计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
2026,35(1):129−140 [doi: 10.15888/j.cnki.csa.010037] [CSTR: 32024.14.csa.010037]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

RISC-V 对比 ARM64 的跨架构内核性能评估与缺陷挖掘①

吴雨薇[1,2,3], 徐天宇[4], 杨　旺[1], 于佳耕[1]

[1](中国科学院 软件研究所, 北京 100190)
[2](中国科学院大学南京学院, 南京 211135)
[3](中科南京软件技术研究院, 南京 211100)
[4](北京华为数字技术有限公司, 北京 100085)
通信作者: 于佳耕, E-mail: jiageng08@iscas.ac.cn

摘　要: RISC-V 正处于蓬勃发展的阶段, 其软件生态的移植和构建工作也在持续推进. 由于操作系统本身的复杂性, 其性能评估通常仅停留在模块级别, 而难以在函数实现层面进行系统性的评估. 在软件适配过程中, 往往会借鉴 ARM 等成熟架构的优化策略. 本文提出了一种基于跨架构比较的细粒度内核性能评估与缺陷挖掘方法. 该方法通过跨架构上下文匹配和架构特定的性能异常检测, 使评估重点从模块级别转向函数级别. 本文将该方法应用于 Linux 5.10 内核的 RISC-V 与 ARM 之间的性能对比, 揭示了 RISC-V 在性能上的不足. 通过选定的测试套件进行实验, 在 25 个被识别为异常的上下文中, 检测出了 9 个 RISC-V 的性能问题, 其中 80% 被归类为高优先级问题, 这表明该方法的评估准确率达 80%, 该方法能够在架构适配过程中快速、准确地识别潜在的性能问题.

关键词: RISC-V; 操作系统; 跨架构; 性能评估; 细粒度对比

引用格式: 吴雨薇,徐天宇,杨旺,于佳耕.RISC-V 对比 ARM64 的跨架构内核性能评估与缺陷挖掘.计算机系统应用,2026,35(1):129–140. http://www.c-s-a.org.cn/1003-3254/10037.html

Cross-architecture Kernel Performance Evaluation and Defect Mining: RISC-V vs. ARM64

WU Yu-Wei[1,2,3], XU Tian-Yu[4], YANG Wang[1], YU Jia-Geng[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Nanjing, Nanjing 211135, China)
[3](Nanjing Institute of Software Technology, Nanjing 211100, China)
[4](Beijing Huawei Digital Technologies Co. Ltd., Beijing 100085, China)

Abstract: RISC-V is rapidly evolving, with the porting and development of its software ecosystem advancing. Due to the complexity of operating system itself, performance evaluations are often limited to the module level, making it difficult to conduct systematic analysis at the function level. The software adaptation of RISC-V often draws on optimization strategies from mature architectures like ARM. This study proposes a method for fine-grained kernel performance evaluation and defect detection based on cross-architecture comparison. The method shifts the evaluation focus from the module level to the function level by using cross-architecture context matching and architecture-specific anomaly detection. It was applied to the Linux 5.10 kernel to compare the performance of RISC-V and ARM, which revealed the limitations of RISC-V. Experiments using a targeted test suite identified 9 performance issues from 25 anomalous contexts on RISC-V, with 80% classified as high priority. The result demonstrates that the proposed method achieves an evaluation accuracy of 80%, indicating the effectiveness and accuracy of the method in identifying performance issues during architecture adaptation.

① 基金项目: 国家重点研发计划 (2023YFB4503902)
收稿时间: 2025-05-12; 修改时间: 2025-06-24; 采用时间: 2025-07-29; csa 在线出版时间: 2025-10-11
CNKI 网络首发时间: 2025-10-13

© 中国科学院软件研究所 http://www.c-s-a.org.cn

Key words: RISC-V; operating system; cross-architecture; performance evaluation; fine-grained comparison

指令集是计算机操作系统与芯片之间的接口规范, 对计算机性能、兼容性以及应用范围产生深远影响. RISC-V 以其开放、简洁、稳定、免费以及可模块化的特性而备受青睐, 在未来有望广泛应用于终端设备、个人电脑、服务器和其他计算设备领域[1]. 随着 RISC-V 的快速发展, 面向该架构的操作系统移植适配成为最近研究的热门课题. 目前, 国内外主流操作系统版本已逐步添加对 RISC-V 架构的支持, 包括 Ubuntu、Debian、openEuler、openKylin、Android、OpenHarmony、FreeRTOS、FreeBSD 等, 覆盖了嵌入式系统、终端设备、桌面计算和服务器领域.

尽管近年来围绕 RISC-V 的研究和开发取得了一定进展, 其生态系统整体仍显不够成熟. 随着 RISC-V 架构逐步进入数据中心、边缘智能、工业控制等对性能要求高的场景中, 操作系统的性能表现日益成为影响整个平台应用能力的关键因素[2]. 尽管越来越多操作系统已支持 RISC-V, 但其软件生态仍处于发展阶段, 移植后的系统常常暴露出性能瓶颈, 如响应延迟高、资源利用率低、调度/中断等系统操作效率不佳等问题[3]. 在将操作系统移植到不同架构的过程中, 可能面临多方面的性能挑战, 包括指令集的差异、移植难度、软件优化不足, 以及生态系统支持的局限等. 因此, 有必要深入研究用于评估移植后操作系统性能的方法, 为性能优化提供指导.

操作系统是一款庞大而复杂的基础软件, 其内核包括众多子系统和模块, 不同的应用场景通常需要采用不同的评估方法[4]. 正如 Mogul[5]所指出的, 现实世界中对操作系统的评估具有挑战性, 因为很难对整个系统的性能进行定量评估. 目前的工作[6–8]往往利用具有实用性的基准测试, 以不同的指标评估操作系统不同模块的外在表现. 该测试方法能够对操作系统的外在性能表现进行评估, 对不同类型的操作系统进行现实应用层面的对比. 但已有性能评估方法忽略了操作系统内核函数的性能表现, 缺少对其内部执行情况的系统化分析. 这限制了我们在更细粒度的层面上定位系统性能缺陷以及提供更精准的优化方向的能力.

在操作系统软件的适配与移植研究中, 不同应用场景下即便在同一硬件架构上, 仍会表现出不同的性能特征, 这导致仅在单一架构下进行评估难以揭示移植本身带来的性能影响. 尤其是在 RISC-V 这类新兴架构中, 由于编译链、指令语义支持、运行时环境仍在持续演进, 仅依赖 RISC-V 架构内部的横向比较, 无法准确衡量移植适配带来的性能缺陷. 因此, 有必要以成熟架构 (如 ARM) 作为基准, 通过跨架构对比相同软件特征与操作路径的表现差异, 来评估系统适配效果、定位潜在性能瓶颈. 在进行跨架构的软件性能比较时, 必须综合考虑硬件因素带来的干扰. 微架构由其多个模块构成, 整体性能会受到软件特性的影响, 因此很难仅通过一个单一的数值来简单评估两个不同微架构之间的性能差异[9]. 虽然基于模拟的方法在一定程度上可用于微架构评估, 但在包含像操作系统这样庞大的基础软件的环境中, 模拟仿真速度令人难以接受. 在现实环境中, 研究人员尝试通过样本的预处理进行跨架构的性能预测, 对于具有相似软件特征程序的跨架构性能评估有较高的准确性. 然而操作系统执行特征与计算密集程序的特征区别较大[10], 软件环境更加复杂, 难以通过预处理对性能进行跨架构的预测. 因此, 真实场景中的跨架构的操作系统内核性能对比评估依然是一个具有挑战性的问题.

针对上述问题, 本文提出一种跨架构的操作系统细粒度性能评估方法, 主要贡献概述如下.

(1) 基于 RISC-V 基础软件适配性能问题的研究需求, 提出一种跨架构的基础软件细粒度性能对比方法.

