机器学习软件的性能问题分析仍处在较为初级的阶段，已有的缺陷类型总结仍是“一例一特征”，缺乏深度理解。另一方面，尽管基于特定模式的性能缺陷检测的准确率一般较高，额外开销也一般较小，但其的应用条件较为苛刻。一是该方法高度依赖历史已知缺陷归纳特征。对于刚刚发布的软件，无从得知其性能缺陷类型，也就无法设计工具有效检测性能缺陷。二是对于未曾调研过的软件，均需要领域专家对其历史缺陷库中的性能缺陷特征进行挖掘和归纳，泛化能力较弱。

5　基于 Profiling 技术的性能缺陷检测

软件开发或测试人员使用传统的 Profiling 工具(如 LinuxPerf)检测性能缺陷。由于性能缺陷缺乏显式的观测特征，该技术将程序执行缓慢作为检测目标，枚举不同测试输入组合逐渐使缺陷暴露。图 8 展示了基于 Profiling 技术的缺陷检测流程，首先开发者通过程序分析等手段确定需要进行执行时间分析的程序点位，然后迭代搜索使程序执行缓慢的输入，最终分析检测性能缺陷代码。

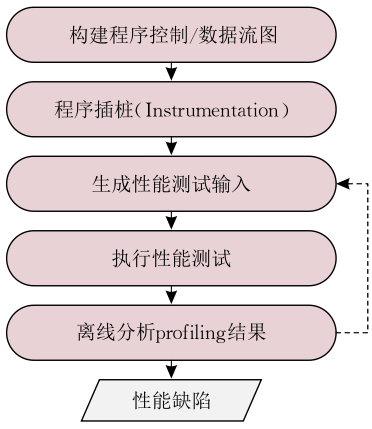

图 8　基于 Profiling 技术的一般流程

基于 Profiling 的方法主要面临两大挑战：(1) 软件输入一般为一系列元素的排列组合，存在组合爆

炸问题;(2)Profiling 输出结果复杂,如图 9 所示,包含大量函数的多层调用关系(下层函数调用上层函数),开发者难以快速检测出软件真正的缺陷,在该案例中,hb_ot_ma 中存在低效算法,而 hb_ot 中存在必要 I/O 操作,无法通过软件代码优化,单以绝对执行时间判定缺陷准确率不高。为辅助开发者检测性能缺陷,研究人员从提高检测效率和提高精确度两个角度展开研究。

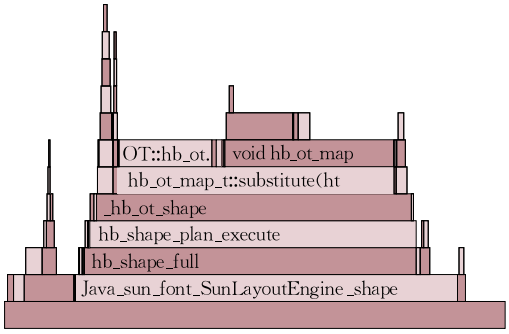

图 9 传统 Profiling 工具原始输出示例

5.1 检测效率提升

Linux 内核和 JVM 虚拟机均提供了性能瓶颈检测工具,如 Perf[69]、YourKit[70] 等,这些工具可检测出给定测试输入下耗时最长的函数;然而,测试输入搜索空间庞大,暴力枚举会产生组合爆炸问题,大大限制了这些方法的有效性。为解决这一挑战,研究人员从不同角度提升检测效率,脉络如图 10 所示。

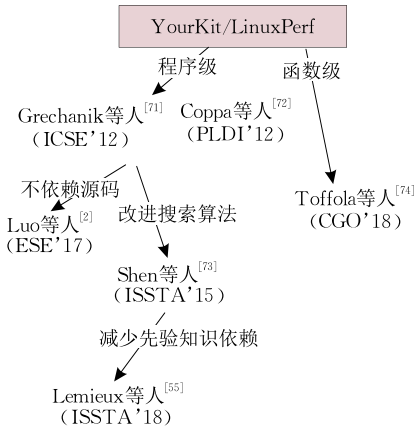

图 10 基于 Profiling 的检测效率提升研究脉络

Grechanik 等人[71]认为导致程序运行缓慢的输入往往具备特定特征,例如在保险费计算软件中,输入一个曾有骗保历史的用户可能会使计算更复杂,触发性能瓶颈。因此提出了一种基于反馈的自学习模型 FOREPOST,根据历史搜索结果优化未来迭代搜索策略,提高检测效率和准确率。在此工作基础上,Luo 等人[2]改进了 FOREPOST 工具,考虑到该工具需要程序源码,但测试人员可能无法获得软件所有源码,该研究利用程序执行 trace 作为回馈信息,不断生成导致程序运行变慢的测试输入。

类似地,Coppa 等人[72]提出了一种基于增长率变化的瓶颈检测工具,搜索使函数执行时间快速增长的输入,有效检测因复杂度过高导致的性能瓶颈。进一步,Shen 等人[73]提出,尽管给定任意输入,Profiling 工具都可以检测出瓶颈,但大型程序逻辑往往十分复杂,给定输入仅能测试到少量代码,可能遗漏大量含有潜在性能瓶颈的代码。因此,该研究组提出了一种基于搜索的瓶颈检测技术 GA-Prof,采用遗传算法搜索使得软件执行时间最长的测试输入,相比 FOREPOST[71],找到相同数量性能瓶颈所消耗的时间更短,效率提升一个数量级。

类似地,Lemieux 等人[55]提出,可利用 fuzzing 技术为性能测试生成测试样例,仿照传统 fuzzing 技术不断探索新程序执行路径的方式,该团队设计实现 PerfFuzz,基于 fuzzing 框架,探索能让某个程序点执行更多次数的输入,从而找到程序瓶颈。然而,不同于 GA-Prof[73] 和 FOREPOST[71],PerfFuzz 无需先验知识定义规则,或者构造高度结构化输入,具有更好的普适性。

Toffola 等人[74]指出,前人工作[71-72]设定了较强假设,即给定的测试输入已经使程序执行到存在瓶颈代码段。然而大型程序的路径存在爆炸问题,盲目地开展瓶颈检测效率非常底下,因此该团队聚焦单个函数的瓶颈检测,设计实现了基于测试生成的检测工具 PerfSyn。给定待测函数,PerfSyn 不断生成使目标函数执行时间增长的测试,并将生成问题转化为搜索问题,利用图搜索算法有效寻找使得目标函数执行缓慢的测试。

5.2 检测精度提升

基于 Profiling 的方法通常会将记录的函数按运行时间降序排列,将执行耗时最长的函数判定为潜在性能缺陷。耗时长可能是因为频繁调用某个函数,或调用了一个返回很慢的函数。然而上述两类情况均无法表明该段代码存在性能瓶颈:后台进程的函数调用可能返回较慢,但并不影响用户性能体验;被频繁调用的某个函数可能只是分散在较长时间段内被多次调用。为此,研究人员引入更多知识从而更加准确判断潜在缺陷代码,如图 11 所示。

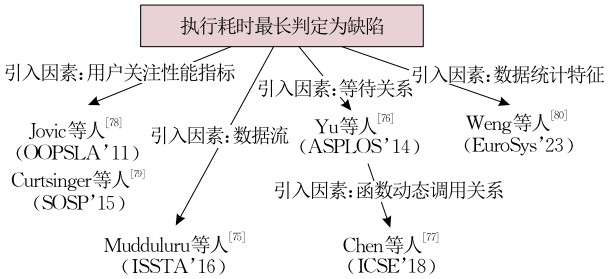

图 11　基于 Profiling 的检测精度提升研究脉络

(1)引入数据流。Mudduluru 等人[75]指出,已有工作大多采用基于 Ball-Larus Profiling 的思想,即以控制流图为基础,统计不同分支的执行次数和时间。然而此类方法的优势在于检测因重复计算(即程序重复进入某条分支)导致的性能瓶颈,然而对内存的低效使用导致的瓶颈往往并不会表现此类特征。因此该研究组考虑程序对象的"use-define"关系,构建混合流图(Hybrid Flow Graph,HFG),并基于此进行 Profiling 检测性能瓶颈。

