软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(7):3022−3040 [doi: 10.13328/j.cnki.jos.007336] [CSTR: 32375.14.jos.007336]

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

深度学习编译器缺陷实证研究: 现状与演化分析*

沈庆超[1], 田家硕[1], 陈俊洁[1], 陈 翔[2], 陈庆燕[3], 王 赞[1]

[1](天津大学 智能与计算学部, 天津 300350)

[2](南通大学 人工智能与计算机学院, 江苏 南通 226019)

[3](山东航空学院 信息工程学院, 山东 滨州 251700)

通信作者: 陈俊洁, E-mail: junjiechen@tju.edu.cn

摘 要: 深度学习编译器已被广泛应用于深度学习模型的性能优化和部署. 与传统编译器类似, 深度学习编译器也存在缺陷. 存在缺陷的深度学习编译器会导致编译失败或者产生错误的编译结果, 甚至有时会带来灾难性的后果. 为了深入理解深度学习编译器缺陷的特性, 已有工作针对深度学习编译器早期的 603 个缺陷进行研究分析. 近年来, 深度学习编译器在快速迭代更新, 伴随着大量新特性的引入和旧特性的弃用. 与此同时, 一些针对深度学习编译器缺陷的检测工具已被开发出来. 因此, 需要分析之前对深度学习编译器缺陷的研究结论是否依然适用. 此外, 缺乏对缺陷症状、根因、位置三者之间关系的深入挖掘, 并且缺乏对触发缺陷的回归测试用例特征和修复缺陷的补丁特征的研究. 为了深入分析当下深度学习编译器缺陷特征和缺陷分布随时间的演化过程, 收集当前 3 款主流深度学习编译器 (即 Apache 的 TVM、Facebook 的 Glow 和华为的 AKG) 中的 613 个近期修复的缺陷, 并对缺陷的根因、症状、位置等特征进行人工标注. 基于标注结果, 从多个不同角度深入挖掘缺陷的分布特征,并与已有研究进行对比分析. 同时, 对触发缺陷的回归测试用例和修复缺陷的补丁进行研究. 最终获得 12 个主要研究发现, 以全面了解深度学习编译器缺陷现状与演变过程, 并为深度学习编译器缺陷的检测、定位、修复提供一系列可行的指导方案. 最后, 为了验证这些研究发现的有效性, 开发了一款基于优化配置的测试工具 CfgFuzz. CfgFuzz 通过对编译配置选项进行组合测试, 最终检测到 8 个 TVM 缺陷, 其中 7 个缺陷已被开发人员确认或修复.

关键词: 深度学习编译器; 缺陷分析; 实证研究; 缺陷检测; 缺陷特征

中图法分类号: TP311

中文引用格式: 沈庆超, 田家硕, 陈俊洁, 陈翔, 陈庆燕, 王赞. 深度学习编译器缺陷实证研究: 现状与演化分析. 软件学报, 2025, 36(7): 3022–3040. http://www.jos.org.cn/1000-9825/7336.htm

英文引用格式: Shen QC, Tian JS, Chen JJ, Chen X, Chen QY, Wang Z. Toward Understanding the Current Status and Evolution of Deep Learning Compiler Bugs. Ruan Jian Xue Bao/Journal of Software, 2025, 36(7): 3022–3040 (in Chinese). http://www.jos.org.cn/1000-9825/7336.htm

Toward Understanding the Current Status and Evolution of Deep Learning Compiler Bugs

SHEN Qing-Chao[1], TIAN Jia-Shuo[1], CHEN Jun-Jie[1], CHEN Xiang[2], CHEN Qing-Yan[3], WANG Zan[1]

[1](College of Intelligence and Computing, Tianjin University, Tianjin 300350, China)

[2](School of Artificial Intelligence and Computer Science, Nantong University, Nantong 226019, China)

[3](School of Information Engineering, Shandong University of Aeronautics, Binzhou 251700, China)

Abstract: Deep Learning compilers (DL compilers) are widely applied to optimize and deploy deep learning models. Similar to traditional compilers, DL compilers also possess bugs. The buggy DL compilers can cause compilation failures, generate incorrect compilation results

* 基金项目: 国家自然科学基金 (62322208, 12411530122)
本文由“新兴软件与系统的可信赖性与安全”专题特约编辑向剑文教授、陈厅教授、杨珉教授、周俊伟教授推荐.
收稿时间: 2024-08-24; 修改时间: 2024-10-15; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-03

and even lead to disastrous consequences sometimes. To deeply understand the characteristics of DL compiler bugs, the existing works have analyzed 603 early bugs in DL compilers. In recent years, DL compilers have been updated rapidly, along with the introduction of a large number of new features and the abandonment of some old ones. At the same time, several bug detection approaches for DL compilers have been developed. Therefore, it is necessary to analyze whether the previous research conclusions on DL compiler bugs are still applicable. In addition, there is a lack of in-depth exploration of the relationship among the symptoms, root causes, and locations of bugs, and the characteristics of bug-revealing tests and bug-fixing patches have not been studied. To deeply analyze the evolution process of the current DL compiler bug characteristics and distribution over time, 613 recently fixed bugs in three mainstream DL compilers (i.e., TVM of Apache, Glow of Facebook, and AKG of Huawei) are collected in this study, and the characteristics such as root causes, symptoms and locations of bugs are manually labeled. Based on the labeling results, this study deeply explores the distribution characteristics of bugs from multiple dimensions and compares them with that in the existing works. Meanwhile, we further investigate the characteristic of bug-revealing regression tests and bug-fixing patches. In total, this study summarizes 12 major findings to comprehensively understand the current situation and evolution of DL compiler bugs and provide a series of feasible suggestions for the detection, location, and repair of DL compiler bugs. Finally, to verify the effectiveness of the research findings in this work, a testing tool CfgFuzz based on optimized configuration is developed. CfgFuzz conducts combinatorial tests on compilation configuration options and finally detects 8 TVM bugs, 7 of which have been confirmed or fixed by developers.

Key words: deep learning compiler; bug analysis; empirical study; bug detection; bug characteristic

深度学习 (deep learning, DL) 已经被广泛应用到各个领域, 例如自动驾驶[1]、人脸识别[2]和软件工程[3−6]. 近年来, 为了实现 DL 模型简易部署和高效推理目标, 深度学习编译器 (DL 编译器) 应运而生. DL 编译器以预训练的 DL 模型作为输入, 通过执行一系列优化操作后, 生成可在特定硬件平台上高效执行的二进制代码.

作为深度学习领域的基础性软件, DL 编译器发挥着日益重要的作用, 存在缺陷的 DL 编译器将引起广泛的危害. 具体而言, DL 编译器中的缺陷有两方面的潜在影响. 一方面, DL 编译器的缺陷可能会传播到所有经其编译的 DL 模型中, 为大量的 DL 模型埋下安全隐患. 另一方面, DL 编译器的缺陷使得对 DL 软件的异常诊断变得复杂, 开发人员难以判断 DL 软件的异常表现究竟源于模型构造阶段的错误, 还是 DL 编译器在编译过程中引入的缺陷, 这无疑加剧了 DL 软件中缺陷的排查难度. 因此, 深入理解 DL 编译器的缺陷并提升 DL 编译器的质量至关重要.

DL 编译器借鉴了传统编译器的编译流程方案来解决模型优化部署问题, 但这两类编译器在处理对象和优化方式层面有着显著的不同[7]. 一方面, 传统编译器以一种高级编程语言 (例如, Java、C++) 的源码程序作为输入, 并输出另一种低级编程语言形式 (例如, 汇编语言、机器码) 的程序代码. 和传统编译器的编译对象不同, DL 编译器处理的程序 (即 DL 模型) 缺乏明确的逻辑结构. 另一方面, DL 编译器具备特有的多级代码中间表示 (intermediate representation, IR) 和大量针对深度学习特性设计的优化操作 (如算子融合). 因此, 已有针对传统编译器的缺陷检测和定位技术对于 DL 编译器并不适用.

为了了解 DL 编译器缺陷的特征, Shen 等人[8]首次对编译器缺陷展开实证研究. 该工作对 TVM[9]、Glow[10]和 nGraph[11]这 3 款 DL 编译器中 603 个缺陷展开研究, 并得到了一系列切实可用的发现和启示, 推动了 DL 编译器测试技术的发展[12,13]. 近年来, 随着 DL 编译器的经历快速发展和迭代更新, 已有发现难以适用于当前 DL 编译器的缺陷检测与调试, 其主要原因可总结为如下 3 点. (1) 已有工作所研究的 DL 编译器已发生很大的变化, 例如, TVM 编译器在近 3 年的代码变更次数 (即项目仓库中已合并的拉取请求次数) 超过 5 000 次, nGraph 项目已经停止维护; 近些年来涌现出一些新的 DL 编译器 (比如, 华为的 AKG[14]和 Intel 的 OpenVINO[15]). (2) 已有工作研究的缺陷均为 2020 年 11 月之前检测并修复的, 距今时间较为久远. (3) 已有工作只关注缺陷本身的特征, 并没有对触发缺陷的回归测试用例以及修复缺陷的补丁特征进行挖掘. 回归测试和缺陷补丁的研究同样有助于自动化缺陷检测与修复方法的设计.

为了更加全面地调研当下 DL 编译器缺陷的特征, 本文选择 3 款 DL 编译器作为研究对象, 包括 Apache 的 TVM[9]、Facebook 的 Glow[10]和华为的 AKG[14]. 对于每款 DL 编译器, 我们收集了 2020 年 11 月之后, 且超过 15 个月时间的新修复缺陷. 经过人工检查和去重后, 我们最终获得了 613 个 DL 编译器缺陷, 并对每个缺陷的根因、症状和位置进行人工标注. 接着, 本文采用对比分析的方式, 分别从缺陷的根因、症状和位置这 3 个方面分析 DL

编译器缺陷分布随时间的变化情况. 同时, 本文进一步深入分析了 DL 编译器缺陷的根因、症状、位置三者之间的内在联系情况. 此外, 为了进一步挖掘缺陷检测与修复的特征, 本文对触发 DL 编译器缺陷的回归测试用例和修复 DL 编译器缺陷的补丁进行了研究.

对于缺陷数据的标注结果, 我们识别出 13 个缺陷根因和 6 个缺陷症状. 基于缺陷标注结果的分析, 我们得到了 12 个主要发现, 并为今后的 DL 编译器缺陷检测和调试提供一系列建议. 此外, 基于本文的研究发现, 设计了一款高效的 DL 编译器测试工具 CfgFuzz. 经过 24 h 的测试, CfgFuzz 检测到了 8 个新的 TVM 缺陷, 其中 7 个缺陷已经被开发者确认或修复. 该实验结果进一步验证了本文研究发现的有效性.