(2) 将该方法应用于 Linux 5.10 内核适配 RISC-V 的性能缺陷评估, 挖掘出 9 处架构间实现的性能差异, 并设计实验进行验证.

(3) 可将该方法更广泛地应用于 RISC-V 基础软件适配性能评估, 将性能异常范围从模块级别缩小到函数级别, 为后续更具体的性能缺陷分析提供方向.

1 相关工作

为了更好地理解当前研究领域的现状, 本节将从 RISC-V 与 ARM 性能对比测试、操作系统评估方法以及跨架构评估方法这 3 方面讨论相关工作.

1.1 RISC-V 与 ARM 性能对比

RISC-V 与 ARM 同为精简指令集架构, 研究人员

© 中国科学院软件研究所 http://www.c-s-a.org.cn

常将二者进行对比, 对 RISC-V 上的一些移植软件进行性能评估测试.

Fibich 等人[11]对各种未经修改的开源 BLAS 库的性能和内存消耗进行了基准测试并进行了讨论, 并且特别关注支持 Linux 的嵌入式平台, 其结果表明在选择的测试中, ARM 平台的性能优于 RISC-V. Kim 等人[12]讨论了 32 位 RISC-V 和 ARM Cortex-M4 环境下的 PIPO 实现方法, 在二者平台上分别进行实现优化, 结果与现有参考实现相比, ARM 的性能提升要高于 RISC-V. Imianosky 等人[13]对在 RISC-V 和 ARM 处理器上运行的 CCSDS 123 算法的性能和功耗进行了评估, 并对使用的两种实时操作系统进行了评估, 结果表明 ARM 处理器比 RISC-V 提供更高的性能和更低的能耗.

Fibich 等人[11]指出 BLAS 库的性能差异可能是由于编译优化差异、具体程序的实现差异以及架构对并行的支持导致的, 而 Kim 等人[12]将 PIPO 实现的性能差异原因归结到 RISC-V 对库的优化比例低于 ARM. 当前的性能对比测试主要集中在工具库和应用软件的性能测试上, 而非基础软件的测试. 此外, 在研究性能差异的原因时, 往往只进行粗粒度的猜测, 而缺乏细粒度的分析.

1.2 操作系统评估方法

关于操作系统的行为评估和性能评价, 可以大致划分为两个主要类别: 第 1 类重点关注性能的评估, 主要侧重于对操作系统的外在表现进行模块化评估; 第 2 类则主要关注可靠性, 着重评估操作系统内核的行为.

第 1 类方法通常涉及对操作系统的不同模块性能进行多角度测试. Marieska 等人[8]使用中断延迟、任务切换时间、抢占时间和死锁打破时间作为性能指标, 对基于内核与嵌入式的实时操作系统进行性能评估. Boras 等人[7]对 3 种不同的 Linux 桌面发行版进行性能评估并探究其对处理器、内存、图形系统和磁盘驱动器性能的影响, 使用专门针对特定计算机组件的 3 种不同基准测试工具进行测量.

第 2 类方法则对操作系统内核的行为进行分析, 主要用于检测恶意入侵和系统崩溃. Liu 等人[14]提出一种新的系统调用跟踪特征提取方法, 目的是提取与系统调用名称无关的特征, 提取的特征所代表的样本可以直接用于跨平台应用的情况, 用于基于主机的跨平台异常检测. Ehsan 等人[15]提出了一种基于主机的异常检测方法, 使用半监督算法结合基于 PCA 的特征提取技术对系统调用跟踪数据进行恶意入侵检测. 本研究结合了这两类思路, 通过在上下文层面对不同架构下的性能表现进行统计, 并从中挖掘性能缺陷, 以便更细粒度地确定可优化的点.

1.3 跨架构评估方法

部分研究尝试通过全方位的分析, 对架构进行整体的评估. Eeckhout[16]在 2010 年提出了架构的整体性能评估框架, 将其分为指标选择、测试负载设计、结果分析模型等步骤, 通过一些架构间共用的指标, 来从不同的角度刻画架构的不同方面的性能, 将其整合尝试得到架构的完整描述. 这种方法仅提供了一个粗略的总体框架, 具体的应用场景需要根据需求进行自定义调整. Bharadwaj 等人[17]利用 gem5 模拟器对 x86 与 ARM 架构进行性能比较, 结合不同内存模型, 从 CPI、指令混合、缓存未命中率、DRAM 带宽和平均功耗等方面评估了架构差异对系统性能的影响. 对架构进行精细的性能分析, 需要对相关测试领域有非常精确的把握, 并且能够将架构各个因素的影响进行归一化剥离. 而随着处理器和应用场景日益复杂化, 架构间各个因素也存在错综复杂的交互影响, 依靠分析评估架构性能也愈发困难.

此外, 部分研究利用样本的预处理, 对具有相似特征的软件进行跨架构的性能预测, 在某些场景下有较高的准确性. Nichols 等人[18]提出通过收集多个高性能计算平台的性能计数器数据, 构建回归模型预测应用在不同架构下的相对性能, 并用于多资源调度优化. Mahdavi[19]提出了一种跨架构性能预测方法, 利用 Paraver 工具收集并行应用程序在 x86 和 POWER9 架构上的硬件性能计数器数据, 构建参考系统与目标系统之间的性能映射关系, 用于在不同平台间预测应用执行时间, 但对于我们关注的操作系统关键路径、上下文切换、内核调度等具有高度系统交互性的场景缺乏针对性支持, 难以适用于操作系统级的适配性与异常评估需求. 基于样本的跨架构性能预测方法往往依赖于样本的选择, 而操作系统内核的测试环境负载和场景复杂度高、变化快, 难以提供足够的具有相似特性的预训练数据, 无法进行后续的跨架构预测评估.

本文采用的跨架构差异自动化评估模型, 不进行精细化考虑架构各因素的具体影响, 节省了相关知识的学习成本. 使用无监督的方式, 充分利用数据而不依

© 中国科学院软件研究所 http://www.c-s-a.org.cn

赖于其样本的预处理, 通过异常检测方法来划分不同架构之间的正常合理差异和性能缺陷点, 由此完成跨架构的性能评估.

2 方法

目前尚缺乏针对 RISC-V 架构在物理开发板环境下上下文功能执行层面的诊断评估研究. 其挑战主要源于微架构设计日益复杂以及操作系统本身规模庞大. 在分析微架构与操作系统结合时的性能相互依赖关系时, 这种复杂性进一步增加了研究难度. 此外, RISC-V 的快速发展对内核移植与适配的优化需求也提出了更高要求.

为了解决这一问题, 本文提出了一种基于动态收集内核函数运行时性能数据的方法, 通过聚类和分类的自动化评估方法, 区分出正常与需要优化的关键异常点. 架构之间的上下文匹配用于过滤出在不同架构下执行相同内核路径的上下文, 并收集其相对性能差异数据. 微架构的自动化性能评估则用于判断微架构性能差异的合理范围, 并通过匹配结果识别性能异常点, 以便为软件适配提供依据. 该方法的整体流程如图 1 所示.

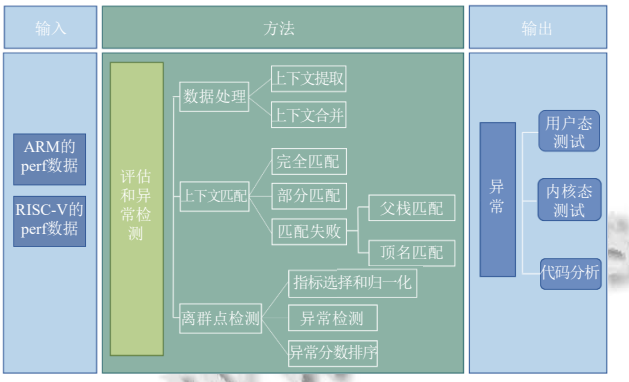


图 1 操作系统性能分析方法

首先, 基于 RISC-V 和 ARM 的真实物理开发板运行环境, 使用 perf 工具对内核关键函数的运行过程进行动态性能数据采集; 其次, 跨架构上下文匹配机制, 用于自动识别在不同平台下执行路径相同或高度相似的上下文实例, 在此基础上, 提取匹配上下文中的性能数据, 构建聚类与分类模型, 自动计算并排序每个上下文实例的异常得分; 最后, 针对异常分数排名靠前的高优先级实例, 结合实际运行表现进行验证, 从而实现对潜在内核性能缺陷的识别与确认.