(2)引入等待关系和调用关系。Yu 等人[76]指出,传统的 Profiling 方法可以定位程序耗时最高的函数,后根据函数调用关系定位候选缺陷代码;锁竞争分析可以找出竞争频繁的代码区域,然而两者无法互相兼顾。为覆盖两种情况,提高基于 Profiling 方法的精度,作者提出可以构建程序的事件等待图(Wait graph,既包含调用关系,又包含竞争等待关系),从而更加全面理解当前程序的性能堵点,定位缺陷代码。Chen 等人[77]同样认为,已有技术使用消耗的资源或代码段执行频率作为 Profiling 的目标过于粗糙,因为真正存在性能缺陷的代码很可能并不存在于执行最慢或最频繁的代码段中。Profiling 技术应把聚焦点从某段代码或某个函数,放宽到多个函数的调用关系及其多次执行的综合 Profiling 结果上。基于此,该研究组涉及了工具 Speedo,通过分析动静态调用关系,综合研判函数的优化收益,更加准确地寻找程序瓶颈。结果显示 Speedo 比已有经典工具 YourKit 的瓶颈定位能力准确 3～4 倍。

(3)引入不同函数对用户关注的性能指标的影响。Jovic 等人[78]提出,瓶颈定位应关注真正会对用户关注的性能指标有影响的函数,而非仅关注函数的执行速度本身。该研究组通过收集软件运行时信息,分析不同函数对常见用户关注性能指标的影响,基于此重新排序潜在的性能瓶颈函数,有效帮助开发人员检测性能瓶颈。Curtsinger 等人[79]认为,无论是应关注影响用户体验的瓶颈[78],还是应综合考虑函数调用关系之间的关系[77],这些方法的都是主观定义了瓶颈定位的关注重点。而该团队采用了另一种较为巧妙的方式,通过给函数手动增加"暂停"来减缓函数的时间,而没有被增加暂停的函数就等同于获得了性能优化。通过这一方式判断"优化"不同函数所获得的全局收益,从而锁定软件中影响最大最的性能瓶颈。

(4)引入数据统计特征。Weng 等人[80]提出,已有的基于 Profiling 的故障定位手段仅关注耗时最长代码,而缺陷还可能存在于相关代码中,但缺乏有效辅助定位信息。基于大量观察,作者指出,在性能故障发生时,某些特定变量的值会呈现某种特征,例如在一个低效循环中,某个循环变量总是取值很大,而未触发故障时该变量值很小。基于此,作者设计了基于数据流辅助的缺陷定位方法,首先观测故障发生和未发生时,关键变量在特定时刻的取值分布差异,然后基于数据流逆推缺陷代码。

5.3　本章小结

已有方法主要生成使程序执行最慢的输入以检测缺陷,该方法将程序输入建模为有限的变量序列,从而将生成问题转化为搜索问题。然而大规模软件的输入往往十分复杂,例如数据库,除用户输入的查询语句外,数据库表的大小、结构和其他表关系都与性能密切相关。因此,当前方法无法针对大规模软件快速生成触发缺陷的输入。另一方面,精度提升方法主要依靠分析程序调用关系推断缺陷可能存在的函数,然而这些信息通常较为宏观,性能缺陷发生原因各异,可能无法准确检测缺陷。未来工作一是可以考虑多维度的输入生成技术,面向大规模软件覆盖更多触发性能缺陷的条件,二是可以考虑增加判断信息,从系统指标、调用关系等方面综合分析检测缺陷函数。

6　基于性能测试的性能缺陷检测

软件测试是检测缺陷的经典方法。针对一般缺陷,已有大量相关研究。然而对于性能缺陷,传统基于测试的方法无法直接迁移,主要原因一是性能缺陷的触发条件更加复杂;二是性能缺陷缺少有效的测试预言。因此研究人员探索基于性能测试的缺陷检测。

性能测试是指为检测在常见负载下软件性能是否达到预期所采用的测试方式,它与压力测试(Stress Testing)和负载测试(Load Testing)具有一

定共性和差异[10]，如图 12 所示。从测试输入角度看，压力测试使用的是超过正常范围的极端负载，而性能与负载测试使用的是正常负载；性能测试输入既可以是模拟用户负载，也可以是为了测试某个具体函数/类而生成的测试样例，但负载测试通常仅使用前者。从测试预言角度看，性能测试只关注性能指标的高低，而其他两类测试更多关注功能是否正常、软件是否会崩溃、是否会出现特定系统异常（内存泄漏），也会在部分特定场景关注性能：例如输入一个超大图片，检测某压缩算法性能。这一过程既是性能测试也是压力测试；再如在网络受限情况下，多用户并发访问时，测试网站延迟是否仍在可接受范围内，这一过程同属于三种测试类型。

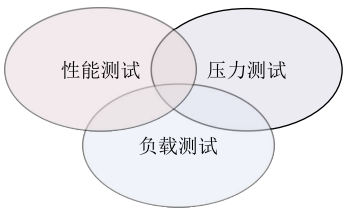

图 12 性能测试、压力测试、负载测试关系图[10]

考虑到上述差异，同时根据前文对性能缺陷的定义，基于性能测试的一般流程可归纳为图 13 所示，与一般软件测试相似，性能测试主要包括测试样例生成和测试预言定义两部分，但软件的性能本身具有一定的不稳定性，还需要通过重复测试缓解不稳定性造成的偏差。本节主要从性能测试样例生成和测试预言挖掘两个角度梳理基于性能测试的缺陷检测，在第 7.5 节中对性能扰动预防进行论述。

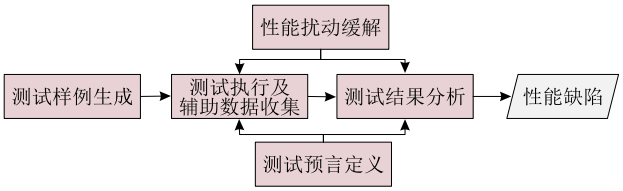

图 13 基于性能测试的一般流程

6.1 面向性能测试的特征调研

软件测试是检测缺陷的经典手段，然而已有工作大多针对功能测试展开研究，研究人员和从业者对性能测试的现状、方法、能力仍知之甚少。面向性能测试的调研工作主要从性能测试样例的生成和测试预言的挖掘两个角度展开。

6.1.1 性能测试测试样例调研

性能测试样例（Test case）是指为检测性能缺陷编写的软件输入，通常包含了测试数据（例如数据库表）、测试输入（例如数据库查询语句）、测试环境（例如数据软件配置、文件系统设置等）。软件负载（Workload）是测试数据和测试输入的统称。

Cotroneo 等人[81]对缺陷触发的条件进行了全面研究，最终归纳出四类典型条件：特定负载、程序状态、运行环境和用户操作，为性能测试的设计设定了框架。Han 等人[82]通过尝试复现两大开源软件中 93 个历史性能缺陷，总结出一系列性能故障复现相关的发现。首先，仅少量故障（18.3%）可被成功复现。作者针对复现失败的故障归纳出 10 类可能的原因，并从复现成功的故障中总结出可提高触发性能缺陷的概率的建议，例如调节特定配置项、提高负载规模等等；该研究还指出，相比于一般性功能故障，性能故障触发难度更高，所需条件更复杂。Cavezza 等人[83]研究了运行环境（内存使用率、硬盘使用率和请求并发度）对 MySQL 故障复现的影响，结果表明，当环境中资源紧俏时（如内存使用率高）触发性能缺陷的成功率可大大提高。

6.1.2 性能测试测试预言调研

测试预言是指执行测试样例后的预期结果的具象化表达，用于判断软件是否通过测试。大量研究[5,7-9]指出软件测试缺乏有效的测试预言是性能缺陷检测的最大挑战。针对测试预言问题，Segura 等人[84-85]探索了用蜕变测试（Metamorphic testing）解决缺乏测试预言问题的可能性，通过实例说明了在特定语义下可通过构建蜕变关系（Metamorphic relation）检测性能缺陷的可能性，例如，预先定义蜕变关系“加载低清图片内存消耗应低于加载高清图片”，然后构建对应测试样例。同时，该研究还针对蜕变关系的通用化提出了一系列挑战，例如蜕变关系的获得依赖领域先验知识等等。Barr 等人[86]对软件测试中的测试预言问题展开综述研究，全面分析了在软件的非崩溃性缺陷检测中，测试预言的构建方式。尽管该研究从理论层面对测试预言构建提供了较强的指导意义，但缺乏对性能缺陷测试预言的启发性研究。