本文主要贡献如下.

- 针对 DL 编译器缺陷展开了全面的实证研究, 研究包含 3 款 DL 编译器中 613 个缺陷数据, 并针对缺陷症状、根因、位置随时间变化情况、缺陷不同因素的内在联系、缺陷补丁与触发缺陷的回归测试用例的特征这 6 个问题展开调研.
- 获得了 12 条主要研究发现, 以全面了解深度学习编译器缺陷及其演变过程, 并为后续的 DL 编译器缺陷检测、定位与修复等研究提供了一系列有益的启示.
- 设计了一款概念验证工具 CfgFuzz, 检测到了 8 个之前未知的 TVM 缺陷, 其中 7 个缺陷已被确认或修复.
- 开源了数据集和 CfgFuzz 源码, 促进未来针对 DL 编译器缺陷检测等相关工作.

本文第 1 节总结针对 DL 编译器缺陷实证研究的相关工作. 第 2 节介绍 DL 编译器的背景知识. 第 3 节介绍数据收集和标注的方法与本文关注的研究问题. 第 4 节回答研究问题并探讨未来的研究方向和潜在解决方案. 第 5 节介绍启发与应用. 第 6 节分析研究效度威胁. 第 7 节总结全文.

1 相关工作

目前, 对缺陷的实证研究工作得到了越来越多的关注. 与本文最相关的工作是 Shen 等人[8]对 DL 编译器缺陷的实证研究. Shen 等人的研究仅仅关注了 DL 编译器发展初期的缺陷特征. 作为一款新型系统软件, DL 编译器近年来经历了高速迭代更新, 已有的研究结论是否适用当前主流 DL 编译器仍有待验证. 此外, Shen 等人的研究只关注了缺陷本身的特性, 缺少从回归测试用例和补丁的角度分析缺陷特征的研究. 为此, 本文收集当前 3 款主流编译器中的 613 个缺陷, 并针对缺陷症状、根因、位置随时间变化情况, 缺陷不同因素的内在联系, 缺陷补丁与触发缺陷的回归测试用例的特征等 6 个问题展开广泛且深入地研究.

此外, 存在较多针对其他类型软件系统的缺陷研究. Sun 等人[16]对 GCC 和 LLVM 的各种缺陷的持续时间、优先级、修复代码、测试用例和位置等特征进行统计和研究. Zhou 等人[17]对 GCC 和 LLVM 优化相关的缺陷进行了深入的研究. Humbatova 等人[18]对深度学习程序的缺陷类别展开了深入研究. Garcia 等人[19]对 TensorFlow 框架内部的缺陷进行了研究, Chen 等人[20]进一步对 4 款 DL 框架的缺陷展开了更全面的研究并评估了已有测试方法对 DL 框架中缺陷的检测效果. Islam 等人[21]及 Zhang 等人[22]对深度学习程序 (例如, TensorFlow[23]程序) 中的缺陷进行了研究. Lu 等人[24]对并发错误进行了实证研究. Di Franco 等人[25]对数值错误进行了实证研究. Han 等人[26]研究了高度可配置软件系统中的性能错误, Wan 等人[27]对区块链系统的错误进行了表征.

与其他类型软件系统的缺陷研究不同, 本文的研究对象是 DL 编译器中的缺陷. 正如前文介绍, DL 编译器与传统编译器 (如 GCC 和 LLVM) 具有不同的特征. 此外, DL 编译器是深度学习的重要基础设施之一, 与 DL 程序和 DL 框架有着显著不同的功能. 除此之外, 本文进一步分析了 DL 编译器缺陷随时间的演化特征并关注缺陷的补丁和测试用例的特性, 进而从更多的角度分析 DL 编译器的缺陷特征及其演化规律.

2 背景知识

图 1 显示了 DL 编译器的架构. 编译过程根据功能可以分为 3 个主要阶段. (a) 模型加载阶段: DL 编译器将从各种深度学习库 (例如, PyTorch[28]和 Keras[29]) 构建的 DL 模型作为输入, 并将它们转换为统一的高级中间表示

(High-level IR). DL 模型是一个有向无环图, 其节点表示运算符 (即张量的计算函数), 计算图的边代表数据流. 这种 High-level IR (也称为计算图级别 IR) 有助于隐藏来自不同 DL 框架构造的 DL 模型的差异, 从而简化优化执行. DL 模型中的每个运算符都会转换为语义上等效的一个或多个 IR 表达式. 例如, Keras 模型中的 Conv2D 运算符以及所有参数设置 (例如, *filters* 和 *strides*) 都将转换为 TVM 的 High-level IR 中的 *nn.conv2d*. (b) High-level IR 转换阶段: DL 编译器对 High-level IR 执行与硬件无关的优化 (例如, 算子融合), 以提高计算效率. (c) Low-level IR 转换阶段: High-level IR 被转换为 Low-level IR, 在此期间执行特定于硬件的优化 (例如, 内存延迟隐藏). 最后, DL 编译器基于 Low-level IR 生成指定目标硬件上的可执行代码.

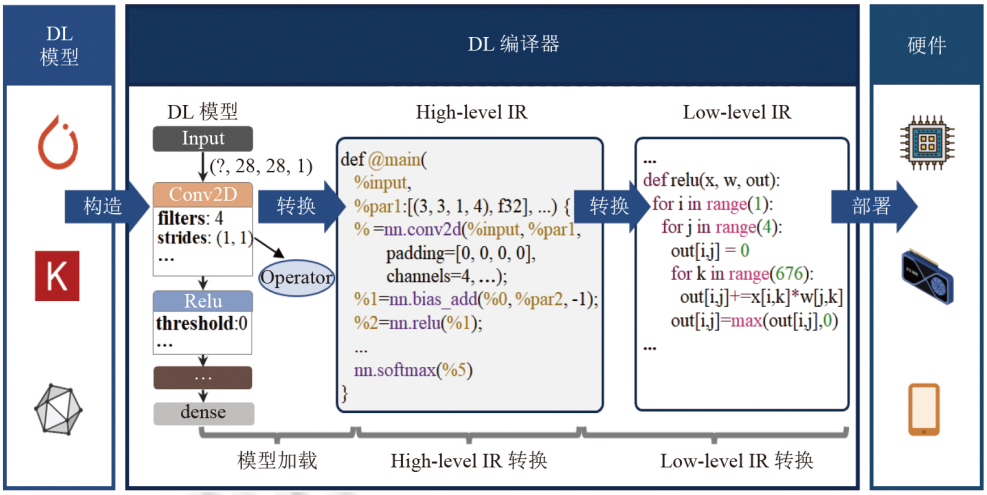

图 1 DL 编译器架构

在研究 DL 编译器缺陷特征时, 症状、根因与位置是 3 个关键的分析维度. 缺陷症状是指编译器在运行过程中表现出的异常行为, 通常为用户或开发者可以直接观察到的现象 (如, 崩溃). 根因则是引起缺陷的深层次根本原因 (如, API 误用). 位置则指存在缺陷的代码所处的编译阶段 (如, 模型加载阶段). 除了缺陷本身特征, 本文还从触发缺陷的回归测试用例和修复缺陷的补丁两个维度挖掘缺陷的特征. 基于这 5 个维度对缺陷分布及其演化方向进行全面分析, 有助于指导研究人员更有效地设计缺陷检测、定位和自动修复的方法.

本工作选择了当前 3 款主流 DL 编译器 (即 Apache 的 TVM[9]、Facebook 的 Glow[10]和华为的 AKG[14]) 作为实验研究对象. 鉴于 nGraph 仓库已停止维护, 本文不再研究 nGraph, 而是新增了当前主流的 AKG 编译器作为新的研究对象. 这 3 款编译器在结构和实现上具有多样性. 例如, TVM 在 Low-level IR 转换阶段使用机器学习的方法实现自动调优 (auto tuning). Glow 将 High-level IR 的每个算子转化成多个 Low-level IR 的原语以实现算子空间的缩减, 使得新的目标硬件平台只需专注于少量的线性代数原语. AKG 不包含模型加载阶段, 而是在其上游程序 MindSpore[30]中进行模型加载. 这种多样性有助于我们更全面地了解和研究 DL 编译器缺陷的特性.

3 研究方法与分类

3.1 研究问题

DL 编译器的快速发展带来了新功能的引入和老旧功能的淘汰, 然而这种快速迭代也伴随着不同类型的缺陷的产生. 了解这些缺陷的分布及其变化趋势, 可以为未来的缺陷检测、定位与修复提供指导[31]. 此外, 回归测试用例和补丁中包含丰富的缺陷特征[32], 然而已有的研究工作仅仅关注了 DL 编译器缺陷本身的特征, 忽略了触发缺陷的回归测试用例和修复缺陷的补丁角度的研究. 因此, 本工作设计了以下 6 个研究问题其中 RQ1 到 RQ3 关注 DL 编译器缺陷的分布情况与缺陷分布随时间的变化情况, RQ4 进一步探索研究根因、症状、位置三者之间的内

在联系, RQ5 到 RQ6 从触发缺陷的测试用例和修复缺陷的补丁角度分析 DL 编译器的缺陷特征.

RQ1: DL 编译器缺陷的根因类型分布及其随时间的变化趋势如何?

深入研究 DL 编译器缺陷的根因类型, 能够揭示缺陷产生的本质原因, 这是提升编译器质量、减少未来缺陷的关键. 随着技术的不断进步和 DL 编译器的快速迭代, 新特性的引入与旧功能的弃用可能导致缺陷根因的多样性和动态变化. 因此, 分析根因的分布及其变化趋势, 对于制定有效的缺陷检测、定位和修复策略至关重要.

RQ2: DL 编译器缺陷的症状分布及其随时间的变化趋势如何?

缺陷的症状直接关联到其对 DL 编译器功能和性能的实际影响. 同时, 了解不同症状的分布及其变化趋势, 有助于指导测试人员设计针对特定症状类型的 DL 编译器测试方法. 确保关键缺陷被及时发现并修复.

RQ3: DL 编译器缺陷的位置分布及其随时间的变化趋势如何?

DL 编译器包含 3 个主要阶段, 不同阶段存在缺陷的数量存在差异. 明确缺陷的位置分布, 可以帮助开发者识别代码中的“脆弱点”, 从而让缺陷检测和调试工作更有针对性. 同时, 随着 DL 编译器的不断迭代发展, DL 编译器不同阶段缺陷分布会发生变化, 研究缺陷位置的变化规律对于适应 DL 编译器不断演化的需求, 优化测试、调试和定位策略具有重要意义.

RQ4: 缺陷根因、症状、位置三者之间的内在关联如何?