2.1 跨架构上下文匹配

2.1.1 架构的上下文栈

在操作系统内核中, 部分函数 (如 memset) 可能在不同上下文中被调用, 导致性能受上下文影响而难以直接比较. 因此, 需确保对比的是不同架构下功能一致的上下文. 实验中使用调用栈表示上下文, 通过分析工具采集的调用路径, 识别各上下文中顶层函数所代表的具体内核路径, 并进行性能数据分离分析, 如图 2 所示.

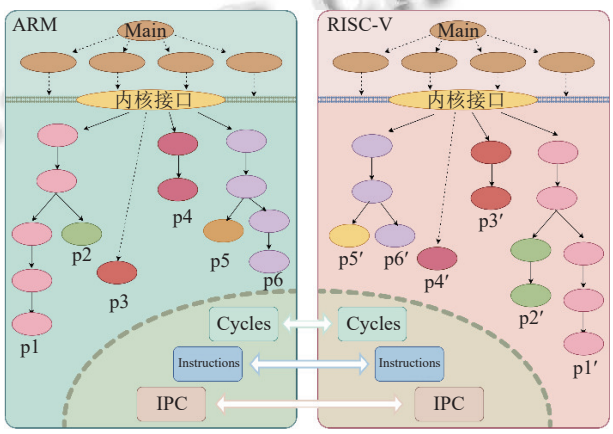


图 2 跨架构内核路径

上下文匹配依赖于调用栈函数的跨架构相似性, 内核的统一性使得相同功能通常在不同架构下产生高度相似的栈结构, 从而简化了匹配过程. 若内核态匹配结果为 0, 表示调用栈结构完全一致, 如图 2 中的完全匹配调用栈对所示. 可视为最理想的匹配情形, 有助于准确对比该功能在不同架构下的性能差异.

2.1.2 上下文匹配算法

调用栈是内核上下文的标识, 其中的函数分为用户态和内核态. 内核态函数描述了 Linux 内核响应用户请求的执行路径, 这也是本文的重点——操作系统内核性能的消耗分析.

跨架构上下文匹配的核心思想是, 在另一种架构中为每个上下文寻找最相似的调用栈, 并据此比较它们的性能. 在初始化匹配之前, 调用栈会以进入内核的接口为界 (例如, RISC-V 的 ret_from_syscall 和 ARM 的 el0 处理接口), 划分为用户态和内核态. 之后, 匹配过程仅在内核路径中进行.

内核态调用栈的匹配算法从调用栈的底部向顶部逐层匹配函数, 直到匹配完成或失败, 并返回体系结构间的调用栈层级差异. 该返回值的含义如下.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

(1) 0: 表示完全匹配.

(2) 正值: 表示 ARM 调用路径更深.

(3) 负值: 表示 RISC-V 调用路径更深.

(4) 负无穷: 表示匹配失败.

如果返回值为 0, 表示两个调用栈中的函数按顺序完全匹配, 如图 2 所示的 p1 和 p1′. 这种理想匹配可以精确计算上下文的性能.

如果返回值不为 0, 表示调用栈之间存在部分匹配. 由于体系结构和工具功能的差异, 某一体系结构可能无法收集与另一体系结构同等深度的函数数据. 在这种情况下, 如果某一体系结构缺少较深层次的函数数据, 应将这些函数的开销合并到其调用方中, 如图 2 所示的 p2 和 p2′、p6 和 p6′. 算法会尽量找到最接近的匹配, 以减少校正损失. 如果两个上下文在不同体系结构中完全匹配, 则可以合并开销; 若完全匹配不存在, 则创建一个匹配上下文以进行校正. 如果一个上下文在另一个体系结构中有多个部分匹配, 所有部分应合并进行全面校正.

若返回值为无穷大, 则表示无法匹配调用栈. 针对体系结构差异和工具适应性, 额外考虑了两种匹配情况.

顶名匹配只依赖调用栈的顶部函数, 减少了工具差异的影响. 若某一函数触发的事件在一个体系结构中低于阈值, 而在另一个体系结构中超过阈值, 则可能无法捕获完整的调用栈信息. 尽管路径相同, 因缺失调用栈数据导致匹配困难, 但这些特征可能仍反映出体系结构差异, 因此需要将这些函数包括在匹配中, 以便后续进行离群点检测, 如图 2 所示的 p3 和 p3′、p4 和 p4′.

父栈匹配比较两个体系结构中相似功能的路径. 某些调用栈从相同父栈接口进入, 但在特定执行路径中存在架构差异. 尽管如此, 仍应基于父函数进行匹配, 如图 2 所示的 p5 和 p5′. 为了保持更高的匹配精度, 应避免过于深入调用栈底部.

最后, 计算所有匹配的相对性能差异, 并汇总成一个数据集合, 为后续分析提供支持.

2.2 微架构性能评估

由于 RISC-V 与 ARM 同属精简指令集架构 (RISC), 指令集和微架构模式具备高度共性, 可将硬件视为黑盒进行对比, 从而聚焦于 RISC-V 的软件适配问题, 为后续优化提供支持.

2.2.1 指标选择和归一化

指标的选择在评估过程中至关重要. 在本实验中, 我们不仅考虑了反映整体性能的能力, 还兼顾了后续的跨架构规范化.

如图 3 所示, 程序性能受源代码、编译器、指令集架构 (ISA)、微架构及系统环境等多因素影响. 软件主要在生成静态机器码前起作用, 硬件则影响动态执行行为. 为识别软件相关性能问题, 需剥离微架构影响, 选用标准化、跨架构通用的指标进行评估.

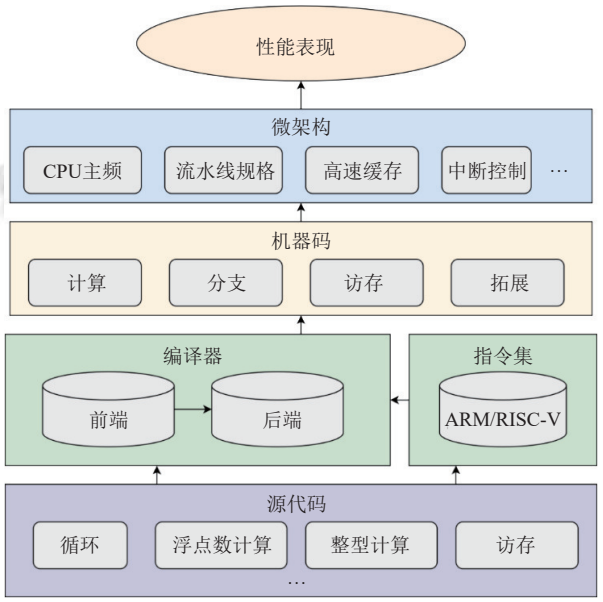


图 3 计算机系统性能影响因素

动态指令数 (instructions): 表示完成某段程序实际执行的指令总数, 受编译器、ISA 和架构共同影响. RISC-V 在支持不同指令类型方面存在不足, 因此选择该指标反映整体代码效率.

周期数 (cycles): CPU 周期是处理器操作的基本单位, 反映整体性能. 与执行时间不同, 周期数排除了时钟频率差异, 提供更稳定的性能数据. 周期计数作为硬件事件可通过性能监控单元 PMU 收集, 并可用于跨架构的宏观和微观性能评估.