6.2 测试样例构建

（1）配置相关的测试样例生成。Han 等人[34]发现，软件性能缺陷通常需要在特定配置取值下才能触发。基于此，设计实现了面向配置相关性能缺陷的检测工具 PerfLearner。该工具首次将配置作为测试输入纳入性能缺陷检测技术的考虑范围。首先通过对 300 个历史缺陷的特征调研，发现大量（41%）性能缺陷都需要特定配置才能触发；调研还发现，触发性能的缺陷满足局部性原理：历史上曾触发过缺陷

的配置更易再次触发缺陷，且总是某一小部分(5%)配置项更容易触发缺陷。基于以上发现，PerfLearner采用自然语言处理(Natural Language Processing)和信息获取(Information Retrival)技术从历史缺陷报告中提取频繁出现的配置项，辅助已有基于组合测试的缺陷检测工具 GA-Prof [73]更快检测配置相关性能缺陷。Petsios 等人[56]针对特定算法(如排序算法)的性能缺陷检测设计了一种测试输入生成方法 SlowFuzz，采用模糊测试框架，结合运行时监控，不断搜索使得算法执行时间、运行指令等指标增长的测试输入，搜索算法最差情况下的测试输入。然而该方法仅能针对具体算法，无法对复杂软件进行测试输入生成。

(2)循环结构中性能缺陷的测试样例生成。低效循环结构是前人[15,17,47-48]研究的热点问题。Dhok 等人[87]利用了前人设计的测试预言，提出了一种面向循环中重复计算、无用计算等性能缺陷的测试生成方法 Glider。主要通过分析程序中可能存在风险的嵌套循环代码段，识别其中列表类型结构，然后增大其规模以生成单元测试。若不使用前人总结的测试预言，也支持用户构建更加通用的针对循环结构的测试预言。当程序运行时间随输入增大而超线性增长时，软件很可能存在性能缺陷。部分研究以此为测试预言，生成能够触发这一现象的测试样例。Wei 等人[88]首先将测试样例建模为一个元素序列，然后在程序输入所遵循的语法约束条件下，不断变异测试样例，然后通过遗传算法不断筛选能够使得计算图(Recurrent Computation Graph)复杂度增加的测试样例，最终生成一个能够触发出程序最差情况(Worst-case Complexity)的测试样例。

(3)测试样例复用。Stefan 等人[89]分析了近十万个 Java 项目，发现仅极少数(0.37%～3.4%)项目的开发者会编写性能单元测试，且相比于一般的功能单元测试，性能单元测试通常在项目的研发后期才被引入，且通常需要准备较大负载、执行较长时间(大于 4 h)。Leitner 等人[90]调研了 111 个 Java 项目，发现绝大多数项目没有配备专职性能测试开发人员，而项目中仅有少量性能测试，并且设计十分简单，无法覆盖软件大部分代码。同时，开发者通常仅在用户发现性能问题后，才编写针对该性能缺陷的测试代码。针对上述问题，Ding 等人[91]提出可以复用软件功能测试检测性能缺陷，通过调研云服务框架软件中，共计 127 个性能缺陷和单元测试发现，80.3%的缺陷都可以被至少一个单元测试覆盖。然而，对于每个性能缺陷，却仅有少部分(9.2%～20.6%)单元测试可以测出缺陷修复前后的差异，原因在于，其余单元测试并未覆盖到缺陷代码。针对这一问题，可以通过测代码的优先级排序或者测试选择/排序(Test case selection/prioritization)提高性能缺陷检测的效率。对于前者，Huang 等人[92]指出，在演化过程中只有部分代码更新需要重新执行性能测试。因此该团队通过调研引入历史性能缺陷的代码变更特征(上下文、变更修改对象等)总结出一系列静态规则，自动筛选被测优先级更高的代码变更。从另一方面，Mostafa 等人[93]指出，筛选应聚焦性能测试样例而非代码变更。基于此，该团队针对每个版本的软件，构建不同测试样例对软件性能的影响模型，评估代码变更可能对性能影响的程度，从而筛选更可能触发性能缺陷的测试样例。

6.3　测试预言挖掘

(1)基于蜕变测试。蜕变测试(Metamorphic Testing)原是在功能测试中，缺乏测试预言时采用的技术。例如验证 $\sin(x)$ 是否计算准确可通过与 $\cos(x-\pi/2)$ 计算结果比对得知。在性能测试领域也可类似构建这种蜕变关系实现缺陷检测。Liu 等人[19]针对数据库软件，提出可以借用蜕变测试的思想，构建语义等价的查询语句，当观测到查询语句的延时出现差异时，可能存在性能缺陷。基于此，该团队开发了性能缺陷检测工具 AMOEBA，通过语法结构(例如改变 WHERE 和 GROUP BY 的运算顺序)和语法表达式(例如用“IN”运算代替“<”运算)两个层面的变异构建语义相同的查询语句，同时引导变异向能使两查询语句性能产生差异的方向进行。Ba 等人[94]针对性能测试耗时长、缺乏测试预言两大挑战，从数据库入手，构建“条件更严格的查询语句相比于条件更放松的查询语句，不应返回更大基数的查询结果”这一基本测试预言，然后巧用数据库“EXPLAN”机制，避免执行大量实际测试导致的测试开销，有效检测出大量历史未知性能缺陷。Johnston 等人[95]受 Segura 等人[84-85]的启发，进一步探索了基于蜕变关系的性能缺陷检测，遗憾的是，缺乏系统性的设计和充分的评估。Nolasco 等人[96]针对蜕变关系提取自动化程度低的问题，提出了一种针对特定类型蜕变关系提取方法，可自动检测语义相同但顺序不同的方法调用序列。当存在缺陷时，两个序列的执行时间可能存在较大差异。以此可构建性能测试预言。

(2)基于软件配置。软件配置作为控制资源适

配环境的重要接口，与性能关系密切，通过挖掘配置与性能的关系可构建测试预言。此类研究有两种思路：一是通过建模软件正常运行的性能，间接辅助性能异常检测；二是建模软件存在性能故障时性能与配置的关系，直接进行性能缺陷检测。

第一类研究中，Chen 等人[97]提出了用符号执行等技术建立了工作负载与性能之间的分布关系，全面刻画软件性能特征，典型特征例如程序的最佳情况和最坏情况执行时间，并利用这种分布关系可视化性能缺陷特征，为开发者理解软件运行正常和有性能故障时的差异提供了指导。Li 等人[98]归纳了 Java 软件中，配置的调整对软件性能的影响模式，例如随着配置值的增加，函数执行时间线性增加等。并设计实现自动化工具 LearnConf 分析配置相关的代码模式，推断配置对性能的影响模式，帮助开发者更好的理解软件正常运行时，性能和配置的关系。针对软件配置对性能的影响，Velez 等人[99]利用白盒分析，建模软件在不同负载下，配置对软件局部的性能影响。相比于大量已有的基于黑盒配置-性能建模的方法，该方法不仅具有更好的可解释性，也具有更高的准确率。该研究可以帮助开发者更好的理解软件的性能。例如理解何处耗时高以及为何某个操作（如加密）耗时较高。

第二类方法中，He 等人[100]提出，软件配置作为控制资源适配环境的重要接口，与性能关系密切。而已有研究[7,101]表明，在服务器端软件中，超半数的性能缺陷都与配置相关。因此作者对配置相关的性能缺陷展开调研，发现配置调节具有一定预期，而调节后性能变化与预期的不一致可以作为性能缺陷检测的测试预言。因此作者从软件说明和文档中自动提取配置调节预期，并基于性能测试检测配置相关的性能问题。

（3）基于编程最佳实践。不同软件虽然功能各异，但许多业务逻辑十分相似，例如格式转换、数据缓存、内存动态申请与释放等等。在大量的编程实践中，开发人员逐渐形成了一些性能最佳编程实践，并总结出违反最佳实践会导致性能故障的典型场景。Trubiani 等人[102]基于当前较为公认的 7 条最佳编程实践，在 Java 程序中具象化了违反最佳实践的方式，例如一种违反最佳实践的方式是“某任务长时间独占内核，是程序运行的主要瓶颈”，在该研究中被具象化为“大量线程（被阻塞线程除以总线程数达到一定比例）被阻塞，且程序的最慢函数执行时间、处于关联路径中的所有函数平均执行时间等要素之间满足一定关系”。基于此构建测试预言，设计自动化检测工具 JPAD，并使用 Load test 作为测试样例，在 Java 软件中检测出大量缺陷。值得注意的是，该研究所使用的最佳实践来自于 1998～2003 年的研究成果，但仍检测出了许多历史未知的性能缺陷，这证明了当前软件开发过程中，仍旧缺乏自动化工具辅助开发者避免引入性能问题。