缺陷的根因、症状和位置是理解缺陷全貌的 3 个重要维度. 探索它们之间的内在联系, 可以构建更全面的缺陷知识图谱, 揭示缺陷产生的深层次原因和规律. 这种综合视角有助于开发更精确的缺陷检测方法、设计更有效的缺陷修复方法, 并提升 DL 编译器的整体质量和稳定性.

RQ5: 回归测试用例与历史测试用例的差异程度如何?

修复缺陷的过程中, 开发者通常还会增加或修改源码中配套的单元测试用例, 使其能够触发该缺陷. 了解回归测试用例与历史测试用例的差异程度能够在一定程度上反映已有测试用例与触发新缺陷的差异程度, 并能够为未来 DL 编译器缺陷检测方法的设计提供指导方向.

RQ6: 修复 DL 编译器缺陷的复杂程度如何?

缺陷修复的复杂程度是衡量缺陷影响范围和修复难度的重要指标. 通过分析真实缺陷的修复补丁大小、修复过程中涉及的代码量等因素, 可以评估缺陷修复的整体复杂性和所需资源. 这对于制定合理的修复计划、评估自动化修复技术的可行性和效果具有重要意义. 同时, 该问题的研究还有助于指导未来 DL 编译器设计和开发过程中如何降低缺陷的复杂性和修复成本.

3.2 数据收集

为了回答以上 6 个研究问题, 本文研究选取了当前 3 款主流的 DL 编译器作为研究对象, 包括 Apache 的 TVM[9], Facebook 的 Glow[10]以及华为的 AKG[14]. 为了分析 DL 编译器缺陷的特点, 我们参考已有工作从 DL 编译器的 GitHub 仓库中搜集已被合并到项目源码中的拉取请求 (pull request, PR)[19,33]. 为了确保数据的准确性和相关性, 我们从 PR 中筛选出标题或标签中至少包含一个与缺陷修复相关的关键词 (包括 fix、defect、error、bug、issue、mistake、incorrect、fault 和 flaw) 的 PR. 本文选择已合并的 PR 作为研究数据来源主要是基于以下两个考虑: 首先, 已合并的 PR 表明缺陷已经得到了开发者的认可并完成了修复, 因此它们能够真实反映 DL 编译器在实际使用中遇到的问题; 其次, 这些 PR 通常包含缺陷的描述, 开发者的讨论和缺陷的详细修复过程, 这些信息是深入分析缺陷的根因, 症状与位置重要的信息源. 然而根据关键词筛选出的 PR 存在非缺陷修复的任务 (如功能增强或代码重构). 此外, 存在多个缺陷在一个 PR 中被同时修复的情况. 因此, 在后续的数据处理过程中, 我们进一步对收集到的 PR 进行了详细的筛选, 过滤掉所有非缺陷修复功能的 PR, 并将包含多个缺陷的 PR 拆分开, 以确保我们分析的数据集专注于缺陷修复且每条数据只包含一个缺陷.

遵循已有研究工作[8,19,32]的缺陷分析方式, 本文采用人工标注的方式对缺陷的症状、根因等因素进行标注. 由于 DL 编译器新增的缺陷数量规模较大, 人工分析的方式无法支持对所有新增缺陷的标注, 为此, 我们从所有收集到的 PR 中随机筛选了一部分 PR 进行了深入的分析和分类, 表 1 的第 2 行即为手动分析的缺陷相关的 PR 数量,

并在表 1 的最后一行中列出了从中识别出的缺陷数量. 经过严格的筛选和分析, 我们从 914 个缺陷相关的 PR 中识别出 613 个缺陷, 包括 437 个 TVM 缺陷、84 个 Glow 缺陷和 92 个 AKG 缺陷.

表 1 缺陷数据统计

编译器	PR数量	缺陷数量
TVM	663	437
Glow	127	84
AKG	124	92
总和	914	613

3.3 分类与标注过程

为了获取缺陷的根因、症状和缺陷在 DL 编译器 3 个阶段中的位置, 本文遵循已有研究工作的方式[8,20,32], 采用人工标注的方式对每个缺陷进行标注. 具体来说, 为了标注缺陷的根因与症状, 本工作使用已有工作[34−36]中的根因和症状分类法作为初始分类法, 然后按照标注数据的实际特性对其进行调整, 使其适应新增 DL 编译器缺陷的特征[21]. 为了标注缺陷出现的位置, 我们使用第 2 节中定义的 3 个不同编译阶段作为 DL 编译器位置类别. 对于每个 DL 编译器中的源码文件, 我们通过阅读文档、源码及注释, 理解每个源码文件的功能, 并将其归纳到 DL 编译器的不同编译阶段. 以上所有过程均为两位作者共同完成.

基于上述方式获取的缺陷根因、症状与位置类别, 以及每个源码文件对应的编译阶段, 两位作者根据已有研究的开放编码方案[20], 分别独立地对于每个缺陷进行人工分析和标记. 具体而言, 针对每个缺陷, 两位作者首先深入理解缺陷修复的代码变更、与缺陷关联的问题 (issue) 的描述内容和开发者在 PR 或 issue 中的讨论. 在标记过程中, 为了评估两位作者之间标注结果的一致性, 本文采用 Cohen's Kappa 系数[37]作为衡量标准. 在初步标记的前 5% 数据中, Cohen's Kappa 系数仅为 40%, 这显示了初始标注过程中存在一定比例的不一致性. 因此, 我们组织了一次针对性的标记培训. 随后, 在包含前 5% 数据的前 10% 标记结果中, Cohen's Kappa 系数显著提升至 86%, 标注的一致性得到了显著提高. 通过进一步的讨论和修正不一致之处, 在后续的标记工作中, Cohen's Kappa 系数均保持在 94% 以上. 对于每次标记过程中出现的不一致点, 两位作者都会与第 3 位作者进行深入讨论, 以确保所有缺陷都能得到一致且准确的标记结果.

此外, 我们采用半自动化的统计策略, 理化分析触发缺陷的回归测试用例修改程度和修复缺陷的补丁的大小. 具体而言, 在分析回归测试用例与已有测试用例的差异程度时, 我们首先通过人工检查判定回归测试用例属于完全新增的类别还是基于已有测试用例修改得到的. 对于基于已有测试用例进行修改得到的回归测试用例, 我们进一步统计其代码变更的行数. 代码变更的行数新增的代码行数与删除的代码行数总和. 需要注意的是, 一行代码的修正涉及两行代码的变更, 包括删除原先错误的代码和新增一行正确的代码. 鉴于缺陷修复补丁的大小受到代码注释、不同函数大小等多种因素的显著影响, 我们决定采用受影响的函数个数作为度量标准. 具体而言, 我们通过分析每个缺陷修复过程中涉及的文件更改行号, 并与相应版本的文件进行比对, 从而精确计算出每个缺陷修复所影响的函数数量. 这种方法能够更为准确地反映缺陷修复补丁的实际规模和影响范围.

4 结果与分析

4.1 RQ1: 根因

4.1.1 根因分类结果

基于上述分类和标注, 我们确定了以下 13 个引起 DL 编译器缺陷的根因.

● 逻辑错误. 此类缺陷一般涉及多个语句或代码块中的逻辑. 代码逻辑指的是算法的实现逻辑, 例如在计算图上的优化. 根据缺陷发生的位置, 将这类缺陷分为两个子类. (1) 错误的优化代码逻辑: 如第 2 节中所述, 优化是 DL 编译器的核心功能. DL 编译器通常包含计算图级别的 High-level 优化和硬件级别的 Low-level 优化等多级优化策略. 这一子类缺陷发生在各种优化实现的位置. (2) 错误的非优化代码逻辑: 此类缺陷包括除了优化代码以外

发生的逻辑缺陷.

● 张量形状错误. 这类缺陷是指与张量形状 (shape) 或布局 (layout) 相关的问题. 张量的形状与布局对模型的性能起着重要的作用, 其中张量形状代表了每个维度中的元素数量, 而布局则描述了张量在内存中的表示方式[7]. 而且这类缺陷大部分发生在张量形状匹配、张量形状转换、张量形状推理、数据布局转换等操作过程中.

● API 误用. 这类缺陷主要源于开发者对 API 的理解或使用不当而产生. 具体表现为在编程过程中错误地选择了 API、运用了不恰当的参数、在 API 使用过程中遗漏或过度进行了条件检查, 以及不必要或缺失的 API 调用等情况.

● 类型错误. 这类缺陷包含类型转换和类型推理等类型相关问题. DL 编译器以计算图为核心, 其中节点 (node) 代表深度学习运算符 (operator), 而边 (edge) 则代表张量数据 (tensor). 此类别包含 3 个子类. (1) 节点类型错误: 这个类别是指涉及节点类型定义或使用的问题. 在 DL 编译器中, 每个节点的输入可以是 0 个或多个张量, 输出为一个或多个张量. (2) 张量类型错误: 这个类别是指涉及张量数据类型的问题. 张量是一个包含单一数据类型元素的多维数组 (或称为多维矩阵), 用于表示深度学习中的数据和参数. (3) 传统数据类型错误: 这个类别是指涉及 DL 编译器中除了节点和张量之外, 还使用到的常规数据类型的问题. 这些常规数据类型包括传统软件系统中广泛使用的整型、浮点型、字符串等, 它们用于描述编译器内部状态、配置参数等.

● 异常处理错误. 此类缺陷是对异常情况的错误管理. 包括在理应触发异常的情形下, 编译器未能正确抛出异常; 或者在正常情况下错误地抛出了异常. 同时还包括提供不准确或有误导性的异常信息, 使得调试和故障排除变得困难.

● 配置错误. 该类型的缺陷是由 DL 编译器中的错误的配置选项引起的, 例如 CmakeLists.txt 文件的错误配置信息.

● 赋值错误. 这类缺陷包括变量的错误初始化或赋值, 包括变量在使用前未进行声明或初始化就直接被引用, 或者变量被赋予了与预期不符的值.

● 并发问题. 此类缺陷涉及面向并发结构 (比如, 线程、临界区、锁) 的错误操作. 如死锁, 线程过多导致内存溢出等.

● 不兼容. 此类缺陷是由 DL 编译器内部各个组件之间或 DL 编译器与第三方库 (例如, PyTorch 和 cuDNN) 之间的接口、功能或规范不匹配造成的. 不兼容缺陷通常与技术差异、版本不匹配、缺少必要的支持等因素密切相关.

● 数值计算错误. 这类缺陷包括错误的数值类计算或使用, 例如不正确的使用运算符或操作数、非法除零、缺少操作符或操作数等.

● 注册问题. 此类缺陷包括头文件的缺失或者错误引入, API 或者类的属性缺少注册时直接调用.