每指令周期数 (CPI): CPI 衡量每条指令所需的时钟周期数, 不同架构的流水线差异会影响 CPI. CPI 较高表示吞吐量低. 作为微架构指标, CPI 反映了处理器对指令集架构 (ISA) 的支持, 直接影响系统性能.

此外, 这 3 种指标在其他跨架构评估方法中也被广泛使用[14–17].

尽管不同架构之间的指令功能和延迟可能有所不同, 但本文提出的方法通过自动评估可接受的差异范围, 确保性能差异在正常范围内. 当差异超出正常范围

© 中国科学院软件研究所 http://www.c-s-a.org.cn

时, 说明机器代码存在显著不同, 值得进一步探究.

本研究旨在定位操作系统内核移植到 RISC-V 过程中的性能问题. 通过计算 RISC-V 与 ARM 的性能比值, 得出标准化的跨架构性能差异数据, 为自动化架构评估提供依据. 较大的比值表示 RISC-V 性能较差, 便于异常检测识别.

2.2.2 异常检测

不同上下文中的跨架构差异揭示了架构在多场景中的表现. 偏离正常集群的性能差异被视为异常, 可能存在软件适配问题, 需要进一步识别.

架构差异的正常范围指的是上下文差异在一定范围内的集中程度. 在本研究中, 不同上下文展示了不同的软件特征, 表现为不同的跨架构性能指标比值. 研究者如 Vassiliadis 等人[20]通过 IPC (每周期指令数) 比值评估了不同微架构间的整体性能差异. 他们的研究表明, SPEC 程序中每个程序的 IPC 比值相对稳定.

正常 (normal) 的软件特征会在指定架构上表现出硬件事件比值的紧密集中, 如图 4 中的蓝点所示, 表明这些软件特征在架构间适配良好. 红色的离群点 (outlier) 则提示某些上下文的性能差异超过正常范围, 表明这些功能在架构适配中存在问题, 需进一步改进.

为了自动区分正常差异与离群点, 本研究集成了几种成熟的离群点检测方法, 包括基于聚类的传统算法, 如 LOF[21]和 LUNAR[22], 基于累积分布函数的算法, 如 ECOD[23], 以及机器学习算法, 如 GAAL[24]和 LSCP[25].

为识别最适合此场景的离群点检测方法及其参数设置, 本文分析了多种算法的结果. 随后, 基于选定算法的结果, 评估架构差异并确定性能离群点.

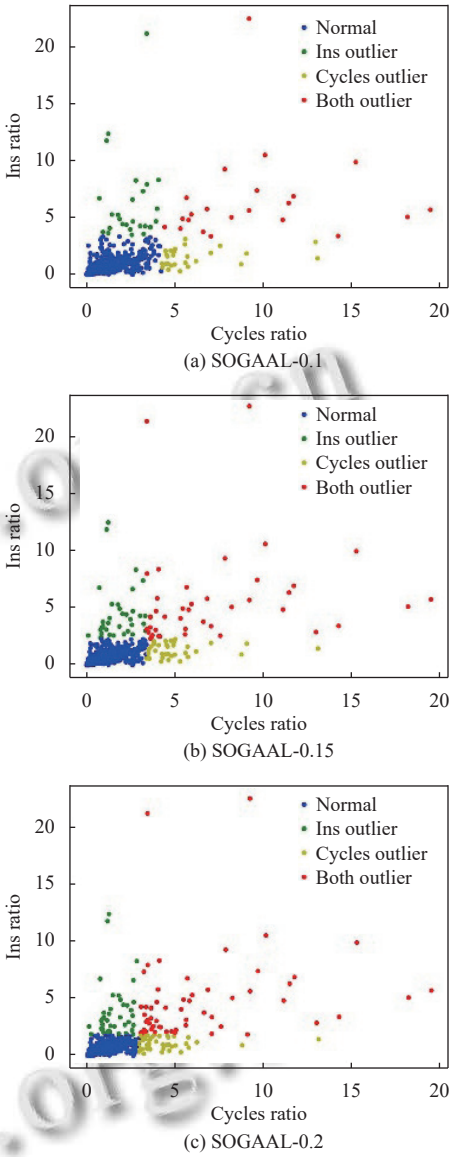
2.2.3 异常分数计算与排序

在离群点检测后, 需要计算上下文的离群点得分, 以便对离群点进行优先级排序. 本实验采用一种直观的方法来评估优先级: 具有较高离群率、更大比例和较大架构差异的离群点优先级更高. 具体方法如下.

(1) 单指标得分计算: 计算离群率、比例和架构差异比值的乘积, 得到单一指标在该上下文中的得分.

(2) 全指标得分计算: 将同一上下文中的 3 个指标得分相加, 得到该上下文的综合得分. 如果某一指标未表现为离群点, 则该指标的离群点得分视为 0.

(3) 排序与分析: 对具有异常的上下文进行排序, 根据得分优先处理离群点. 随后, 对高优先级的性能离群点进行源代码深入分析, 识别异常背后的问题.



(a) SOGAAL-0.1

(b) SOGAAL-0.15

(c) SOGAAL-0.2

图 4 用于阈值检查的异常检测算法示例

3 实验环境

3.1 软硬件环境

对于硬件平台的选择, 应尽量选取结构相近的, 减少架构间的差异. 本实验选取 RISC-V 架构的 Lichee Pi 4A[26]和 ARM64 架构的 Raspberry Pi 4B 开发板, 其板卡硬件指标如表 1 所示. 这两款开发板都采用了多核乱序执行的架构, 同时支持硬件事件计数器, 这些硬件特性为后续的实验提供了重要的基础.

为减少基础软件类型和版本的差异带来的性能差异, 本实验在两块板卡上对齐了基础软件配置, 如表 2 所示. 为避免编译器优化引入偏差, 统一采用-O0 编译选项, 并启用-g 和-fno-omit-frame-pointer 以完整

© 中国科学院软件研究所 http://www.c-s-a.org.cn

获取调用栈信息. 此外, RISC-V 与 ARM64 架构的 Linux 内核尽量开启相同选项, 以保障内核执行路径一致. 主要需要开启的选项有 CONFIG_SMP, CONFIG_PERF_EVENT, CONFIG_SPARSEMEM, CONFIG_SLUB 等.

表 1 硬件平台信息

平台	处理器	架构类型	核心数	主频	流水线规格	分支预测	中断控制器	L1 I-Cache	L1 D-Cache	L2 Cache	内存	性能监测计数器
Raspberry Pi 4B	Cortex-A72	ARMv8-A	4	1.8 GHz	超标量乱序执行	BTB, GHB	GICv2	48 KB	32 KB	1 MB	LPDDR4-4 GB	PMU
Lichee Pi 4A	XuanTieC910	RV64GC	4	1.85 GHz	超标量乱序执行	BTB, BHT	PLIC	64 KB	64 KB	1 MB	LPDDR4 X-8 GB	PMU

表 2 跨平台对齐的基础软件组件版本

基础软件	编译器	内核	操作系统	C运行库
类型版本	gcc 11.3	Linux 5.10	armbian 23.05	glibc 2.35

3.2 工具和测试基准

本实验使用能够动态深入采集性能数据的 Linux 开源性能分析工具 perf[27]. 原因如下.

(1) 硬件模块的需要. 本实验选择的硬件板卡都配备有性能监测计数器 (performance monitor counter, PMC) 模块, 可以完成对硬件事件的记录, 需借助 perf 访问 PMU 获取性能数据.

(2) 数据可靠. 虽然有研究[28]指出 RISC-V 对 perf 工具的支持仍有不足, 但记录如 cycles 和 instructions 等关键事件仍较可靠, 而在 ARM 架构上, 对硬件计数器的使用已经相对成熟[29,30], 事件数据记录也是可靠的.

(3) 跨架构适配性. perf 开源、内置于 Linux 内核, 适合在内核移植过程中同步适配, 优于对架构依赖强的商业工具, 如 VTune.