6.4 本章小结

尽管开发人员为功能测试编写大量测试（单元测试、系统测试等），但极少软件有专用的性能测试集。未来研究可重点关注为软件生成性能测试。另一方面，已有方法虽已针对测试预言问题展开了研究，但都仅能针对特定类型软件或在特定场景下设计测试预言，例如针对低效循环缺陷，可通过匹配特定模式检测；针对配置相关的性能缺陷，可匹配配置调节的性能预期检测缺陷。然而，当前仍缺少普适性的方法，性能缺陷检测的最大挑战仍未有效解决，未来研究可在性能测试方面展开研究。

7 共性问题及对策

基于对上述工作的分析，本章主要梳理在性能缺陷检测中，研究人员遇到的共性问题及应对措施，主要从性能测试样例的选择、缺陷检测能力的评估和性能扰动的处理三个方面阐述。

7.1 性能测试样例

性能测试样例生成。基于动态手段的性能缺陷检测需要输入触发缺陷。图 14 展示了表 2 中研究所采用的四种输入来源。其中，绝大部分（83.1%）的研究工作都是直接使用现有测试，包括软件自带的功能测试或者较为流行的第三方测试基准，少数研究工作会为性能测试自动生成测试输入。

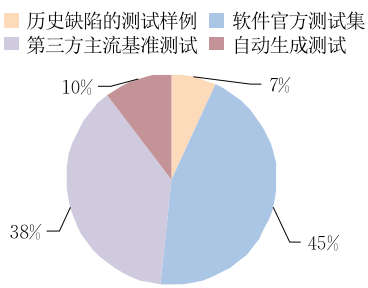

图 14 动态性能缺陷检测测试样例生成手段

对于通用性、基础性较强的软件，如数据库、编译器、科学计算库等，一般都具备较为成熟的基准测试，这些基准测试工具包含了压力测试工具（如

Apache Benchmark)、负载测试工具(如 sysbench-tpcc、ycsb)和性能测试工具(如 SPEC CPU 系列)，还有近期研究为 python 构建的、可对标 SPEC CPU 数据集[103]，这些都为检测软件的性能缺陷提供了很好的基础。尽管如此，这些工具通常只能产生最典型的测试输入，例如 sysbench-tpcc 所覆盖的数据库结构和查询语法仅仅是冰山一角。已有大量研究表明，性能缺陷的触发需要更为特定的输入，因此也有许多团队探索将功能测试中的单元测试集或系统测试集改装为性能测试。对于非基础软件，通常没有通用的基准测试集，软件自身也极少为性能测试编写测试样例。因此，未来研究可关注性能测试，特别是应用软件的测试样例生成。

7.2　评测指标

性能缺陷检测研究通常以检测出的缺陷数量评估工具的能力，又可细分为检测出的历史已知缺陷和历史未知缺陷数量。能够检出后者说明了工具的检测能力。表 2 展示了性能缺陷检测工具在评估中检测出的性能缺陷数量。注意到，不同工具区分性能缺陷的粒度不同，例如部分工具将一处代码更改算作一个缺陷，或将整个程序整体优化算作一个缺陷，因此绝对数值的差异不具有指导意义。

绝大部分工具都检测出了历史未知的缺陷，个别工作虽仅在历史已知缺陷上进行了验证，但均从工具效率和对比上说明了相较前人工作的先进性。注意到，表 2 中大量工作使用了历史已知缺陷评估检测工具，这些工作选取已知缺陷的方式主要分为三类：(1) 在开源库(软件官方缺陷库、开源论坛)中搜索、确认历史缺陷。此类方法可确保缺陷的真实性和样本的广泛性，但耗时长；(2) 从同类研究论文中沿用历史缺陷。此类方法更加方便，且便于与已有方法展开对比，但广泛性不足；(3) 人工注入缺陷。此类方法仅使用与缺陷模式较为固定的软件，通用性、真实性较低。为平衡充分性、真实性、人工开销等因素，部分研究采用“1＋2”式样本选择方法，本文归纳了已有研究收集历史已知性能缺陷的普遍方法，可供后续研究参考。

表 2　不同性能缺陷检测工具测出的性能缺陷

研究工作	工具名称	历史已知缺陷	缺陷选取方式	历史未知缺陷	目标软件
FSE'13[47]	Cachetor	/	/	14	大型 Java 软件
ICSE'13[15]	Toddler	11	开源库	42	中大型 Java 软件
OOPSLA'14[54]	EventBreak	2	开源库	4	中大型 Web 软件
ICSE'14[57]	—	/	/	20	ORM 软件
ICSE'15[17]	Caramel	/	/	150	大型基础软件
PLDI'15[20]	CLARITY	20	开源库	72	Collection 库
ISSTA'16[50]	SyncProf	13	开源库	—	大型基础软件
FSE'16[87]	GLIDER	12	开源库	34	中型 Java 软件
ASPLOS'17[43]	REDSPY	/	/	3	小型基准测试程序
CCS'17[56]	SlowFuzz	/	/	9	小型程序
ASE'18[34]	PerfLearner	7	开源库＋复用	—	C/C++大型软件
EuroSys'18[33]	PCatch	15	开源库	7	Java 大型服务系统
FSE'19[45]	JXPerf	31	开源库	6	Java 小型软件
ASE'20[100]	CP-Detector	43	开源库＋复用	13	C/C++大型软件
ISSTA'20[18]	DPFuzz	/	/	4	机器学习库
ASE'21[63]	MDPerfFuzz	/	/	216	MarkDown 编译器
ICSE'22[19]	AMOEBA	/	/	39	数据库
FSE'22[37]	DeepPerf	/	/	105	深度学习软件
TSE'23[102]	JPAD	90	开源库＋复用	4	Java 大型服务系统
ICSE'24[94]	CERT	/	/	13	数据库

(i) 目标软件自身的缺陷仓库。部分软件使用自身开发的仓库如 MySQL、H2 等。

(ii) 通用缺陷管理系统，如 JIRA(MongoDB、Hadoop 等)、Bugzilla(httpd、gcc 等)、GitHub Issue List、Launchpad 等。

(iii) 开源问答社区、博客，如 StackOverflow、Docker forum 等。

当前，性能缺陷检测技术主要针对中小型软件，对于大型软件，尤其是 C/C++大型软件的检测技术仍有较大研究空间。其存在的主要挑战是：(1) 主流方法需要对程序数据/控制流进行精准构建，而 C/C++程序语言特性复杂，相关研究仍存在较多空白；(2) 已有方法总结的低效模式虽具有典型性，但无法充分适应大规模软件的代码复杂度，当前研

究尚以验证思路、探索新手段为主。

7.3 不同性能检测方法对比

本文梳理了三种主要流派的性能缺陷检测方法。其中，基于特定模式的检测方法最为主流，此类方法的准确率相对较高且一般不需要进行大量的盲目测试，因此效率也较高。但针对特定模式的方法理论上仅能检测该模式的缺陷。尽管如此，基于特定模式的方法可以从已知模式直接推导出修复的策略，缩短了传统的“检测-诊断-修复”三个环节的流程。

相比之下，基于测试的方法在测试上消耗更多的时间以覆盖到更多代码。同时，此类方法依赖于有效的测试预言，而目前仍缺少较为通用的测试预言。此外，相比于基于模式的方法，此类方法的输出是触发缺陷的测试样例，因此可能需要开展额外的故障诊断工作来定位缺陷代码行。这类方法的优势是不会局限于某一特定缺陷类型，覆盖面相比基于模式的方法更大。

传统的基于 Profiling 的方法既没有模式作为先验信息，也没有测试预言指明测试方向，导致在检测的准确性和普适性方面均不如前两类方法。但基于 Profiling 的方法的灵活性更强，通常可以通过调整参数从不同层面（语句级、函数级、类级等）理解性能瓶颈，对于经验丰富的测试人员，可以在此类方法的辅助下，快速定位性能缺陷。