● 拼写错误. 这类缺陷是由于开发人员粗心导致的代码拼写错字, 比如将变量名“annotations”误写为“annotation”.

● 其他. 当缺陷的根因不常见, 并且不属于上述任何一类时, 本文将其归纳为此类别.

注册问题为本文识别出来的 DL 编译器缺陷根因, 此类缺陷主要因为缺少 API 注册而直接调用而导致的. 14 个此类缺陷中有 12 个出现在 TVM 中, 占比 85.71%. 分析发现, 这类缺陷与 TVM 中复杂的双向跨语言 API 调用密切相关. 具体而言, TVM 使用外部函数接口 (FFI) 实现 Python 和 C++代码双向调用功能, API 跨语言调用需要先注册再调用. 缺少注册而直接调用跨语言声明的 API 将会导致程序崩溃. 尽管这个根因是新识别的根因, 但在 Shen 等人[8]收集的缺陷中已包含了 5 个此类缺陷, 由于数量不够显著, 被归纳为其他. 除了注册问题以外, 其他缺陷根因类别都已经被已有工作识别出来的, 因此, DL 编译器迭代更新几乎不会引入新的缺陷类别.

发现 1. 尽管 DL 编译器经历了长时间、大幅度迭代更新, 但缺陷根因的类别已近乎被全面识别.

4.1.2 根因分布情况

图 2 描述了 DL 编译器不同根因的缺陷数量分布对比情况. 其中旧数据 (图的左半部分) 是指已有工作[8]中研究的 603 个早期 DL 编译器缺陷, 新数据 (图的右半部分) 是指本文研究的 613 个近期被修复的 DL 编译器缺陷. 从图中可以看出, 逻辑错误是当下 DL 编译器最常见的缺陷根因. 118 个此类缺陷中, 包括 73 个是由错误的优化代

码逻辑造成的缺陷, 45 个由错误的非优化代码逻辑造成的缺陷. 随着深度学习的迅猛发展, DL 编译器中的优化算法也随之频繁更新, 以便与深度学习算法和目标硬件平台的快速发展相适应. 例如, 在过去 3 年中, TVM 的源码平均每行发生了 0.61 次修改 (TVM 版本区间: e31564e1e–d85ea256f). 因此, 复杂且频繁迭代的代码逻辑会导致在 DL 编译器的实现中引入更多的逻辑缺陷. 此外, 由于优化算法的复杂性和需要处理不同类型硬件的优化需求, 优化代码往往比非优化代码更复杂. 因此, 优化相关的逻辑缺陷占比 61.86%, 高于非优化的逻辑错误.

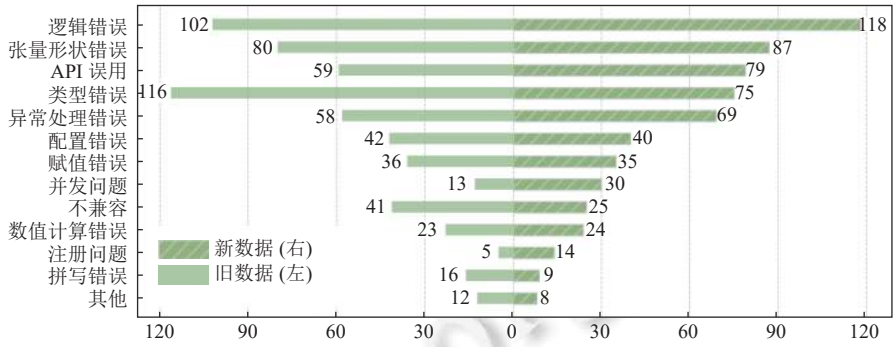

图 2 不同时期缺陷根因分布对比图

发现 2. 逻辑错误已成为 DL 编译器缺陷的主要根因, 占比高达 19.25%. DL 编译器的复杂逻辑和高频迭代使得逻辑错误 (特别是优化相关的逻辑错误) 占比显著上升.

张量形状问题上升至第 2 类最常见的缺陷根因, 包含 87 个缺陷, 占 DL 编译器缺陷的 14.19%, 包括 53 个 TVM 缺陷、9 个 Glow 缺陷和 15 个 AKG 缺陷. 由于张量在 DL 编译器中发挥重要作用, 且张量形状属于张量的一个重要属性, 所以 DL 编译器中除张量类型转换操作外, 也存在大量针对张量形状的操作. 深入分析发现, 当下 DL 编译器引入了动态的张量形状推理的新特性, 这使得 DL 编译器关于张量形状问题的缺陷数量与比例有一定量的上升. 具体来说, 87 个张量形状问题的缺陷中有 22 个缺陷和动态张量形状推理相关. 图 3 展示了一个张量形状推理错误导致性能缺陷的例子, 如果一个模型中有多个 *alter_op*, 第 1 个 *alter_op* 操作不会触发类型推断, 所以在执行后续的 *alter_op* 操作时, 张量形状信息将无法自动获得. 因此, 开发者通过手动添加张量形状动态推理操作来获取数据的对应张量形状.

```
-  m = get_const_tuple(data.checked_type.shape)[1]
+  try:
+      m = get_const_tuple(data.checked_type.shape)[1]
+  except ValueError:
+      data_infered = relay.transform.InferType()(tvm.IRModule.from_expr(data))["main"]
+      m = get_const_tuple(data_infered.ret_type.shape)[1]
```

图 3 一个与张量形状错误相关的缺陷 (TVM#PR-7308)

发现 3. 由于张量形状动态推理等新特征的引入使得张量形状问题的缺陷比例有所上升.

类型问题从最常见的缺陷根因下降为第 4 常见的缺陷根因, 占比下降 7.01%. 此类根因共引入 75 个缺陷, 包括 TVM 中 54 个、Glow 中 14 个以及 AKG 中 7 个. 进一步分析发现, 这类缺陷包含 3 个子类, 分别是节点类型错误, 张量类型错误和传统数据类型错误. 具体来说, 在 75 个类型缺陷中, 30 个缺陷是由于张量类型错误引起的, 40 个缺陷是由于传统数据类型错误引起的, 5 个缺陷是由节点类型错误引起的. 从中我们发现张量类型错误是类型错误中最常见的子类. 这是因为张量在 DL 编译器中频繁出现且发挥着重要作用. DL 编译器中的所有深度学习计算都依赖于一个或多个张量. 因此, 在编译过程中存在大量的类型转换 (尤其是隐式类型转换) 和类型推断操作.

这个结果表明, 在 DL 编译器中处理类型问题是非常容易出现错误的, 尤其是张量类型问题. 由于其易错性和重要性, 开发者应该更加关注张量类型方面的操作. 尽管类型错误是一类较为容易引入缺陷的根因, 但是分析发现, 关于类型的数据结构已经设计较为完备, 例如, 在 3 年的时间里, TVM 源码中与数据类型相关的数据结构定义文件行代码变更频次不足 0.16, 显著低于总体的代码修改频率 (即 0.61). 本文所研究的 3 款 DL 编译器中均没有很大的代码变更, 这也导致此类缺陷占比下降较为显著.

发现 4. 类型错误在 DL 编译器缺陷中依然是一类常见的根因, 特别是张量类型错误. 但其占比下降较为显著 (减少 7.01%). 这主要得益于类型相关的数据结构及代码已较为完善.

4.2 RQ2: 症状

4.2.1 症状分类

- 崩溃. 崩溃是指 DL 编译器在编译过程中意外终止, 通常会在终止后抛出一个错误信息.
- 结果错误. 结果错误是指在 DL 编译器没有意外终止的情况下, 以期望之外的方式产生了一个错误的结果或中间结果, 例如优化后生成了非等价的代码中间表示.
- 性能问题. 性能问题是指 DL 编译器所花费的时间成本或内存开销远远大于开发者或用户的期望值, 期望值是回归测试中 DL 编译器上一个版本所达到的性能, 或特定硬件的性能要求.
- 挂起. 挂起意味着 DL 编译器持续运行了很长一段时间, 但没有输出期望的结果.
- 构建失败. 构建失败是指 DL 编译器的安装过程意外终止.
- 未报告. 除了上述类别之外, 还有一些 DL 编译器错误, 其症状不属于以上任何一类或不能通过获取到的任何信息确定, 则将其分类为未报告.

4.2.2 症状分布情况

图 4 中展示了按症状类别的缺陷数量分布. 从图中可知, 崩溃一直是 DL 编译器缺陷最常见的症状, 该类缺陷占缺陷总数的 55.46%–59.37%. 在本工作的缺陷数据中, 分别有 244 个 TVM 缺陷、53 个 Glow 缺陷和 43 个 AKG 缺陷的症状表现为崩溃. 目前对 DL 编译器的测试工作[13,38–41]均使用了模型编译过程是否发生崩溃作为测试预言, 并检测到较多此类缺陷. 其中, Tzer[37]检测到了 32 个 TVM 崩溃缺陷, NNSmith[41]检测到了 13 个 TVM 崩溃缺陷, HirGen[13]检测到了 8 个崩溃缺陷, 然而, 这些测试技术均没有应用到 Glow 和 AKG 上. 将已有测试方法迁移应用到其他 DL 编译器上或许可以揭露出 Glow 和 AKG 中更多的崩溃缺陷. 此外, 设计能够同时检测所有 DL 编译器的通用测试方法变得尤为重要.

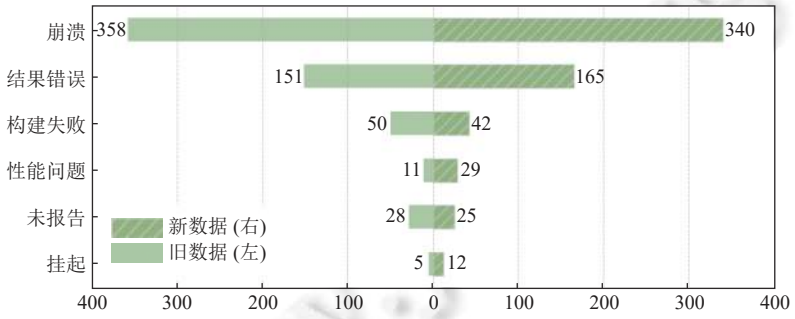

图 4 不同时期缺陷症状分布对比图

结果错误是指 DL 编译器产生错误的代码中间表示或非预期结果. 根据图 4 可知, 结果错误同样属于常见症状, 在不同时期的此类缺陷的占比为 25.04%–26.92%, 其中包括 115 个 TVM 缺陷、21 个 Glow 缺陷和 29 个 AKG 缺陷. 由于此类缺陷会在编译的过程中将缺陷传播到所有受编译的 DL 模型中, 从而引起更广泛的缺陷影响. 为了捕获此类缺陷, NNSmith[41]使用模型编译前后在相同输入下的预测结果是否一致作为测试预言, 并检测到 3 个此类缺陷. HirGen[13]使用 DL 编译器在不同优化级别或不同编译模式下对模型的编译结果是否相同作为新的测

试预言, 并检测到 9 个结果错误类型的缺陷. MT-DLComp[12]针对 DL 模型设计了等价的测试输入实现对 DL 编译器的测试, 并检测到 4 个此类缺陷. 这些测试预言能够捕获一定量的引起模型预测结果发生改变的缺陷. 然而, 这些测试预言无法检测到一些不会导致预测结果错误的缺陷. 因此, 设计一些细粒度的测试预言, 比如直接对比 IR 的结果是否正确可以暴露出更多此类缺陷.