为了系统评估多样复杂的内核执行路径并为未来的应用场景扩展奠定基础, 本实验通过基准测试触发内核路径, 而非手动选择语句序列进行评估. 测试基准需要具备以下特性: 能够触发内核响应, 并且在大部分时间内运行在内核态. 本次实验选用的测试集包括 rt-tests、hackbench 以及 UnixBench 中的部分测试用例 (例如 context1 和 pipe 等). 这些工作负载在实验板的初步测试中表现出稳定的性能, 因此适用于重复测试, 并将其平均值作为最终结果.

虽然 Lmbench 测试集能够全面地评估操作系统各个方面的性能, 但在实验板卡上的性能表现并不稳定, 因此未被选择. 而 SPEC 测试集主要用于评估计算能力, 通常运行在用户态, 因此并不适用于本文研究的场景.

3.3 上下文匹配结果

本实验中调用栈的收集和匹配详细信息见表 3. 覆盖率的计算方式为: 匹配上下文的百分比与整个内核上下文的百分比之比, 如表 3 所示, 而不是匹配上下文与整个内核上下文的比值. 当两个架构执行相同的工作负载时, 调用栈信息的总数和体积是相同的. 不同架构执行相同内核路径的这一相似性, 强化了上下文匹配的事实依据.

表 3 上下文采集和匹配

架构	采集总数	调用栈	匹配总数	调用栈匹配
RISC-V	1 304	616	734	411
ARM	1 157	616	693	426

需要注意的是, perf 未收集到许多函数的调用栈信息, 正如在第 2.1 节的上下文匹配算法中提到的顶名匹配部分所讨论的那样. 这部分数据不稳定且影响有限. 因此, 直接采用匹配的上下文数量与总内核上下文数量的比值作为评估标准可能并不合理, 因为主要的上下文信息已成功匹配.

本实验中每个测试集的上下文匹配覆盖率见表 4. 尽管架构间的适配和执行存在差异带来了一些挑战, 但总体匹配覆盖率仍然达到了 70%. 有效匹配的上下文数量足以支持后续的异常检测算法. 同时, RISC-V 上的内核消耗了更多的事件, 性能低于 ARM 架构.

表 4 测试集的内核匹配百分比 (%)

测试集	内核函数占比	匹配成功占比	覆盖率
rt-tests	76.82, 69.59	66.15, 51.38	86.11, 73.84
hackbench	90.02, 78.95	81.59, 64.24	90.64, 81.37
UnixBench	83.55, 78.91	55.76, 49.30	66.74, 62.48
平均	—	—	81.16, 72.56

注: 逗号前后分别代表RISC-V和ARM数据

3.4 异常检测算法

本实验采用多种异常检测算法评估微架构间性能差异, 重点考察识别率与召回率. 识别率反映分类效果, 召回率表示识别出的异常占总异常的比例. 为评估识

© 中国科学院软件研究所 http://www.c-s-a.org.cn

别率, 使用轮廓系数衡量算法分类合理性, 数值越高说明类别区分越明显. 各性能指标分别应用不同算法进行异常检测, 并计算对应轮廓系数[31] (见表 5). 实验发现, 异常比例增大时算法识别效果下降, 与图 4 趋势一致. 因此, 异常值比例不宜设置过高.

由于缺乏标签, 召回率评估需依赖人工验证. 为提高召回率, 我们采用较宽松的异常定义, 通过分析检测图的边界条件扩大异常识别范围, 并结合排序与过滤减少误报, 实现更精准的性能差异评估. 异常比例设置需适中, 不同异常检测算法及其参数配置结果如图 5 所示, 根据对比结果, 图中红色点表示被异常检测算法判定为异常的实例. 为进一步聚焦于周期数 (cycles)、动态指令数 (instructions) 与每周期指令数 (CPI) 同时存在偏高情况的异常点, 我们对多种检测算法进行了对比. 在图 5 中可见, SOGAAL-0.15 更有效地识别出了处于比值高区、疑似存在性能瓶颈的关键实例. 最终选用能兼顾高召回率与识别准确性的 SOGAAL-0.15 算法. 异常比例过低会遗漏问题, 过高则导致识别结果失真.

表 5 异常检测方法的轮廓系数

异常检测算法	周期数	动态指令数	IPC
SOGAAL-0.05	0.86	0.874	0.824
SOGAAL-0.1	0.775	0.806	0.784
SOGAAL-0.15	0.704	0.746	0.705
SOGAAL-0.2	0.649	0.683	0.638
LSCP-0.05	0.829	0.843	0.866
LSCP-0.1	0.745	0.762	0.751
LSCP-0.15	0.655	0.725	0.658
LSCP-0.2	0.583	0.642	0.594
ECOD-0.05	0.806	0.792	0.81
ECOD-0.1	0.712	0.708	0.726
ECOD-0.15	0.624	0.634	0.641
ECOD-0.2	0.546	0.567	0.565
LUNAR	0.751	0.735	0.76
LOF	0.775	0.806	0.784

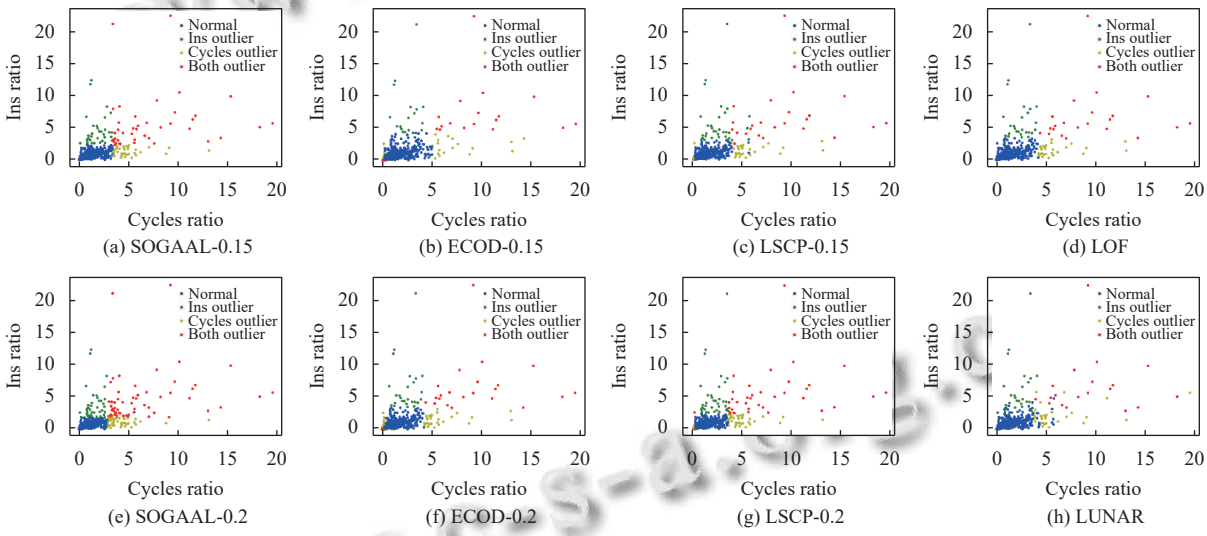


图 5 不同异常检测算法及参数配置的可视化结果

4 结果

根据第 2 节中阐述的方法, 本实验识别出相较于 ARM 在 RISC-V 上表现异常的 25 个上下文, 详见表 6. 随后, 对内核代码的进一步审查以及实验验证揭示, 80% 的异常可以归因于 RISC-V 架构上的性能差异. 此外, 4% 的异常源自 CPU 以外的环境因素, 剩余的 16% 仍待进一步确认原因.

4.1 上下文异常原因发现与分析

本实验对表 6 中的性能异常进行分析, 审查了每个调用栈最上层函数及其相关的宏和内联函数实现, 以挖掘潜在差异. 由于宏和内联函数常被编译器优化并融入调用函数中, 需特别关注其对性能的隐性影响.