7.4 编程语言对软件性能缺陷检测的影响

从性能缺陷特征看，不同编程语言的软件中，性能缺陷存在与语言特性相关的特有特征，但性能缺陷检测方法万变不离其宗。

性能缺陷特征上，存在编程语言无关的一般特征。例如，低效循环模式可能会出现在任何一种支持循环结构的编程语言中，Toddler[15] 就针对相同缺陷模式设计了 C/C++ 和 Java 两种版本的缺陷检测工具。另一方面，编程语言特性也会带来不同的缺陷特征。例如，Java 多使用数据封装库（Collection）对大规模数据进行基本操作，而一旦封装库的实现不够高效，便会带来性能故障[20]。此外，Java 的 JVM 特性也会导致其在翻译指令时引入性能缺陷[45]。ORM 软件是使用 Ruby 语言实例化对象，完成关系型数据库操作的一类软件。众所周知，数据库操作不当极易产生低效查询，引发性能问题，因此 Ruby 代码如出现低效查询则会导致性能缺陷[57]。Markdown 编译器经常使用回溯算法实现其语言功能，因此恶意构造的Markdown代码很容易引起编译器发生严重性能故障[63]。深度学习库多用 python 实现，如在模型训练中没有使用适当的加速机制，则会造成性能缺陷，在 python 代码中，则可能表现为特定的低效 API 调用序列[37]。

尽管各种语言编写的软件性能缺陷具有各自特征，但本质上都是因为进行了低效的计算。检测性能缺陷的关键是找准低效计算的载体。例如前文所述情况中，Java 中的循环体、Markdown 中的递归调用、Ruby 中的查询语句结构等等。由于其载体的不同，可能需要从不同角度观测性能缺陷发生时的特征，并采用不同的技术手段在执行测试时采集特征数据，然后制定不同的规则来判定缺陷。因此，对于未来研究，如需检测不同语言编写的程序中的性能缺陷，应当首先研究可能存在大量计算的语言特性，继而在检测方法、判定规则等问题上展开详细设计。

7.5 性能扰动与测试运行环境

性能测试的稳定性通常会被环境因素影响，例如系统预热、垃圾回收等；即使在完全相同的环境中重复执行相同测试，也可能会产生不同的结果。因此，在性能测试中，为获得更为准确的性能值，通常需要重复多次测试，从而获得稳定结果。然而性能测试本身通常耗时较多，多次重复的开销更是成倍增长。当前研究工作通常是采用经验数值或者避开这一问题。实际上目前已有较为公认的工具可以借鉴：可分为离线方法和在线方法，离线方法主要是在固定环境（I/O、内存、网络等）中通过工具 CONFIRM[104] 对不同性能指标的稳定性进行一次性的评估，根据统计学原理，确定合理的重复测试次数；在线方法则是在不确定环境中，通过工具 PT4Cloud[105]，或者其扩展版 Laaber[106] 在性能测试进行的过程中，不断评估已测得的性能指标，判断其是否已达到统计学意义上的稳定。

另一方面，测试执行的环境也与测试的结果密切相关。例如，对于存储密集型软件，在不同的文件系统日志级别下，性能的差异十分显著。Wu 等人[107] 研究了在不同数据库引擎和系统垃圾回收策略状态下，数据库的性能差异。Wang 等人[108] 也发现，基准测试工具的多种参数设置、操作系统开关、硬件设置等均会对实验结果产生影响。因此，在开展相关研究时，如具备详细分析上述环境设置对结果影响能力的条件，则应当进行相应分析。否则，需根据目标软件的典型应用场景确定环境设置，确保实验结果的代表性。

8　未来研究方向

尽管大量研究在性能缺陷检测上取得了突破，但仍未解决性能缺陷检测的核心问题——缺乏测试预言。同时，随着人工智能软件的发展和普及，已有的检测手段也无法很好检测智能软件的性能缺陷。基于此，本文展望未来研究，提出 7 点潜在研究方向。

(1) 面向复杂软件的测试输入生成

当前，已有大量工作针对规则化输入软件(如正则表达式解析引擎、bzip2 等小型工具程序)设计面向性能测试的测试输入生成技术。规则化程序的输入生成问题可转化成一个较为有限搜索空间的搜索问题。但对于复杂软件，如数据库、Web 服务器等，测试输入还包含库表、网页内容等等复杂负载。传统的搜索算法或遗传算法无法有效生成性能测试输入。已有研究表明，复杂软件的现有功能测试具备一定检测性能缺陷能力，但存在规模不够、测试输入要素不全(如未考虑软件配置)等问题。未来研究可探索基于现有功能测试改造研究，填补复杂软件性能测试生成的空白。

(2) 面向机器/深度学习程序的性能缺陷检测

随着人工智能技术的发展与普及，大量软件引入智能算法、智能模块。智能程序的性能也愈发关键，例如在自动驾驶汽车中，智能算法必须对突发情况在极短时间内做出正确反映。然而，当前大量工作关注智能程序的正确性问题，并开发了如基于神经元覆盖率的测试技术，却仅有少量工作关注智能程序、智能赋能的传统程序的性能问题。智能程序中性能问题的表现方式、根因、触发方法更是鲜有研究，因此未来工作可针对该问题展开系统性的研究。

(3) 性能缺陷注入

针对传统缺陷，已有大量研究通过注入故障研究软件测试的充分性、软件自身的容错机制、软件缺陷检测技术有效性等等。然而当前的故障注入主要针对一般缺陷(导致软件崩溃或功能异常)，缺乏对性能缺陷的注入。在性能缺陷检测相关研究中，研究人员通常需要在目标软件历史缺陷库中寻找足够数量的历史缺陷才能评估方法的有效性，然而历史记录的性能缺陷通常较少，且多用于缺陷特征总结，因此搜集足量历史性能缺陷成为制约相关研究评估工作的一大短板。在程序中简单插入延时函数对性能缺陷的模拟十分有限，Chen 等人[109] 采用变异测试(Mutation Testing)的方式向程序中注入特定代码模式的缺陷(如在循环中加入产生无用结果的计算等)来检测现有缺陷检测工具的鲁棒性。未来研究可考虑利用软件代码语义等效替代、系统环境干预等方式，探索更加丰富的性能缺陷注入手段。

(4) 基于蜕变测试的性能缺陷检测

基于性能测试的缺陷检测的最大挑战之一是缺乏有效的测试预言。解决测试预言的两类典型方法：一是差分测试(Differential Testing)，即在两个软件版本中运行同一测试样例；二是蜕变测试(Metamorphic Testing)，即在某一版本软件中运行一组语义关系已知的测试样例。对于前者，已有大量关于性能回归问题(Performance regression)的研究，而后者仍不够成熟。主要挑战在于难以挖掘较为通用的测试间的语义关系。Liu 等人[19] 开创了一种思想：构建语义等价的一组输入，当观测其性能出现差异时，则认定存在缺陷。未来工作可进一步探索在不同领域如何有效构建语义等价的测试输入。

(5) 基于软件配置的性能扰动缓解

性能扰动产生的根因大致可分为硬件的随机性和操作系统的随机性。其中操作系统的随机性来源十分广泛：锁竞争、系统缓存、系统调度等等。已有工作一般采用重复测试的方法，用统计学方法来消除随机性。然而重复测试无法避免地带来更大测试开销。硬件随机性通常无法通过软件手段探测和避免，但操作系统带来的随机性存在消除的可能性。软件配置作为软件和操作系统交互的关键接口，可能成为消除随机的抓手。未来工作可建立配置和系统行为之间的关系，然后利用配置合理消除系统带来的随机性，同时兼顾测试的覆盖性。

(6) 性能缺陷自动修复

针对一般缺陷，已有大量自动修复方法。然而针对性能缺陷，代码没有逻辑错误，且大量证据表明，其缺陷代码通常范围更大。因此已有自动修复方法无法直接应用于性能缺陷修复。特定类型的性能缺陷通常具有一定低效代码模式特征，因此未来研究可基于特定模式研究性能缺陷自动修复，填补相关研究的空白。