发现 5. 崩溃和结果错误是 DL 编译器中两类最常见缺陷症状. 结果错误类型的缺陷难以检测, 且为此类缺陷设计高质量的测试预言依然存在挑战.

由图 4 可知, 挂起和性能问题的症状占比很低, 和已有研究[8]对比发现, 这两种症状的缺陷所占用的比例提高了 1.56 倍上. 性能缺陷具有隐蔽性和缺少合适的测试预言, 导致目前并没有针对 DL 编译器性能缺陷设计的测试方法. 由于提升模型性能为 DL 编译器的核心功能, 所以此类缺陷具有较大的危害性. 鉴于此类缺陷数量的快速增多, 因此, 设计一种针对 DL 编译器性能缺陷的检测方法的重要性也逐渐提高.

发现 6. 在 DL 编译器中, 挂起和性能问题虽占比不高, 但增长迅速, 数量已增长了 1.56 倍. 因此, 设计针对此类缺陷的检测方法变得愈发重要.

4.3 RQ3: 位置

表 2 中展现了不同时期 DL 编译器在各阶段缺陷数量占比. 其中旧数据为已有工作[8]中使用的 603 个早期 DL 编译器缺陷数据. 从表中可以发现, 和传统编译器缺陷分布不同, DL 编译器在 3 个阶段均较多的缺陷占比. 与传统编译器前端不同, DL 编译器需要处理来自各种 DL 框架的模型, 并将每个 DL 框架中支持的 DL 算子转换成功能等价的 High-level IR. 由于 DL 框架中的算子也在快速迭代和发展, DL 编译器需要及时且准确地适配 DL 框架中各种 DL 算子的修改逻辑. 不及时或不正确的算子转换逻辑更新以适配 DL 框架最新的算子实现将会导致 DL 编译器前端出现新的缺陷. 需要注意的是, 由于 AKG 编译器缺少模型加载阶段, 其模型加载功能在另一前置系统 MindSpore 中完成. 因此, AKG 并没有模型加载阶段.

表 2 不同 DL 编译器各阶段缺陷数量占比 (%)

DL编译器	数据来源	模型加载	High-level IR转换	Low-level IR转换
TVM	旧数据	25.50	38.26	36.24
	本工作	14.65	30.81	54.55
Glow	旧数据	27.34	35.16	37.50
	本工作	44.30	26.58	29.11
nGraph	旧数据	4.27	73.50	22.22
AKG	本工作	—	28.74	71.26
合计	旧数据	21.36	45.12	33.52
	本工作	**16.55**	**29.89**	**53.56**

从整体上来看, Low-level IR 转换阶段的缺陷超过 High-level IR 转换阶段, 然而不同 DL 编译器之间缺陷位置的分布有着显著的差异, 因此, 需要对每个编译器分别研究. 一方面, 本文研究的 AKG 的 3 个核心功能: 规范化、自动调度和后端优化中的后两个均是围绕 Low-level IR 设计并实现的. 另一方面, 已有研究工作[8]中的 nGraph 则是围绕计算图优化 (即 High-level IR 优化) 进行设计实现的. 对于 TVM 来说, 相较于模型加载阶段和 High-level IR 转换阶段, Low-level IR 转换阶段的缺陷比例从之前的 36.24% 升至 54.55%. 其主要原因是 TVM 的 Low-level IR 转换阶段是一个极其复杂的任务. 同时, 相较于 High-level IR 转换阶段通用的图级别优化算法, Low-level IR 转换阶段的优化任务需要结合特定的硬件特性, 从而需要不断地修改优化代码以提升底层算子原语在硬件上的执行效率, 复杂的优化任务和适配各种硬件特性是导致 Low-level IR 转换阶段缺陷比例不断上升的一个主要原因.

发现 7. DL 编译器缺陷分布紧密关联于其架构设计. 其中, TVM 在 Low-level IR 转换阶段的缺陷比例快速上升, 归因于该阶段频繁的代码变更所引入的新缺陷.

4.4 RQ4: 根因-症状-位置之间的内在联系

4.4.1 根因-症状之间的内在联系

表 3 展示了缺陷在不同症状和根因下的数量. 作为最普遍的缺陷根因, 逻辑错误最主要的症状是结果错误, 占比 57.63%. 与崩溃缺陷不同, 结果错误类别的缺陷只有在特定的测试预言才能被捕获. 因此, 针对代码逻辑错误, 特别是优化逻辑错误设计相应的缺陷检测方法并设计细粒度的测试预言以捕获更多此类缺陷依然存在挑战. 对于张量形状错误、API 误用、类型问题和异常处理错误这 4 类较为常见的缺陷类型, 他们最主要的症状为崩溃. 对于这几类缺陷, 崩溃仍然是一种简单且有效的测试预言.

表 3 缺陷在不同症状与根因下的数量分布

根因	崩溃	结果错误	构建失败	性能问题	未报告	挂起	总计
逻辑错误	41	69	1	5	1	2	119
张量形状问题	73	8	0	3	2	0	86
API误用	42	18	1	10	6	2	79
类型问题	50	18	1	1	4	1	75
异常处理错误	51	10	1	3	1	3	69
配置错误	4	1	29	3	2	1	40
赋值错误	20	10	1	0	4	0	35
并发问题	14	9	1	2	2	2	30
不兼容	17	2	4	1	1	0	25
数值计算错误	10	13	0	0	1	0	24
注册错误	10	1	2	0	0	1	14
拼写错误	6	1	1	0	1	0	9
其他	2	5	0	1	0	0	8
总计	**340**	**165**	**42**	**29**	**25**	**12**	**613**

此外, 基于缺陷症状分析其与根因的内在联系有助于开发者快速识别缺陷的根因、对缺陷进行定位, 从而减少缺陷调试时长. 从表 3 可以看出, 构建失败缺陷的根因主要为配置错误和不兼容问题. 这类缺陷主要由于配置文件存在的错误引起的, 本文提供的数据集可以促进构建失败自动修复[42]的未来研究. 性能问题和挂起是两种调试, 定位难度极高的缺陷类别. 这两类缺陷的根因主要和逻辑错误、API 误用和异常处理错误相关. 这些发现有助于加速开发者对缺陷的调试与定位过程.

发现 8. 在 DL 编译器中, 逻辑错误更容易引起错误的编译结果; 而张量形状错误、API 误用、类型问题和异常处理错误常致编译过程崩溃. 构建失败多源于配置错误和不兼容问题; 性能下降和挂起错误则多由逻辑错误、API 误用和异常处理不当所致.

4.4.2 根因-位置之间的内在联系

表 4 展示了 DL 编译器缺陷在不同根因和位置下的数量分布. 由于存在部分缺陷无法分类到 DL 编译器 3 个阶段中的任何一个阶段 (比如, 编译安装 DL 编译器所使用的配置文件中的缺陷), 因此关于缺陷位置分布仅分析可以分类到这 3 个阶段的缺陷. 由表可知, 逻辑错误出现在 Low-level IR 转换阶段的数量显著高于模型加载和 High-level IR 转换阶段. Low-level 转换阶段比前两个阶段有着更为复杂的代码逻辑, 尤其是优化相关的代码逻辑. 具体来说, 模型加载阶段实现了将 DL 框架中算子运算逻辑转换成 High-level IR, 由于存在 DL 框架中算子实现的代码逻辑作为参考, 所以此阶段的代码逻辑实现起来较为简单. 同时, 相比较 High-level IR 转换阶段抽象程度更高的代码逻辑, Low-level IR 转换阶段涉及更底层、更具体的操作, 例如寄存器分配、指令选择和优化等. 这些细粒度的硬件特性相关信息提高了 Low-level IR 转换阶段的逻辑难度, 从而更容易引入逻辑错误.

此外, 从表 4 中可知, 90% 的并发缺陷都发生在 Low-level IR 转换阶段. 目前对 low-level 转换阶段的测试工作仅有 Tzer[37]一项, 其并没有检测到任何并发缺陷, 因此我们需要在测试 Low-level IR 转换过程中考虑到并发可能引起的缺陷. 此外并发缺陷触发条件复杂度高, 复现难度大, 与运行时环境关系密切, 因此需要针对并发问题设

定特定的检测方法.

表 4 缺陷在不同根因与位置下的数量分布

根因	模型加载	High-level IR转换	Low-level IR转换	总计
逻辑错误	18	33	67	118
张量形状问题	24	48	14	86
API误用	12	18	48	78
类型问题	11	30	34	75
异常处理错误	11	11	43	65
赋值错误	4	4	24	32
并发问题	0	3	27	30
不兼容	6	3	14	23
数值计算错误	4	6	13	23
注册错误	2	5	7	14
其他	0	2	6	8
拼写错误	1	4	3	8
配置错误	0	1	1	2
总计	**93**	**168**	**301**	**562**

发现 9. DL 编译器的逻辑错误倾向于出现在 Low-level IR 转换阶段. 此外, 并发问题均发生在 Low-level IR 转换阶段, 且现有方法均未能识别出任何并发缺陷.

4.4.3 症状-位置之间的内在联系

表 5 展示了不同阶段缺陷的症状. 从该表中我们可以看出, 模型加载阶段中有 73.12% 的缺陷会导致程序崩溃, 显著高于 High-level IR 和 Low-level IR 转换阶段的崩溃类型缺陷. 因此, 针对每个前端 DL 框架构造语法多样性强的测试用例将是一种检测模型加载阶段缺陷潜在可行的方案. 语法多样性强指的是测试用例应该尽可能覆盖各种不同的算子和参数配置, 以更好地覆盖模型加载阶段的各种边界情况.