此外, 自旋锁的获取与释放会暂时禁用中断, 影响 perf 的数据采集, 使临界区内的性能数据难以分离. 为准确评估其性能影响, 需对这些代码段进行细致分析.

通过上述方法全面分析内核代码, 结合性能异常上下文, 本研究识别出 9 个 RISC-V 架构问题, 分类如下.

(1) 功能缺失: RISC-V 架构中, 部分函数存在功能缺失, 如 memcpy 和 copy_from_user 在处理非对齐地址时效率低, cmpxchg_double 的架构差异也影响性能.

(2) 指令执行差异: RISC-V 缺乏部分对应指令, 需使用替代方案, 如 memset、atomic64_read 和 smp_

© 中国科学院软件研究所 http://www.c-s-a.org.cn

store_release 的汇编实现差异, 造成执行效率下降.

(3) 适配优化不足: 性能差异还源于编译器和基础软件支持不同, 如对__builtin_memcpy 支持不一致, 导致 WRITE_ONCE 等操作效率差异.

表 6 RISC-V 性能异常高优先级上下文

函数	所在测试用例	部分调用栈	异常指标	性能缺陷原因
__get_user_pages	sigwaittest, ptsematest	mprotect_fixup, populate_vma_page_range, __get_user_pages	cycles, ins	—
hrtimer_start_range_ns	cyclictest	do_nanosleep, hrtimer_start_range_ns	cycles, IPC	WRITE_ONCE, cpu_relax
handle_mm_fault	svsematest	populate_vma_page_range, __get_user_pages, handle_mm_fault	cycles, ins, IPC	—
unix_stream_read_generic	hackbench	unix_stream_recvmsg, unix_stream_read_generic	cycles, IPC	memset, memcpy
__slab_free	hackbench	kmem_cache_free, __slab_free	cycles, ins, IPC	cmpxchg_double, cpu_relax
finish_task_switch	cyclictest	do_nanosleep, ⋯, finish_task_switch	cycles	fence and stlr
pick_file	syscall	sys_close, __close_fd, pick_file	cycles, IPC	WRITE_ONCE, fence and stlr
rt_mutex_wait_proxy_lock	pi_stress	futex_lock_pi, rt_mutex_wait_proxy_lock	IPC	—
try_to_wakeup	svsematest	do_semtimedop, wake_up_q, try_to_wake_up	cycles	READ_ONCE, atomic64_read
finish_task_switch	context1	pipe_read, ⋯, finish_task_switch	cycles, ins	fence and stlr
fput	syscall	__se_sys_close, __close_fd, fput	cycles, ins, IPC	READ_ONCE
memset	cyclictest	hrtimer_nanosleep, memset	cycles, IPC	汇编优化差异
get_timespec64	cyclictest	__se_sys_clock_nanosleep, get_timespec64	IPC	copy_from_user
skb_copy_datagram_from_iter	hackbench	unix_stream_sendmsg, skb_copy_datagram_from_iter	cycles, ins, IPC	memcpy
finish_task_switch	signaltest	schedule_hrtimeout_range_clock, ⋯, finish_task_switch	cycles	fence and stlr
lru_add_drain_cpu	ptsematest	lru_add_drain, lru_add_drain_cpu	cycles, ins	WRITE_ONCE
find_next_zero_bit	syscall	__se_sys_dup, find_next_zero_bit	cycles, ins	__ffs
skb_realease_data	hackbench	consume_skb, skb_release_data	cycles, IPC	WRITE_ONCE, atomic64_read
__se_sys_semtimedop	svsematest	ret_from_syscall, __se_sys_semtimedop	cycles, ins, IPC	—
futex_wait_setup	ptsematest	futex_wait, futex_wait_setup	cycles, IPC	atomic64_read
skb_set_owner_w	hackbench	sock_alloc_send_pskb, skb_set_owner_w	IPC	atomic64_read
unix_stream_sendmsg	hackbench	sock_write_iter, unix_stream_sendmsg	IPC	memcpy, copy_from_user
sock_read_iter	hackbench	new_sync_read, sock_read_iter	cycles	—
finish_task_switch	ptsematest	do_nanosleep, ⋯, finish_task_switch	cycles, ins	fence and stlr
unlink_anon_vmas	spawn	free_pgtables, unlink_anon_vmas	cycles, ins	WRITE_ONCE

识别出的差异有助于理解性能异常, 并为后续优化提供指导. 结合 3 项指标, 实验更全面准确地分析了潜在问题, 如表 7 所示.

表 7 已识别问题统计与使用不同优先级度量的异常值覆盖率

指标	top-5%	top-10%	top-15%	top-20%	top-30%
cycles	1 (50%)	2 (50%)	4 (67%)	6 (71%)	8 (81%)
ins	0 (0%)	1 (33%)	2 (60%)	4 (71%)	6 (80%)
IPC	1 (50%)	3 (60%)	5 (83%)	5 (88%)	7 (92%)
all	3 (50%)	5 (67.5%)	7 (75%)	9 (80%)	9 (78%)

4.2 架构间差异验证评估

本文通过 3 种不同的方法——用户态测试、内核态测试和代码实现分析验证和评估了架构间的差异.

4.2.1 用户态测试

一些基础内核函数可以在用户模式下运行, 因此可设计特定的程序来直接评估此模式下的代码执行性能.

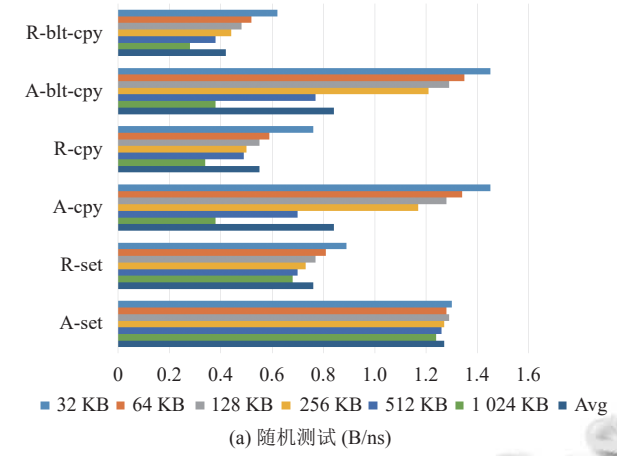
从表 6 可以看出, Linux 内核中的 memset、memcpy 和__builtin_memcpy, 以及在 WRITE_ONCE、READ_ONCE 中使用的这些函数, 对多个异常上下文产生了影响, 这些函数在不同架构之间的实现存在差异.

实验使用了一个从 GitHub 获取的开源工具, 特别评估了这些函数 (memset、memcpy、__builtin_memcpy). 该工具将内核源文件集成, 添加了测试用例, 并编译生成二进制文件以进行性能评估. 结果如图 6 所示, 其中 forward 与 backward 分别表示内存从低地址到高地址、从高地址到低地址的拷贝方向; aligned 表示目标地址按数据类型对齐, unaligned 表示未对齐.