(7) 大模型技术赋能性能缺陷检测

大模型(LLM)技术也在软件缺陷检测领域展现出了应用的前景。Wang 等人[110] 分析了近年来共计 102 篇大语言模型在软件测试领域的应用的文章，从软件测试样例生成、测试预言生成、缺陷定位、

程序自动修复、程序分析、变异测试等多个方面阐述了大模型的应用方法。经分析发现，尽管 LLM 技术已经取得了显著的成效，但仍存在较大挑战。首先，当前 LLM 技术并未应用到性能缺陷检测领域，这是由于大部分 LLM 技术主要是通过把软件测试问题转化为代码静态生成或者自然语言生成问题，从而解决大部分正确性、功能性问题。然而性能缺陷代码一般无正确性问题，而是在在动态运行中某些低效的模式才被显现出来，LLM 目前无法有效认知这些高级知识。未来，能够准确建模的性能缺陷有可能最先被 LLM 技术解决，例如超线性复杂度代码段等。此外，LLM 技术还可应用与性能缺陷检测辅助性工作中，例如历史性能缺陷复现、大规模负载生成等等。除上述研究点外，未来研究可利用 LLM 理解程序语义，探究性能测试预言生成和性能蜕变测试生成等问题。

9 总 结

一直以来，软件性能缺陷给软件供应商带来了大量损失。本文对近 10 年来的性能缺陷检测技术进行了梳理，提出了当前技术的主要技术路线，分析了相关研究的共性问题及对策，归纳了各类方法的异同和优劣，展望了相关研究未来的方向。当前，性能缺陷检测技术还面临一系列挑战，例如通用性不足、效率底下等。此外，新的软件形态，如智能合约、深度学习、国产化软件等也需要全新的性能缺陷检测技术。随着软件应用范围的不断扩大以及用户对软件性能要求的提高，性能缺陷检测在软件开发、维护过程将会发挥更加重要的作用，未来应加强对相关问题的研究。

参 考 文 献

[1] Ministry of Industry and Information Technology, National Development and Reform Commission. Information Industry Development Guide. https://wap.miit.gov.cn/ztzl/lszt/zgzz-2025/wjfb/art/2020/art_37238801bda24a08ab5027ae9b7c46cc.html, 2016(in Chinese)
(工业和信息化部，国家发展改革委. 信息产业发展指南. https://wap.miit.gov.cn/ztzl/lszt/zgzz2025/wjfb/art/2020/art_37238801bda24a08ab5027ae9b7c46cc.html, 2016)

[2] Luo Q, Nair A, Grechanik M, et al. FOREPOST: Finding performance problems automatically with feedback-directed learning software testing. Empirical Software Engineering (ESE), 2017, 22: 6-56

[3] Tanli Meng-Qing, Zhang Ying, Wang Yulin. System testing based on software performance. Software, 2020, 41(11): 1-5(in Chinese)
(谭李孟清，张莹，王玉林. 基于软件性能的系统测试. 软件，2020，41(11)：1-5)

[4] Amazon Found Every 100 ms of Latency Cost them 1% in Sales. https://www.gigaspaces.com/blog/amazon-found-every-100 ms-of-latency-cost-them－1-in-sales

[5] Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs//Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI). Beijing, China, 2012, 47(6): 77-88

[6] Iqbal M S, Krishna R, Javidian M A, et al. CADET: A systematic method for debugging misconfigurations using counterfactual reasoning//Proceedings of the 34th Conference on Neural Information Processing Systems(NeurIPS). Vancouver, Canada, 2023: 1-12

[7] Han X, Yu T. An empirical study on performance bugs for highly configurable software systems//Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement(ESEM). Ciudad Real, Spain, 2016, 23: 1-10

[8] Yan C, Cheung A, Yang J, et al. Understanding database performance inefficiencies in real-world web applications//Proceedings of the 2017 ACM on Conference on Information and Knowledge Management(CIKM). Singapore, 2017: 1299-1308

[9] Yang J, Subramaniam P, Lu S, et al. How not to structure your database-backed web applications: A study of performance bugs in the wild//Proceedings of the 40th International Conference on Software Engineering(ICSE). Gothenburg, Sweden, 2018: 800-810

[10] Jiang Z M, Hassan A E. A survey on load testing of large-scale software systems. IEEE Transactions on Software Engineering(TSE), 2015, 41(11): 1091-1118

[11] Hort M, Kechagia M, Sarro F, et al. A survey of performance optimization for mobile applications. IEEE Transactions on Software Engineering(TSE), 2021, 48(8): 2879-2904

[12] Kang Y, Zhou Y, Xu H, et al. DiagDroid: Android performance diagnosis via anatomizing asynchronous executions//Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE). Seattle, USA, 2016: 410-421

[13] Liu Y, Xu C, Cheung S C. Characterizing and detecting performance bugs for smartphone applications//Proceedings of the 36th International Conference on Software Engineering (ICSE). Hyderabad, India, 2014: 1013-1024

[14] Turner M, Kitchenham B, Brereton P, et al. Does the technology acceptance model predict actual use? A systematic literature review. Information and Software Technology(IST), 2010, 52(5): 463-479

[15] Nistor A, Song L, Marinov D, et al. Toddler: Detecting performance problems via similar memory-access patterns//Proceedings of the 35th International Conference on Software Engineering(ICSE). San Francisco, USA, 2013: 562-571

[16] Song L, Lu S. Performance diagnosis for inefficient loops//Proceedings of the International Conference on Software Engineering (ICSE). Buenos Aires, Argentina, 2017: 370-380

[17] Nistor A, Chang P C, Radoi C, et al. Caramel: Detecting and fixing performance problems that have non-intrusive fixes//Proceedings of the 37th International Conference on Software Engineering (ICSE). Austin, USA, 2015: 902-912

[18] Tizpaz-Niari S, Černý P, Trivedi A. Detecting and understanding real-world differential performance bugs in machine learning libraries//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). Virtual, 2020: 189-199

[19] Liu X, Zhou Q, Arulraj J, et al. Automatic detection of performance bugs in database systems using equivalent queries//Proceedings of the 44th International Conference on Software Engineering (ICSE). Pittsburgh, USA, 2022: 225-236

[20] Olivo O, Dillig I, Lin C. Static detection of asymptotic performance bugs in collection traversals//Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Portland, USA, 2015: 369-378

[21] GCC #27733. Large compile time regression. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=27733

[22] MySQL #21727. sort_buffer_size has an extremely negative impact on a query as it increases. https://bugs.mysql.com/bug.php?id=21727

[23] Nair V, Menzies T, Siegmund N, et al. Using bad learners to find good configurations//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE). Paderborn, Germany, 2017: 257-267

[24] Chen Z, Chen P, Wang P, et al. DiagConfig: Configuration diagnosis of performance violations in configurable software systems//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE). San Francisco, USA, 2023: 566-578

[25] Lepers B, Balmau O, Gupta K, et al. KVell: the design and implementation of a fast persistent key-value store//Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP). Shanghai, China, 2019: 447-461

[26] He H, Xu E, Li S, et al. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. Proceedings of the VLDB Endowment, 2023, 16(7): 1712-1725

[27] Shang W, Hassan A E, Nasser M, et al. Automated detection of performance regressions using regression models on clustered performance counters//Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE). Austin, USA, 2015: 15-26

[28] Chen J, Shang W, Shihab E. PerfJIT: Test-level just-in-time prediction for performance regression introducing commits. IEEE Transactions on Software Engineering (TSE), 2020, 48(5): 1529-1544

[29] Dai T, He J, Gu X, et al. Understanding real-world timeout problems in cloud server systems//Proceedings of the 8th International Conference on Cloud Engineering (IC2E). Orlando, USA, 2018: 1-11

[30] Zhang Chen. Research on Understanding and Detecting Performance Bug in Distributed Systems[M. S. dissertation]. National University of Defense Technology, Changsha, 2019 (in Chinese)
(张晨. 分布式系统中性能缺陷的机理分析及检测技术研究[硕士学位论文]. 国防科技大学, 长沙, 2019)

[31] Huang P, Guo C, Zhou L, et al. Gray failure: The achilles' heel of cloud-scale systems//Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS). Whistler, Canada, 2017: 150-155

[32] Li J, Zhang Y, Lu S, et al. Performance bug analysis and detection for distributed storage and computing systems. ACM Transactions on Storage (TOS), 2023, 19(3): 1-33