表 5 缺陷在不同症状与位置下的数量分布

症状	模型加载	High-level IR转换	Low-level IR转换	总计
崩溃	68	103	160	331
结果错误	23	55	81	159
性能问题	0	2	24	26
未报告	2	5	16	23
构建失败	0	2	10	12
挂起	0	1	10	11
总计	**93**	**168**	**301**	**562**

此外, 37 个性能相关缺陷 (包括 26 个性能问题和 11 个挂起) 中有 34 个出现在 Low-level IR 转换阶段, 占比高达 91.89%. 进一步分析发现, 其中有 23 个和内存优化相关. 由于 Low-level IR 转换阶段中访存开销隐藏是一个重要的优化功能, AKG 和 TVM 的自动调优算法都涉及针对内存的优化操作. 因此, 开发者在修改自动调优组件的时候应该格外注意是否会导致性能下降或挂起的问题. 测试人员也需要重点关注自动调优组件性能问题和挂起类型的缺陷.

发现 10. 崩溃是 DL 编译器模型加载阶段缺陷最常见的症状. 同时, 91.89% 的性能相关缺陷出现在 Low-level IR 转换阶段. 因此, 针对性能缺陷设计的测试方法应考虑 Low-level IR 转换阶段的功能特性.

4.5 RQ5: 回归测试用例与已有测试用例的差异

在修复缺陷时, 开发者通常会使用触发该缺陷的测试用例作为回归测试用例, 并将其加入项目自带的测试套件中. 具体来说, 如果已有的测试套件未能覆盖缺陷相关的源码, 开发者将会在已有测试套件中添加一个全新的回

归测试用例. 另一方面, 如果测试套件能够覆盖到缺陷相关的源码但是缺少触发缺陷的关键因素, 那么开发者将会通过修改已有的测试用例使其能触发该缺陷作为回归测试用例, 以期最小程度地修改测试套件. 因此, 研究回归测试用例修改程度有助于理解触发缺陷测试用例和已有测试套件的关联程度.

由于存在一些修复缺陷的 PR 并不包含相应的回归测试用例, 或者回归测试用例并没有与修复缺陷的补丁在同一 PR 中提交且缺少关联. 为此, 本文分析仅与缺陷补丁同时提交的 286 个回归测试用例. 具体来说, 本文首先判断回归测试用例属于新增的还是基于已有测试套件修改的. 接着, 对于基于修改的测试用例, 本文进一步统计了回归测试用例涉及的变更行数, 变更行数等于删除行数与新增行数的总和. 需要注意的是, 如果将一行代码中部分内容做了修改, 则等价于将原始错误的代码行删除, 并新增一行正确的代码. 统计结果显示, 在研究的 286 个回归测试用例中, 170 个是完全新增的, 116 个是基于已有的测试用例修改而成的. 完全新增的回归测试用例占比高达 59.44%, 这表明 DL 编译器中存在较多的功能点没有被开发人员手写的测试套件覆盖到. 因此, 使用源码覆盖指导的测试用例生成方法是一种潜在有效的缺陷检测方法. 并且已有的测试技术 (比如, HirGen、NNSmith) 都没有使用源码覆盖来指导测试用例生成.

除此之外, 本文进一步统计了 116 个基于修改的回归测试用例的修改行数, 其修改行数分布如表 6 所示. 从表可知, 50% 基于修改的回归测试用例涉及的修改行数小于 10 行. 由于该代码修改行数可能是对多个测试用例同时做的修改. 因此, 通常情况下触发缺陷的测试用例仅需要修改其中一个测试用例即可. 也就是说, 多数情况下, 设计变异算子对测试套件进行轻微修改即可触发此类缺陷. 因此, 使用开发者手写的测试套件作为种子, 并设计针对 DL 编译器缺陷的变异算子是另外一种潜在可行的缺陷检测方法.

表 6　测试用例修改的行数

测试用例修改行数	对应缺陷数量
[1, 5]	42
[6, 10]	16
[11, 15]	9
[16, 20]	11
[21, 25]	5
[26, +∞]	33

发现 11. 59.44% 的回归测试用例是完全新增的, 表明提升源码覆盖能力是设计 DL 编译器缺陷检测方法的一个重要指导方案. 同时, 50% 的修改回归测试用例涉及不到 10 行代码变动, 暗示通过变异已有测试套件生成新的测试用例是另一种可行的缺陷检测方案.

4.6 RQ6: 补丁大小

研究补丁的大小对指导程序的规范化开发和缺陷的自动化修复有较好的指导作用. 为此, 本文从修复缺陷的 PR 中收集每个缺陷对应补丁所涉及函数的个数. 具体来说, 我们使用了 libClang[43]和基于模板的解析规则对代码变更所涉及的源码文件进行解析以获取源码文件中的函数分布情况, 即每个函数在文件中的起始行与结束行. 接着, 通过行号与相应文件的函数分布进行比对获取每个缺陷修复所使用的补丁中修改的函数数量. 需要注意的是, 这种自动解析方法无法处理一个 PR 中包含多个缺陷修复的补丁. 因此, 本文通过人工分析的方式以区分一个 PR 中修复不同缺陷的补丁.

补丁所涉及的函数数量的分布如表 7 所示. 由表可知, 78.30% 的缺陷修复补丁涉及的函数数量在 3 个及以下. 具体来说, 修改了一个函数的补丁占补丁总数目的 46.82%, 所占的比例最高, 而修改了 2 个函数和 3 个函数的补丁数目则分别占 15.33% 和 7.34%. 另外, 补丁涉及函数的个数为 0 的缺陷中, 38.89% 的缺陷根因为配置问题. 这表明补丁不会做出过于复杂的修改, 一般来说补丁只涉及较少部分的修改, 这也与传统软件的缺陷特征相同. 此外, 对于不同的 DL 编译器, 修复缺陷的补丁大小分布也有所不同, 具体来说, 在 AKG 与 Glow 中补丁涉及的函数个数小于或等于 4 的比例较高, 分别达到 35.87% 和 29.76%. 本工作对这些补丁进行了分析, 发现它们往往会在多个不同的函数中进行逻辑相似的修改, 因此尽管这类补丁修改的函数数量较多, 却大多为可以避免的冗余修改. 这

也提醒开发者增强代码复用性以避免修复缺陷时修改过多函数造成冗余修改和维护负担.

表 7 补丁涉及的函数数量占比 (%)

编译器	0	1	2	3	≥4
AKG	0.00	41.30	16.30	6.52	35.87
Glow	2.38	35.71	22.62	9.52	29.76
TVM	11.90	50.11	13.73	7.09	17.16
合计	8.81	46.82	15.33	7.34	21.70

发现 12. 与传统编译器特征相似, 高达 78.30% 的补丁仅针对 3 个或更少的函数进行修改. 且常出现跨多个函数的相似修改模式.

5 启发与应用

5.1 对后续研究工作的启发

基于本文的研究发现, 我们进一步为将来 DL 编译器的缺陷检测、定位和修复等研究方向均提供了一系列可行的指导建议.

首先, 本文识别出一些针对 DL 编译器的缺陷检测任务亟待解决的研究问题并提供了一些潜在的解决方案. 例如, 发现 2、8 和 9 指出逻辑错误, 特别是优化相关的逻辑错误已上升为最普遍的缺陷类型, 目前的测试用例生成方法和测试预言构造方法 (比如, NNSmith、HirGen 和 MT-DLComp) 均存在检测逻辑错误缺陷效率低下的问题. 如何设计能够生成包含优化敏感结构的测试输入并设计细粒度的测试预言以捕获优化逻辑错误类型的缺陷十分重要. 考虑到逻辑错误更容易出现在 Low-level IR 转换阶段, 针对 Low-level IR 设计细粒度的测试预言或许能进一步提高已有测试方法捕获缺陷的能力. 发现 4 指出张量类型错误占比显著下降, 表明类型相关的数据结构和代码逐步完善. 随着类型系统的成熟, 类型缺陷的检测方法应进一步聚焦于识别和处理复杂或隐蔽的类型错误. 例如, 缺陷检测工具可以集成更精确的静态类型分析功能, 自动识别类型不匹配和隐性转换错误. 针对类型相关缺陷的调试方法应更依赖自动化推理, 尤其是增强形状推理机制, 使得在编译阶段即可发现潜在的类型问题. 发现 6 和发现 10 指出深度学习编译器的性能缺陷占比增长迅速, 并且已有的自动化缺陷测试技术 (比如, HirGen、NNSmith 和 Tzer) 均没有检测到此类缺陷. 因此, 设计性能缺陷的检测方法愈发重要. 由于性能问题隐蔽且与硬件特性密切相关, 未来应重点研究基于硬件特性的测试工具, 通过结合静态和动态分析, 提升自动化检测和调试性能问题的能力, 从而减少此类缺陷的影响. 此外, 考虑到超过 62% 的此类缺陷和访存有关, 设计更多内存操作相关的变异算子或许也可以检测到更多此类的缺陷.

发现 11 指出, 59.44% 的缺陷回归测试用例是全新编写的, 即与已有的开发者测试用例没有强关联, 这反映出已有测试用例难以覆盖触发此类缺陷所需覆盖的源码. 因此, 提高源代码的覆盖率是一个潜在的缺陷检测策略. 另一方面, 在基于修改的回归测试用例中, 50% 的回归测试用例修改通常涉及不到 10 行代码. 这一发现表明, 基于 DL 编译器的缺陷特征设计相应的变异算子 (例如, 张量形状或类型变异), 并对现有开发者测试用例进行变异以生成更多新的测试用例, 也是一个潜在有效的缺陷检测策略. 发现 2 和发现 3 表明, DL 编译器中新引入的特性更容易引入缺陷, 比如 TVM 中有 25.29% 的张量形状错误缺陷和新增的动态张量形状推理功能相关, 因此, 在代码持续集成的时候, 对修改变更部分执行静态缺陷分析与缺陷预测将有助于及时的检测到代码变更中引入的缺陷, 减少缺陷的影响, 从而提高 DL 编译器整体的代码质量.

其次, 本文的研究发现可以辅助开发人员对缺陷进行定位与调试. 例如, 发现 1 指出, 尽管深度学习编译器不断引入新功能与特性, 但其缺陷根因的类别已趋于稳定. 因此, 自动化识别缺陷根因是一个潜在可行的研究方向. 调试人员可以基于已知的根因类别, 优先排查常见高频问题, 并结合智能分析工具, 更快速、准确地定位和修复缺陷, 从而进一步提升调试效率. 此外, 发现 8 指出, 构建失败的根因主要是配置错误和不兼容问题, 性能问题和挂起错误主要由逻辑错误, API 误用和异常处理错误引起等, 这些症状与根因的内在联系有助于开发人员快速识别缺

陷的根因以辅助其执行缺陷调试. 发现 9 和发现 10 指出逻辑错误和性能相关的缺陷发生在 Low-level IR 转换阶段的概率远大于其他两个阶段, 这对开发者定位缺陷的具体位置有很大的引导作用, 同时也能促进自动缺陷定位方法的设计. 具体来说, 在进行缺陷排查时, 开发者可以优先关注 Low-level IR 转换阶段的代码, 集中精力检查该阶段潜在的问题, 从而提高缺陷定位的效率. 这不仅减少了排查的工作量, 还可以提升项目的整体开发效率.