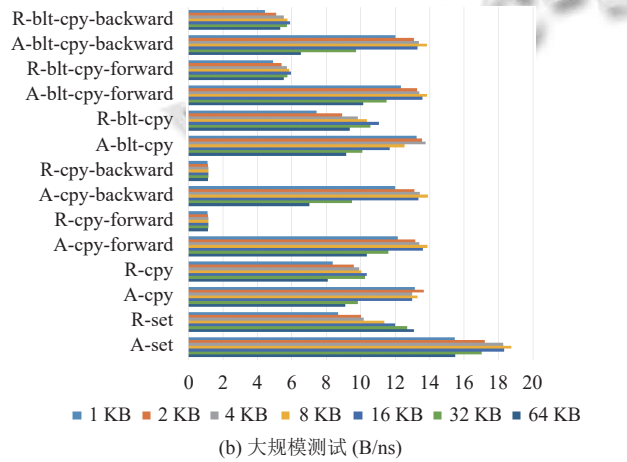
(1) memset 的缺陷. 图 6 显示, 大规模 memset 操作中, 不同架构性能比率在 2 以内, 表明架构间典型差

© 中国科学院软件研究所 http://www.c-s-a.org.cn

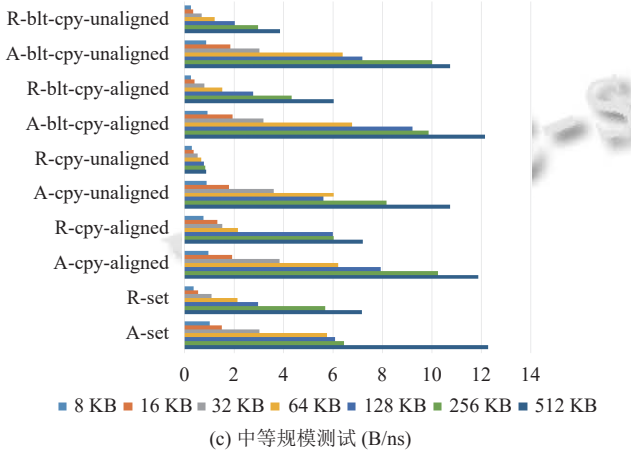
异, 但中等规模下差异更明显, 可能源于实现不同. 一些研究[32]尝试利用 V 扩展来提升其在 RISC-V 上的性能.



(a) 随机测试 (B/ns)



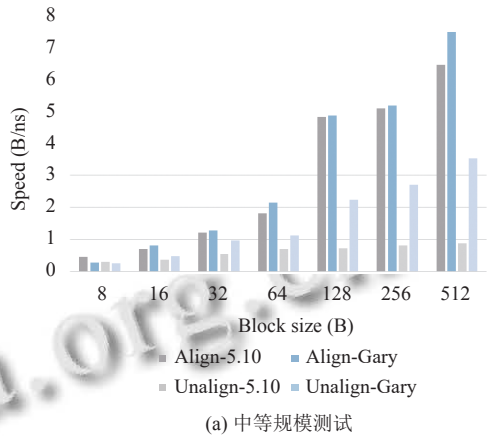
(b) 大规模测试 (B/ns)



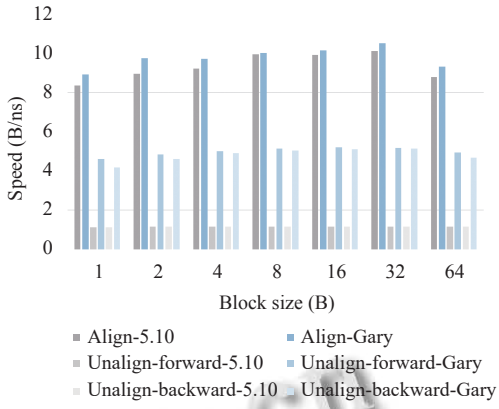
(c) 中等规模测试 (B/ns)

图 6 memset 和 memcpy 速度测试 (A 指 ARM, R 指 RISC-V)

(2) memcpy 的缺陷. 如图 7 所示, RISC-V 在非对齐访问下明显落后于 ARM, 超出正常范围. 原因在于 RISC-V 采用逐字节拷贝, 而 ARM 在处理非对齐后转为按字拷贝. 2021 年 Guo[32]发现并提出了补丁来改进此问题. 该补丁进入投票阶段, 但未被采纳. 补丁虽在非对齐场景更优, 但在对齐场景略有性能损失, 如图 7 所示.



(a) 中等规模测试



(b) 大规模测试

图 7 memcpy 在 RISC-V 上的补丁速度测试

(3) __builtin_memcpy 的差距. 即使在相同版本的 GCC 编译器下, 不同指令集架构的__builtin_memcpy 实现仍有显著差异. RISC-V 上, GCC 内置 memcpy 的性能明显低于 ARM, 影响了 READ_ONCE、WRITE_ONCE 的效率, 并对内核底层数据结构 (如红黑树和链表) 造成性能下降.

性能测试表明, 与 ARM 相比, RISC-V 在内核中广泛使用的函数 (如 memset、memcpy 和__builtin_memcpy) 上表现出较差性能, 揭示了适配需要优化的关键方向.

4.2.2 内核态测试

部分关键函数仅在内核态执行, 受干扰较小. 实验设计了可快速加载卸载的内核模块, 可提升测试效率.

(1) copy_from_user 的缺陷. 测试 1 MB 数据在对齐与非对齐两种情况下的表现如表 8 所示. ARM64 对两者处理差异小, 而 RISC-V 在非对齐场景下性能明显较差.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

表 8 copy_from_user 在内核模块中的开销 (ns)

架构	对齐-1	对齐-2	非对齐-1	非对齐-2
ARM	531 352	534 425	576 769	598 204
RISC-V	265 009	262 669	1 141 021	1 168 687

(2) find_next_zero_bit 的差异. 同样通过内核模块测试 find_next_zero_bit, 在构建的 bitmap 中分别以 0、1 和随机值填充. 表 9 显示, 随机填充最贴近实际环境, RISC-V 在该场景下性能劣势最明显, 暴露出此函数的架构差异.

表 9 find_next_zero_bit 在内核模块中的平均开销 (ns)

架构	1填充	0填充	随机填充
ARM	14 519	10.6	9.77
RISC-V	12 667	30.5	32

对_ffs 函数的测试表明, 同一架构下内核模块中__builtin_ctzl 与 C 实现性能相近, 但不同架构间仍存在差异. Wang[33]的补丁在 Linux 6.7 中引入 Zbb 指令优化位操作, 使编译器在参数为常量时可直接计算结果, 体现了 RISC-V 早期在此类操作上的性能短板.

内核模块虽难完全模拟真实环境, 但可在内核态快速执行测试代码, 有效提升测试效率.

4.2.3 代码分析

部分指令级异常难以通过模块测试验证, 需由 RISC-V 基金会与厂商在指令级优化. 本研究通过分析典型执行差异, 探寻性能差异根因.

(1) cmpxchg_double: 该函数在 Linux 5.10 中由__slab_free 调用, ARM64 架构中提供了特定实现, RISC-V 中则使用统一的实现, 比 ARM64 架构多了 irqsave 和 slablock 的获取与释放. 然而在最新版本的内核中, 已经将 cmpxchg_double 函数移除, 不再使用.

(2) stlr 和 fence: 该函数在 finish_task_switch 和 mark_wake_futex 中被调用, 在 ARM64 用 stlr、RISC-V 用 fence 实现. strl 指令是单向屏障, 仅保证前序访存完成, 不限制后序指令; 而 fence 指令则是双向屏障, 严格限制前后访存顺序. 在实验所使用的支持乱序执行的硬件上, 单向屏障在性能上更具优势.

(3) atomic64_read: 该函数用于原子地读取数据, ARM 通过 ldrexd 指令快速完成, 而 RISC-V 使用 READ_ONCE. RISC-V 缺少特定的指令导致性能差距.

(4) cpu_relax: 该函数用于调度和资源竞争, 在 hrtimer_start_range_ns 和__slab_free 中调用以让出资源. ARM64 使用 yield 指令, 而 RISC-V 则通过无效除 0 指令实现类似效果. 此过程涉及 CPU 调度, 受 ISA、操作系统和架构交互影响, 需在进一步实验分析.

5 总结与展望

本研究提出了一种细粒度内核性能评估与缺陷优化方法, 聚焦上下文匹配与微架构评估, 推动跨架构性能分析与操作系统适配, 为 RISC-V 上的 Linux 优化提供参考. 在对比 RISC-V 与 ARM 架构的过程中, 识别出 25 个高优先级性能异常, 其中 20 个存在实现差异, 检测准确率达 80%. 进一步代码分析发现了 RISC-V 内核中的 9 个问题, 为移植优化提供指导. 该方法不仅适用于内核性能评估, 也可用于其他具有完整运行时调用栈的目标软件, 有助于识别跨架构问题并支持不同内核版本与第三方库的适配优化.