[33] Li J, Chen Y, Liu H, et al. PCatch: Automatically detecting performance cascading bugs in cloud systems//Proceedings of the 13th EuroSys Conference (EuroSys). Porto, Portugal, 2018: 1-14

[34] Han X, Yu T, Lo D. PerfLearner: Learning from bug reports to understand and generate performance test frames//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE). Corum, Montpellier, France, 2018: 17-28

[35] Zhao Y, Xiao L, Bondi A B, et al. A large-scale empirical study of real-life performance issues in open source projects. IEEE Transactions on Software Engineering (TSE), 2022, 49(2): 924-946

[36] Chen Y, Winter S, Suri N. Inferring performance bug patterns from developer commits//Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE). Berlin, Germany, 2019: 70-81

[37] Cao J, Chen B, Sun C, et al. Understanding performance problems in deep learning systems//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE). Singapore, 2022: 357-369

[38] Selakovic M, Pradel M. Performance issues and optimizations in JavaScript: An empirical study//Proceedings of the 38th International Conference on Software Engineering (ICSE). Austin Texas, USA, 2016: 61-72

[39] Song L, Lu S. Statistical debugging for real-world performance problems//Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Portland Oregon, USA, 2014: 561-578

[40] Gong Z, Chen Z, Szaday J, et al. An empirical study of the effect of source-level loop transformations on compiler stability//Proceedings of the ACM on Programming Languages(OOPSLA). Boston, USA, 2018: 1-29

[41] Theodoridis T, Rigger M, Su Z. Finding missed optimizations through the lens of dead code elimination//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Lausanne, Switzerland, 2022: 697-709

[42] Chabbi M, Mellor-Crummey J. DeadSpy: A tool to pinpoint program inefficiencies//Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO). San Jose, USA, 2012: 124-134

[43] Wen S, Chabbi M, Liu X. REDSPY: Exploring value locality in software//Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS). Xi'an, China, 2017: 47-61

[44] Su P, Wen S, Yang H, et al. Redundant loads: A software inefficiency indicator//Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering(ICSE). Montreal, Canada, 2019: 982-993

[45] Su P, Wang Q, Chabbi M, et al. Pinpointing performance inefficiencies in Java//Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE). Tallinn, Estonia, 2019: 818-829

[46] Mertz J, Nunes I. A qualitative study of application-level caching. IEEE Transactions on Software Engineering(TSE), 2016, 43(9): 798-816

[47] Nguyen K, Xu G. Cachetor: Detecting cacheable data to remove bloat//Proceedings of the 9th Joint Meeting on Foundations of Software Engineering(FSE). Saint Petersburg, Russia, 2013: 268-278

[48] Della Toffola L, Pradel M, Gross T R. Performance problems you can fix: A dynamic analysis of memoization opportunities //Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA). Pittsburgh, USA, 2015, 50(10): 607-622

[49] Jin Y, Wang H, Zhong R, et al. PerFlow: A domain specific framework for automatic performance analysis of parallel applications//Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). Seoul, Republic of Korea, 2022: 177-191

[50] Yu T, Pradel M. SyncProf: Detecting, localizing, and optimizing synchronization bottlenecks//Proceedings of the 25th International Symposium on Software Testing and Analysis(ISSTA). Baltimore, USA, 2016: 389-400

[51] Yu T, Pradel M. Pinpointing and repairing performance bottlenecks in concurrent programs. Empirical Software Engineering(ESE), 2018, 23: 3034-3071

[52] Zhou F, Gan Y, Ma S, et al. wPerf: Generic off-CPU analysis to identify bottleneck waiting events//Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation(OSDI). Carlsbad, USA, 2018: 527-543

[53] Hu Y, Huang G, Huang P. Pushing performance isolation boundaries into application with pBox//Proceedings of the 29th Symposium on Operating Systems Principles (SOSP). Koblenz, Germany, 2023: 247-263

[54] Pradel M, Schuh P, Necula G, et al. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation//Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA). Portland, USA, 2014, 49(10): 33-47

[55] Lemieux C, Padhye R, Sen K, et al. PerfFuzz: Automatically generating pathological inputs//Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA). Amsterdam, The Netherlands, 2018: 254-265

[56] Petsios T, Zhao J, Keromytis A D, et al. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security(CCS). Dallas, USA, 2017: 2155-2168

[57] Chen T H, Shang W, Jiang Z M, et al. Detecting performance anti-patterns for applications developed using object-relational mapping//Proceedings of the 36th International Conference on Software Engineering(ICSE). Hyderabad, India, 2014: 1001-1012

[58] Chen T H, Shang W, Jiang Z M, et al. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. IEEE Transactions on Software Engineering(TSE), 2016, 42(12): 1148-1161

[59] Shao S, Qiu Z, Yu X, et al. Database-access performance antipatterns in database-backed web applications//Proceedings of the 36th International Conference on Software Maintenance and Evolution(ICSME). Adelaide, Australia, 2020: 58-69

[60] Yang J, Subramaniam P, Lu S, et al. How not to structure your database-backed web applications: A study of performance bugs in the wild//Proceedings of the 40th International Conference on Software Engineering (ICSE). Gothenburg, Sweden, 2018: 800-810

[61] Chen B, Jiang Z M, Matos P, et al. An industrial experience report on performance-aware refactoring on a database-centric web application//Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). San Diego, USA, 2019: 653-664

[62] Wang Rui. Performance Issue Identification and Diagnosis Method of SaaS Software Runtime Based on Log [Ph. D. dissertation]. Wuhan University, Wuhan, 2019(in Chinese)

(王蕊. 基于日志的 SaaS 软件运行时性能问题的识别与诊断方法[博士学位论文]. 武汉大学，武汉，2019)

[63] Li P, Liu Y, Meng W. Understanding and detecting performance bugs in markdown compilers//Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). Melbourne, Australia, 2021: 892-904

[64] Nusrat F, Hassan F, Zhong H, et al. How developers optimize virtual reality applications: A study of optimization commits in open source unity projects//Proceedings of the 43rd International Conference on Software Engineering (ICSE). Madrid, Spain, 2021: 473-485

[65] Wert A, Happe J, Happe L. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments//Proceedings of the 35th International Conference on Software Engineering (ICSE). San Francisco, USA, 2013: 552-561

[66] He J, Lin Y, Gu X, et al. PerfSig: Extracting performance bug signatures via multi-modality causal analysis//Proceedings of the 44th International Conference on Software Engineering (ICSE). Pittsburgh, USA, 2022: 1669-1680

[67] Dean D J, Nguyen H, Gu X, et al. PerfScope: Practical online server performance bug inference in production cloud computing infrastructures//Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC). Seattle, USA, 2014: 1-13

[68] Han S, Dang Y, Ge S, et al. Performance debugging in the large via mining millions of stack traces//Proceedings of the 34th International Conference on Software Engineering (ICSE). Zurich, Switzerland, 2012: 145-155

[69] Linux Perf Tool. https://perf.wiki.kernel.org/index.php/Main_Page, 2015

[70] GmbH Y. The Industry Leader in .NET & Java Profiling. https://www.yourkit.com

[71] Grechanik M, Fu C, Xie Q. Automatically finding performance problems with feedback-directed learning software testing//Proceedings of the 34th International Conference on Software Engineering (ICSE). Zurich, Switzerland, 2012: 156-166

[72] Coppa E, Demetrescu C, Finocchi I. Input-sensitive profiling//Proceedings of the Conference on Programming Language Design and Implementation (PLDI). Beijing, China, 2012, 47(6): 89-98

[73] Shen D, Luo Q, Poshyvanyk D, et al. Automating performance bottleneck detection using search-based application profiling //Proceedings of the International Symposium on Software Testing & Analysis (ISSTA). Baltimore, USA, 2015: 270-281

[74] Toffola L D, Pradel M, Gross T R. Synthesizing programs that expose performance bottlenecks//Proceedings of the 16th International Symposium on Code Generation and Optimization (CGO). Vienna, Austria, 2018: 314-326

[75] Mudduluru R, Ramanathan M K. Efficient flow profiling for detecting performance bugs//Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA). Saarbrücken, Germany, 2016: 413-424

[76] Yu X, Han S, Zhang D, et al. Comprehending performance from real-world execution traces: A device-driver case//Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Salt Lake City, USA, 2014: 193-206