最后, 本文的研究发现对缺陷的自动修复技术有一定的指导作用. 比如, 发现 12 指出 78.30% 缺陷修复的补丁涉及少于或等于 3 个函数. 这表明大部分的 DL 编译器缺陷可以使用自动化修复技术进行修复. 同时, 本文发现多个不同函数中存在相似的补丁修改情况. 因此, 使用克隆检测技术识别具有相似上下文的函数, 并从中提取部分代码片段辅助设计自动化修复技术或许是一种可行的方案.

5.2 验证性应用

为了验证本文的研究发现是否切实可用, 我们基于本文研究中的两点发现设计了一种简单的 DL 编译器测试工具——CfgFuzz. 一方面, DL 编译器中与优化相关的缺陷占比多且增长显著 (发现 2). 另一方面, 张量形状错误的因为动态张量形状推理特性的而引入而有所增加 (发现 3). 为此, 我们设计一种基于优化组合配置指导的模糊测试方法 CfgFuzz, 并通过虚拟机编译方式实现对 DL 编译器中张量动态推理特性的功能覆盖检测. 需要注意的是, 由于 CfgFuzz 工具仅用于验证本文研究发现是否切实可行性和潜在价值, 因此 CfgFuzz 仅采用了朴素的设计方案. 具体来说, 考虑到 DL 编译器中优化操作设计到 High-level IR 转换阶段图层面的优化和 Low-level IR 转换中硬件相关的优化, 我们设计了一种细粒度的优化组合配置方法, 以充分地探索优化逻辑代码. DL 编译器的优化配置选项包括 High-level IR 转换阶段粗粒度的优化等级, 细粒度的优化 Pass 组合和 Low-level IR 转换阶段优化目标设备与自动调优算法相应的一系列配置选项 (包括自动算法选择、优化器、执行次数等) 等. 为此, CfgFuzz 首先构造了一个 DL 编译器优化选项配置的搜索空间. 具体来说, 优化等级包含 0–4 共 5 个级别, 优化 Pass 组合包含 25 个细粒度的优化算子 (如 FuseOPs), 优化目标设备包含 LLVM 和 CUDA 两种目标语言, 自动调优算法的执行次数范围在本文中设置为 0–20 次. 需要注意, 优化等级为 0 表示仅支持最基本的优化操作; 自动调优次数为 0 表示编译过程中关闭自动调优的功能. 为了更全面地探索不同类别优化组合的关联关系, CfgFuzz 采用组合测试的方法, 在每次测试中从优化配置空间中随机选择一个具体的优化选项, 以深入探索 DL 编译器的不同优化操作与优化组合.

为了验证 CfgFuzz 的有效性, 我们选择最流行的 DL 编译器 TVM 作为待测软件执行测试任务, 并使用 LEMON[44]工具作为测试用例的种子生成工具. 由于发现 5 指出崩溃和结果错误是 DL 编译器中最常见的两类缺陷症状, 我们采用编译是否成功和差分测试作为测试预言. 具体来说, 对于同一模型输入, 如果编译前的 DL 模型与 DL 编译器编译后的 DL 模型在相同的输入输出的预测结果存在显著不同 (即预测结果之间的切比雪夫距离大于 1E–3)[37], 或者 DL 编译器在编译过程中发生程序崩溃, 则认为 CfgFuzz 检测到了一个缺陷. 在编译过程中, 我们使用 TVM 的虚拟机编译方式 (即 *relay.vm.compile*) 实现模型的编译任务实现对 TVM 动态张量形状推理功能的覆盖. 在 24 h 的测试过程中, CfgFuzz 共检测到 8 个缺陷, 其中 7 个缺陷已被开发者确认. 检测到的缺陷详细信息见表 8.

表 8 CfgFuzz 检测到的缺陷

序号	症状	根因	阶段	缺陷状态
1	结果错误	优化相关逻辑错误	Low-level IR转换	已修复
2	崩溃	张量形状错误	High-level IR转换	已确认
3	结果错误	张量形状错误	Low-level IR转换	已确认
4	崩溃	—	—	等待确认
5	结果错误	优化相关逻辑错误	High-level IR转换	已确认
6	崩溃	异常处理错误	High-level IR转换	已确认
7	崩溃	张量形状错误	High-level IR转换	已修复
8	崩溃	张量形状错误	模型加载阶段	已修复

经过分析发现, CfgFuzz 检测到的 8 个缺陷中, 有 3 个缺陷是和优化相关的逻辑错误缺陷, 4 个张量形状错误类型缺陷, 1 个异常处理错误的缺陷, 1 个未知类型的缺陷. CfgFuzz 的检测缺陷的高效性进一步验证了本文关于 DL 编译器缺陷的实证研究所挖掘出来的发现和启示是切实可行的.

图 5 展示了一个 CfgFuzz 检测到的 TVM 缺陷. 在配置选项包含 AlterOpLayout 的优化算子且编译目标平台为 CUDA 的时候, TVM 在编译 vgg16-cifar10 模型的时候会发生会崩溃. 由于 AlterOpLayout 优化算子错误地就地修改 IR 的结构, 而不是在基于拷贝的 IR 做修改, 从而导致模型不等价的转换. 为此, 开发者通过在使用该优化前对模型结构进行深拷贝的方法修复了该缺陷.

```
-  opt_mod, _ = relay.optimize(mod, target, params)
+  mod_clone = deepcopy(mod)
+  opt_mod, _ = relay.optimize(mod_clone, target, params)
+  mod_clone = deepcopy(mod)
```

图 5 一个 CfgFuzz 检测到的缺陷 (TVM# Issue-7979)

6 研究效度的威胁

本节将讨论本文研究方法与数据标注过程中可能遭遇的研究效度威胁. 研究效度威胁是指在研究过程中, 那些可能对研究结果的准确性、可靠性和泛化性构成潜在威胁的因素. 这些威胁因素通常可以分为外部效度威胁和内部效度威胁两大类.

关于外部效度威胁, 首要关注点是数据集的质量. 本研究参考了现有缺陷研究方法, 系统地搜集了 3 款主流 DL 编译器中的 613 个缺陷案例, 这些案例均源于已被合并到项目主分支的拉取请求 (PR) 和通过关键词搜索识别的缺陷修复 PR. 因此, 本文收集的数据集的质量可靠性较强. 作为 DL 编译器缺陷领域的一项重要研究, 本文显著提升了研究的普遍性和代表性. 此外, 由于 DL 编译器不断公布新修复的缺陷, 本研究无法持续跟进所有新修复缺陷会构成了外部效度的一个潜在威胁, 可能影响研究发现的实用性时效. 为了降低这一威胁, 本文在缺陷时间段选取上, 收集了较长时间跨度的缺陷数据集, 并再研究发现中指出 DL 编译器缺陷会随时间发生改变的因素 (比如: 不同类别缺陷的分布) 和不会随时间发生改变的因素 (缺陷根因和症状的类别). 同时, 本文进一步指出随时间发生改变的因素主要的引起原因. 这些研究发现对 DL 编译器后续的研究有着长期的指导意义.

在内部效度方面, 主要挑战在于手动标记过程可能存在的主观性和误差. 为了减轻这些潜在威胁, 本研究采用了双重核查机制. 两位具有丰富开发经验的作者依据通用的开放编码方案[20], 独立进行数据标记工作. 通过计算 Cohen's Kappa 系数验证标记一致性, 结果显示评分者间一致性高达 95%, 这充分证明了标记过程的高度可靠性. 在出现标记不一致时, 会邀请第 3 位资深开发人员参与讨论, 直至达成一致意见, 从而进一步提高标签的可靠性和准确性. 此外, 在标注培训阶段, 本研究还引入了第三方的视角来讨论不一致之处, 以降低因主观判断错误而导致的误判风险. 未来, 我们将考虑采用自动化的标注方式来消除人工标注存在的主观性影响. 训练一个针对缺陷特征的分类模型和使用大语言模型[45]实现自动化标注均为潜在的替代方案, 尤其是借助大语言模型多智能技术[46]实现自动化标注是一个潜在的高效且有效的方法.

7 总 结

为了深入研究当下 DL 编译器缺陷特征及其随时间的演变情况, 本文对当前主流 3 款 DL 编译器 (即 Apache 的 TVM、Facebook 的 Glow 和华为的 AKG) 中 613 个近期修复的缺陷进行深入分析. 通过从缺陷的根因、症状与位置的演变情况, 根因、症状、位置三者之间的内在联系, 触发缺陷的回归测试用例和修复缺陷的补丁特征等角度深入分析 DL 编译器缺陷的现状, 总结了 12 个主要发现. 在此基础上, 我们为未来的 DL 编译器缺陷检测和调试研究提供了一系列可操作的启示. 最后, 基于本文的研究发现与启示, 我们开发了一款轻量级的基于优化配置的测试工具 CfgFuzz. CfgFuzz 通过对编译配置选项 (如优化选项和目标硬件平台选择) 进行组合测试, 成功检测出

了 8 个 TVM 缺陷, 其中 7 个已被开发人员确认或修复. CfgFuzz 的缺陷检测能力进一步证实了本文研究发现的有效性.

References:

[1] Chen CY, Seff A, Kornhauser A, Xiao JX. DeepDriving: Learning affordance for direct perception in autonomous driving. In: Proc. of the 2015 IEEE Int'l Conf. on Computer Vision. Santiago: IEEE, 2015. 2722–2730. [doi: 10.1109/ICCV.2015.312]

[2] Sun Y, Chen YH, Wang XG, Tang XO. Deep learning face representation by joint identification-verification. In: Proc. of the 28th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2014. 1988–1996.

[3] Cao KB, Chen CY, Baltes S, Treude C, Chen X. Automated query reformulation for efficient search based on query logs from stack overflow. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 1273–1285. [doi: 10.1109/ICSE43902.2021.00116]

[4] Chen X, Chen CY, Zhang D, Xing ZC. SEthesaurus: WordNet in software engineering. IEEE Trans. on Software Engineering, 2021, 47(9): 1960–1979. [doi: 10.1109/TSE.2019.2940439]

[5] Watson C, Cooper N, Palacio DN, Moran K, Poshyvanyk D. A systematic literature review on the use of deep learning in software engineering research. ACM Trans. on Software Engineering and Methodology (TOSEM), 2022, 31(2): 32. [doi: 10.1145/3485275]

[6] Yang L, Chen JJ, Wang Z, Wang WJ, Jiang JJ, Dong XY, Zhang WB. Semi-supervised log-based anomaly detection via probabilistic label estimation. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 1448–1460. [doi: 10.1109/ICSE43902.2021.00130]

[7] Li MZ, Liu Y, Liu XY, Sun QX, You X, Yang HL, Luan ZZ, Gan L, Yang GW, Qian DP. The deep learning compiler: A comprehensive survey. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(3): 708–727. [doi: 10.1109/TPDS.2020.3030548]

[8] Shen QC, Ma HY, Chen JJ, Tian YQ, Chung SC, Chen X. A comprehensive study of deep learning compiler bugs. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 968–980. [doi: 10.1145/3468264.3468591]

[9] Chen TQ, Moreau T, Jiang Z, Zheng LM, Yan E, Cowan M, Shen HC, Wang LY, Hu YW, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 579–594.