本研究使用的主要为“on-CPU”性能指标, 未覆盖 I/O 与网络等“off-CPU”行为, 后续需扩展指标并充分考虑硬件因素影响, 以提升评估全面性.

实验还发现, 内核路径匹配存在架构间实现差异和输入参数不一致等挑战. 由于缺乏参数信息, 静态分析难以还原完整执行路径. 未来将专注于完善调试信息收集、提升跨架构匹配精度, 并实现更精准的性能分析.

参考文献

1 Greengard S. Will RISC-V revolutionize computing? Communications of the ACM, 2020, 63(5): 30–32. [doi: 10.1145/3386377]

2 Unwana TE, Udoh EI, Umoh VO. A study of the importance of operating system (OS) in a computer system. Journal of Computer Science Review and Engineering, 2022, 1(1): 1–12.

3 Mezger BW, Santos DA, Dilillo L, *et al*. A survey of the RISC-V architecture software support. IEEE Access, 2022, 10: 51394–51411. [doi: 10.1109/ACCESS.2022.3174125]

4 Mogosanu L, Carabas M, Condurache C, *et al*. Evaluating architecture-dependent Linux performance. Proceedings of the 20th International Conference on Control Systems and Computer Science. Bucharest: IEEE, 2015. 499–505.

5 Mogul JC. Brittle metrics in operating systems research. Proceedings of the 7th Workshop on Hot Topics in Operating Systems. Rio Rico: IEEE, 1999. 90–95.

6 Jo YH, Choi BW. Performance evaluation of real-time Linux for an industrial real-time platform. International Journal of Advanced Smart Convergence, 2022, 11(1): 28–35.

7 Boras M, Balen J, Vdovjak K. Performance evaluation of Linux operating systems. Proceedings of the 2020 International Conference on Smart Systems and Technologies (SST). Osijek: IEEE, 2020. 115–120.

8 Marieska MD, Hariyanto PG, Fauzan MF, *et al*. On

© 中国科学院软件研究所 http://www.c-s-a.org.cn

performance of kernel based and embedded real-time operating system: Benchmarking and analysis. Proceedings of the 2011 International Conference on Advanced Computer Science and Information Systems. Jakarta: IEEE, 2011. 401–406.
9 Smith JE, Sohi GS. The microarchitecture of superscalar processors. Proceedings of the IEEE, 1995, 83(12): 1609–1624. [doi: 10.1109/5.476078]
10 Redstone JA, Eggers SJ, Levy HM. An analysis of operating system behavior on a simultaneous multithreaded architecture. ACM SIGPLAN Notices, 2000, 35(11): 245–256. [doi: 10.1145/356989.357012]
11 Fibich C, Tauner S, Rössler P, *et al*. Evaluation of open-source linear algebra libraries targeting ARM and RISC-V architectures. Proceedings of the 15th Conference on Computer Science and Information Systems (FedCSIS). Sofia: IEEE, 2020. 663–672.
12 Kim Y, Seo SC. Optimized implementation of PIPO block cipher on 32-bit ARM and RISC-V processors. IEEE Access, 2022, 10: 97298–97309. [doi: 10.1109/ACCESS.2022.3205617]
13 Imianosky C, Valim PRO, Zeferino CA, *et al*. Evaluating the CCSDS 123 compressor running on RISC-V and ARM architectures. Proceedings of the 2020 X Brazilian Symposium on Computing Systems Engineering (SBESC). Florianopolis: IEEE, 2020. 1–7.
14 Liu Z, Japkowicz N, Wang RY, *et al*. A statistical pattern based feature extraction method on system call traces for anomaly detection. Information and Software Technology, 2020, 126: 106348. [doi: 10.1016/j.infsof.2020.106348]
15 Aghaei E, Serpen G. Host-based anomaly detection using Eigentraces feature extraction and one-class classification on system call trace data. arXiv:1911.11284, 2019.
16 Eeckhout L. Computer Architecture Performance Evaluation Methods. Cham: Springer, 2010.
17 Bharadwaj SV, Vudadha CK. Evaluation of x86 and ARM architectures using compute-intensive workloads. Proceedings of the 2022 IEEE International Symposium on Smart Electronic Systems (iSES). Warangal: IEEE, 2022. 586–589.
18 Nichols D, Movsesyan A, Yeom JS, *et al*. Predicting cross-architecture performance of parallel programs. Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS). San Francisco: IEEE, 2024. 570–581.
19 Mahdavi K. A hybrid machine learning method for cross-platform performance prediction of parallel applications. Proceedings of the 53rd International Conference on Parallel Processing. Gotland: ACM, 2024. 669–678.
20 Vassiliadis S, Sousa L, Gaydadjiev GN. The midlifekicker microarchitecture evaluation metric. Proceedings of the 2005 IEEE International Conference on Application-specific Systems, Architecture Processors. Samos: IEEE, 2005. 92–97.
21 Breunig MM, Kriegel HP, Ng RT, *et al*. LOF: Identifying density-based local outliers. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. Dallas: ACM, 2000. 93–104.
22 Goodge A, Hooi B, Ng SK, *et al*. Lunar: Unifying local outlier detection methods via graph neural networks. Proceedings of the 36th AAAI Conference on Artificial Intelligence. AAAI Press, 2022. 6737–6745.
23 Li Z, Zhao Y, Hu XY, *et al*. ECOD: Unsupervised outlier detection using empirical cumulative distribution functions. IEEE Transactions on Knowledge and Data Engineering, 2023, 35(12): 12181–12193. [doi: 10.1109/TKDE.2022.3159580]
24 Liu YZ, Li Z, Zhou C, *et al*. Generative adversarial active learning for unsupervised outlier detection. IEEE Transactions on Knowledge and Data Engineering, 2020, 32(8): 1517–1528.
25 Zhao Y, Nasrullah Z, Hryniewicki MK, *et al*. LSCP: Locally selective combination in parallel outlier ensembles. Proceedings of the 2019 SIAM International Conference on Data Mining. Philadelphia: Society for Industrial and Applied Mathematics, 2019. 585–593.
26 Chen C, Xiang XY, Liu C, *et al*. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension: Industrial product. Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA). Valencia: IEEE, 2020. 52–64.
27 Gregg B. Linux perf examples. https://www.brendangregg.com/perf.html. (2023-08-07)[2023-10-07].
28 Domingos JM, Tomas P, Sousa L. Supporting RISC-V performance counters through performance analysis tools for Linux (Perf). arXiv:2112.11767, 2021.
29 Dongarra J, London K, Moore S, *et al*. Using PAPI for hardware performance monitoring on Linux systems. Proceedings of the 2001 Conference on Linux Clusters: The HPC Revolution. Linux Clusters Institute, 2001, 6.
30 Oprofile, a system profiler for Linux. https://oprofile.sourceforge.io. (2023-05-11)[2024-01-05].
31 Rousseeuw PJ. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Journal of Computational and Applied Mathematics, 1987, 20: 53–65. [doi: 10.1016/0377-0427(87)90125-7]
32 Guo G. RISCV: Fix memmove and optimise memcpy when misalign. https://patchwork.kernel.org/project/Linux-riscv/patch/20210216225555.4976-1-gary@garyguo.net/#24196567. (2021-02-16)[2023-10-07].
33 Wang. RISCV: Optimize bitops with Zbb extension. https://github.com/torvalds/Linux/commit/457926b253200bd9bdfae9a016a3b1d1dc661d55. (2023-11-10)[2024-01-05].

(校对责编: 李慧鑫)

© 中国科学院软件研究所 http://www.c-s-a.org.cn