[77] Chen Z, Chen B, Xiao L, et al. Speedoo: Prioritizing performance optimization opportunities//Proceedings of the International Conference on Software Engineering (ICSE). Gothenburg, Sweden, 2018: 811-821

[78] Jovic M, Adamoli A, Hauswirth M. Catch me if you can: Performance bug detection in the wild//Proceedings of the 26th ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA). Portland, USA, 2011: 155-170

[79] Curtsinger C, Berger E D. Coz: Finding code that counts with causal profiling//Proceedings of the 25th Symposium on Operating Systems Principles (SOSP). Monterey, USA, 2015: 184-197

[80] Weng L, Hu Y, Huang P, et al. Effective performance issue diagnosis with value-assisted cost profiling//Proceedings of the 18th European Conference on Computer Systems (EuroSys). Rome, Italy, 2023: 1-17

[81] Cotroneo D, Pietrantuono R, Russo S, et al. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. Journal of Systems and Software (JSS), 2016, 113: 27-43

[82] Han X, Carroll D, Yu T. Reproducing performance bug reports in server applications: The researchers' experiences. Journal of Systems and Software (JSS), 2019, 156: 268-282

[83] Cavezza D G, Pietrantuono R, Alonso J, et al. Reproducibility of environment-dependent software failures: An experience report//Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE). Naples, Italy, 2014: 267-276

[84] Segura S, Troya J, Durán A, et al. Performance metamorphic testing: Motivation and challenges//Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSe-NIER). Buenos Aires, Argentina, 2017: 7-10

[85] Segura S, Troya J, Durán A, et al. Performance metamorphic testing: A proof of concept. Information and Software Technology (IST), 2018, 98: 1-4

[86] Barr E T, Harman M, McMinn P, et al. The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering (TSE), 2014, 41(5): 507-525

[87] Dhok M, Ramanathan M K. Directed test generation to detect

loop inefficiencies//Proceedings of the 4th ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE). Seattle, USA, 2016: 895-907

[88] Wei J, Chen J, Feng Y, et al. Singularity: Pattern fuzzing for worst case complexity//Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(FSE). Lake Buena Vista, USA, 2018: 213-223

[89] Stefan P, Horky V, Bulej L, et al. Unit testing performance in java projects: Are we there yet?//Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering(ICPE). L'Aquila, Italy, 2017: 401-412

[90] Leitner P, Bezemer C P. An exploratory study of the state of practice of performance testing in java-based open source projects//Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering(ICPE). L'Aquila, Italy, 2017: 373-384

[91] Ding Z, Chen J, Shang W. Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering(ICSE). Seoul, Republic of Korea, 2020: 1435-1446

[92] Huang P, Ma X, Shen D, et al. Performance regression testing target prioritization via performance risk analysis//Proceedings of the 36th International Conference on Software Engineering(ICSE). Hyderabad, India, 2014: 60-71

[93] Mostafa S, Wang X, Xie T. Perfranker: Prioritization of performance regression tests for collection-intensive software //Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA). Santa Barbara, USA, 2017: 23-34

[94] Ba J, Rigger M. CERT: Finding performance issues in database systems through the lens of cardinality estimation//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering(ICSE). Lisbon, Portugal, 2024: 1-13

[95] Johnston O, Jarman D, Berry J, et al. Metamorphic relations for detection of performance anomalies//Proceedings of the 2019 IEEE/ACM 4th International Workshop on Metamorphic Testing(MET). Montreal, Canada, 2019: 63-69

[96] Nolasco A, Molina F, Degiovanni R, et al. Abstraction-aware inference of metamorphic relations//Proceedings of the 32nd ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(FSE). Porto de Galinhas, Brazil, 2024: 450-472

[97] Chen B, Liu Y, Le W. Generating performance distributions via probabilistic symbolic execution//Proceedings of the 38th International Conference on Software Engineering(ICSE). Austin, USA, 2016: 49-60

[98] Li C, Wang S, Hoffmann H, et al. Statically inferring performance properties of software configurations//Proceedings of the 15th European Conference on Computer Systems(EuroSys). Heraklion, Greece, 2020: 1-16

[99] Velez M, Jamshidi P, Siegmund N, et al. White-box analysis over machine learning: Modeling performance of configurable systems//Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering(ICSE). Madrid, Spain, 2021: 1072-1084

[100] He H, Jia Z, Li S, et al. CP-detector: Using configuration-related performance properties to expose performance bugs//Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering(ASE). Melbourne, Australia, 2020: 623-634

[101] Li Yun-Feng. Research on Configuration Fault Diagnosis Technology for Performance Optimization[M. S. dissertation]. National University of Defense Technology, Changsha, 2019 (in Chinese)
(李云峰. 面向性能优化的配置故障诊断技术研究[硕士学位论文]. 国防科技大学, 长沙, 2018)

[102] Trubiani C, Pinciroli R, Biaggi A, et al. Automated Detection of Software Performance Antipatterns in Java-Based Applications. IEEE Transactions on Software Engineering(TSE), 2023, 49(4): 2873-2891

[103] Bouzenia I, Krishan B P, Pradel M. DyPyBench: A benchmark of executable Python software//Proceedings of the 32nd ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(FSE). Porto de Galinhas, Brazil, 2024: 338-358

[104] Maricq A, Duplyakin D, Jimenez I, et al. Taming performance variability//Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation(OSDI). Carlsbad, USA, 2018: 409-425

[105] He S, Manns G, Saunders J, et al. A statistics-based performance testing methodology for cloud applications//Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(FSE). Tallinn, Estonia, 2019: 188-199

[106] Laaber C, Würsten S, Gall H C, et al. Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(FSE). Virtual Event, USA, 2020: 989-1001

[107] Wu Y, Arulraj J, Lin J, et al. An empirical evaluation of in-memory multi-version concurrency control. Proceedings of the VLDB Endowment, 2017, 10(7): 781-792

[108] Wang Y, Yu M, Hui Y, et al. A study of database performance sensitivity to experiment settings. Proceedings of the VLDB Endowment, 2022, 15(7): 1439-1452

[109] Chen Y, Schwahn O, Natella R, et al. SlowCoach: Mutating code to simulate performance bugs//Proceedings of the 33rd International Symposium on Software Reliability Engineering (ISSRE). Charlotte, USA, 2022: 274-285

[110] Wang J, Huang Y, Chen C, et al. Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering (TSE), 2024, 50(4): 911-936

HE Hao-Chen, Ph. D., assistant professor. His main research interests include software reliability, software performance, testing and edge computing.

LI Shan-Shan, Ph. D., professor, Ph. D. supervisor. Her main research interests include software reliability and intelligent software engineering.

JIA Zhou-Yang, Ph. D., associate professor. His main research interests include software reliability and general operating system.

YAO Yi-Heng, Ph. D. candidate. His main research interests include intelligent software and industrial software.

ZHANG Yuan-Liang, Ph. D., lecturer. His main research interests include software reliability and software evolution.

WANG Ji, Ph. D., professor, Ph. D. supervisor. His main research interests include software reliability, formalization and program verification.

LIAO Xiang-Ke, Ph. D., professor, Ph. D. supervisor. His main research interests include system software and general operating system.

Background

This paper focuses on a popular research problem: performance bug. For decades, research has been proposed to detect performance bugs. However, performance bugs are hard to detect due to three main reasons: (1) trigger performance bugs need specific workload, (2) the symptoms of performance bugs are usually implicit so they lack of test oracle, (3) performance bugs have many root causes so is hard to design a general tool to solve them.

Existing works tackle performance bugs in three aspects: use specific patterns to detect specific types; try to conclude findings to guide the test oracle for performance bugs; optimize typical Profiling-based method. But existing works can still not design general tools to detect performance bugs and are also not able to scale machine-learning-based software systems.

To guide future potential opportunities in performance bug detection, this paper summarize existing works and conclude the shortcomings and challenges.

The authors of this paper have been working in performance bug detection for 2～6 years and has published 4 full papers (ASE, ICSE, VLDB) in this field. This paper is the guidance of our research program Nos. 62272473, 2023RC1001, 62202474, making a comprehensive study of challenges that are still not being solved.

This research was funded by the National Natural Science Foundation of China (Nos. 62272473, 62202474), the Science and Technology Innovation Program of Hunan Province (No. 2023RC1001).