[10] Rotem N, Fix J, Abdulrasool S, Catron G, Deng S, Dzhabarov R, Gibson N, Hegeman J, Lele M, Levenstein R, Montgomery J, Maher B, Nadathur S, Olesen J, Park J, Rakhov A, Smelyanskiy M, Wang M. Glow: Graph lowering compiler techniques for neural networks. arXiv:1805.00907, 2019.

[11] Cyphers S, Bansal AK, Bhiwandiwalla A, Bobba J, Brookhart M, Chakraborty A, Constable W, Convey C, Cook L, Kanawi O, Kimball R, Knight J, Korovaiko N, Kumar V, Lao YX, Lishka CR, Menon J, Myers J, Narayana SA, Procter A, Webb TJ. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. arXiv:1801.08058, 2018.

[12] Xiao DW, Liu ZB, Yuan YY, Pang Q, Wang S. Metamorphic testing of deep learning compilers. In: Proc. of the 2022 ACM SIGMETRICS/IFIP PERFORMANCE Joint Int'l Conf. on Measurement and Modeling of Computer Systems. Mumbai: ACM, 2022. 65–66. [doi: 10.1145/3489048.3522655]

[13] Ma HY, Shen QC, Tian YQ, Chen JJ, Cheung SC. Fuzzing deep learning compilers with HirGen. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 248–260. [doi: 10.1145/3597926.3598053]

[14] Zhao J, Li BJ, Nie W, Geng Z, Zhang RW, Gao X, Cheng B, Wu C, Cheng Y, Li Z, Di P, Zhang K, Jin XF. AKG: Automatic kernel generation for neural processing units using polyhedral transformations. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 1233–1248. [doi: 10.1145/3453483.3454106]

[15] Andriyanov NA. Analysis of the acceleration of neural networks inference on intel processors based on openVINO toolkit. In: Proc. of the 2020 Systems of Signal Synchronization, Generating and Processing in Telecommunications. Svetlogorsk: IEEE, 2020. 1–5. [doi: 10.1109/SYNCHROINFO49631.2020.9166067]

[16] Sun CN, Le V, Zhang QR, Su ZD. Toward understanding compiler bugs in GCC and LLVM. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 294–305. [doi: 10.1145/2931037.2931074]

[17] Zhou ZD, Ren ZL, Gao GJ, Jiang H. An empirical study of optimization bugs in GCC and LLVM. Journal of Systems and Software, 2021, 174: 110884. [doi: 10.1016/j.jss.2020.110884]

[18] Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P. Taxonomy of real faults in deep learning systems. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 1110–1121. [doi: 10.1145/3377811.3380395]

[19] Garcia J, Feng Y, Shen JJ, Almanee S, Xia Y, Chen QA. A comprehensive study of autonomous vehicle bugs. In: Proc. of the 42nd ACM/IEEE Int'l Conf. on Software Engineering. Seoul: ACM, 2020. 385–396. [doi: 10.1145/3377811.3380397]

[20] Chen JJ, Liang YH, Shen QC, Jiang JJ, Li SC. Toward understanding deep learning framework bugs. ACM Trans. on Software Engineering and Methodology, 2023, 32(6): 135. [doi: 10.1145/3587155]

[21] Islam J, Nguyen G, Pan R, Rajan H. A comprehensive study on deep learning bug characteristics. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 510–520. [doi: 10.1145/3338906.3338955]

[22] Zhang YH, Chen YF, Cheung SC, Xiong YF, Zhang L. An empirical study on TensorFlow program bugs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Amsterdam: ACM, 2018. 129–140. [doi: 10.1145/3213846.3213866]

[23] Abadi M, Agarwal A, Barham P, *et al*. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. arXiv:1603.04467, 2016.

[24] Lu S, Park S, Seo E, and Zhou YY. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Washington: ACM, 2008. 329–339. [doi: 10.1145/1346281.1346323]

[25] Di Franco A, Guo H, Rubio-González C. A comprehensive study of real-world numerical bug characteristics. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering. Urbana: IEEE, 2017. 509–519. [doi: 10.1109/ASE.2017.8115662]

[26] Han X, Yu TT. An empirical study on performance bugs for highly configurable software systems. In: Proc. of the 10th ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. Ciudad Real: ACM, 2016. 23. [doi: 10.1145/2961111.2962602]

[27] Wan ZY, Lo D, Xia X, Cai L. Bug characteristics in blockchain systems: A large-scale empirical study. In: Proc. of the 14th IEEE/ACM Int'l Conf. on Mining Software Repositories. Buenos Aires: IEEE, 2017. 413–424. [doi: 10.1109/MSR.2017.59]

[28] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin ZM, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai JJ, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In: Proc. of the 33rd Conf. on Neural Information Processing Systems. Vancouver: ACM, 2019. 32.

[29] Moolayil J. Learn Keras for Deep Neural Networks. Berkeley: Apress, 2019. [doi: 10.1007/978-1-4842-4240-7]

[30] Tong ZH, Du N, Song XB, Wang XL. Study on MindSpore deep learning framework. In: Proc. of the 17th Int'l Conf. on Computational Intelligence and Security. Chengdu: IEEE, 2021. 183–186. [doi: 10.1109/CIS54983.2021.00046]

[31] Li ZM, Tan L, Wang XH, Lu S, Zhou YY, Zhai CX. Have things changed now? An empirical study of bug characteristics in modern open source software. In: Proc. of the 1st Workshop on Architectural and System Support for Improving Software Dependability. San Jose: ACM, 2006. 25–33. [doi: 10.1145/1181309.1181314]

[32] Ma HY, Zhang WQ, Shen QC, Tian YQ, Chen JJ, Cheung SC. Towards understanding the bugs in solidity compiler. In: Proc. of the 33rd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Vienna: ACM, 2024. 1312–1324. [doi: 10.1145/3650212.3680362]

[33] Nikanjam A, Morovati MM, Khomh F, Ben Braiek H. Faults in deep reinforcement learning programs: A taxonomy and a detection approach. Automated Software Engineering, 2022, 29(1): 8. [doi: 10.1007/s10515-021-00313-x]

[34] Shull F, Godfrey S, Bechtel A, Feldmann RL, Regardie M, Seaman C. Making use of a decade of widely varying historical data: SARP Project—"full life-cycle defect management". Fraunhofer USA Inc., 2008.

[35] Tan L, Liu C, Li ZM, Wang XH, Zhou YY, Zhai CX. Bug characteristics in open source software. Empirical Software Engineering, 2014, 19(6): 1665–1705. [doi: 10.1007/s10664-013-9258-8]

[36] Thung F, Wang SW, Lo D, Jiang LX. An empirical study of bugs in machine learning systems. In: Proc. of the 23rd IEEE Int'l Symp. on Software Reliability Engineering. Dallas: IEEE, 2012. 271–280. [doi: 10.1109/ISSRE.2012.22]

[37] Vieira SM, Kaymak U, Sousa JMC. Cohen's Kappa coefficient as a performance measure for feature selection. In: Proc. of the 2010 Int'l Conf. on Fuzzy Systems. Barcelona: IEEE, 2010. 1–8. [doi: 10.1109/FUZZY.2010.5584447]

[38] Yang CY, Deng YL, Lu RY, Yao JY, Liu JW, Jabbarvand R, Zhang LM. WhiteFox: White-box compiler fuzzing empowered by large language models. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA2): 296. [doi: 10.1145/3689736]

[39] Shen QC, Tian YQ, Ma HY, Chen JJ, Huang LL, Fu RF, Cheung SC, Wang Z. A tale of two DL cities: When library tests meet compiler. arXiv:2407.16626, 2024.

[40] Liu JW, Wei YX, Yang S, Deng YL, Zhang LM. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA1): 73. [doi: 10.1145/3527317]

[41] Liu JW, Lin JK, Ruffy F, Tan C, Li JY, Panda A, Zhang LM. NNSmith: Generating diverse and valid test cases for deep learning compilers. In: Proc. of the 28th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Vancouver: ACM, 2023. 530–543. [doi: 10.1145/3575693.3575707]

[42] Lou YL, Chen JJ, Zhang LM, Hao D, Zhang L. History-driven build failure fixing: How far are we? In: Proc. of the 28th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Beijing: ACM, 2019. 43–54. [DOI: 10.1145/3293882.3330578]

[43] Schaub S, Malloy BA. Comprehensive analysis of C++ applications using the libClang API. In: Proc. of the 23rd Int'l Conf. on Software Engineering and Data Engineering. New Orleans: Int'l Society for Computers and Their Applications, 2014. 131–136.

[44] Wang Z, Yan M, Chen JJ, Liu S, Zhang DD. Deep learning library testing via effective model generation. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 788–799. [doi: 10.1145/3368089.3409761]

[45] Hou XY, Zhao YJ, Liu Y, Yang Z, Wang KL, Li L, Luo XP, Lo D, Grundy J, Wang HY. Large language models for software engineering: A systematic literature review. ACM Trans. on Software Engineering and Methodology, 2024, 33(8): 220. [doi: 10.1145/3695988]

[46] Guo TC, Chen XY, Wang YQ, Chang RD, Pei SC, Chawla NV, Wiest O, Zhang XL. Large language model based multi-agents: A survey of progress and challenges. In: Proc. of the 33rd Int'l Joint Conf. on Artificial Intelligence. 2024. 8048–8057.

沈庆超(1997－), 男, 博士生, CCF 学生会员, 主要研究领域为软件测试, 深度学习编译器测试.

陈翔(1980－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为智能软件工程, 软件仓库挖掘, 经验软件工程.

田家硕(2001－), 女, 硕士生, 主要研究领域为软件测试, 代码审查.

陈庆燕(1979－), 女, 讲师, 主要研究领域为软件测试, 深度学习.

陈俊洁(1992－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为软件分析与测试.

王赞(1979－), 男, 博士, 教授, CCF 专业会员, 主要研究领域为基础软件测试, 智能系统测试.