计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(5): 1191−1208, 2026

基于内核函数调用频率的云应用性能监测指标

黄紫微　姚益祁　包云岗　王　卅
（处理器芯片全国重点实验室（中国科学院计算技术研究所）　北京　100190）
（中国科学院大学计算机科学与技术学院　北京　101408）
（huangziweivv@gmail.com）

A Performance Detection Metric for Cloud Applications Based on Kernel Function Call Frequencies

Huang Ziwei, Yao Yiqi, Bao Yungang, and Wang Sa
（*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190）
（*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 101408）

Abstract　Improving resource utilization is important for cloud service providers to achieve cost reduction and efficiency, as the cost of construction and operating costs in datacenters are extremely high. However, existing metrics such as benchmarks are often scenario-specific and lack adaptability. Some metrics, such as simulated or pre-execution solutions are struggle to cope with sudden workload fluctuations. Therefore, cloud service providers urgently require a real-time performance metric for monitoring performance and managing resources. Existing real-time metrics often rely on software or hardware support, and most fail to accurately identify resource bottlenecks. To address this challenge, we propose KFCMetric (metric based on kernel function call frequencies), a performance metric based on the frequency of kernel function calls. KFCMetric employs static kernel analysis techniques to generate kernel function call graphs and utilizes a random forest algorithm to select kernel functions that exhibit a strong correlation with application performance and resource status. The behavior of these kernel can dynamically reflect the runtime resource state of applications and accurately pinpoint the root causes of performance bottlenecks. Building on KFCMetric, we further introduce a resource management system, KFCSys. Using dynamic kernel instrumentation techniques, KFCSys captures the calls and returns of these kernel functions and uses a variance-based scaling algorithm to measure the deviation of data points from normal. Based on the real-time status of latency-critical applications, KFCSys dynamically adapts resource allocation to meet the resource requirements of latency-critical applications, ensuring their service quality. Experimental results indicate that KFCMetric achieves an accuracy of 96.7% in detecting resource bottlenecks, and its interference detection accuracy shows up to a 31.4% improvement over CPI. In terms of resource management, KFCSys reduces the average tail latency by up to 30.4% compared with PARTIES, and by 40.7% compared with the standard Linux resource management system. Moreover, KFCSys decreases the proportion of performance violation time by 10% relative to PARTIES. Therefore, in scenarios where hardware or software support is not available, KFCMetric, constructed from kernel function features, is a novel and robust performance metric for performance monitoring and resource management.

收稿日期：2024-12-29；**修回日期**：2026-01-12
基金项目：中国科学院战略性先导科技专项（XDA0320000, XDA0320300）；国家自然科学基金项目（62090022, 62090023, 62172388, U24B6012）
This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences（XDA0320000, XDA0320300）and the National Natural Science Foundation of China（62090022, 62090023, 62172388, U24B6012）.
通信作者：王卅（wangsa@ict.ac.cn）

Key words cloud computing; performance metric; resource management; datacenters; kernel instrumentation

摘 要 云计算数据中心的建设和运维成本居高不下，因此提升资源利用率、实现“降本增效”一直是云服务提供商尤为关注的话题。然而，现有的性能指标如基准测试，通常基于特定场景，缺乏适应性。而模拟运行或预运行的方案难以应对负载的突然变化。因此，云服务提供商迫切需要一种实时的性能指标来进行性能监测和资源管理。但是，现有的实时指标通常依赖于软硬件支持，且大多数无法准确识别资源瓶颈。为此，提出了 KFCMetric（metric based on kernel function call frequencies），一种基于内核函数调用频率的性能指标。KFCMetric 通过静态内核分析技术生成内核函数调用图，并利用随机森林筛选出与应用性能和资源状态高度相关的内核函数。这些内核函数的行为能够实时反映应用程序的运行状态，并精确定位性能瓶颈的根源。基于 KFCMetric，进一步构建了资源管理系统 KFCSys。KFCSys 借助内核动态插桩技术实时捕获这些内核函数的调用与返回信息，并通过方差倍数法来量化数据点偏离正常值的程度来预警性能下降，从而根据延迟敏感型应用的实时状态，调度器动态调整资源分配，以满足延迟敏感型应用的资源需求。实验结果表明，KFCMetric 在资源瓶颈检测方面的正确率达到 96.7%，且在干扰检测上相比 CPI 的正确率最高提升 31.4 个百分点。在资源管理方面，KFCSys 相较于 PARTIES 最高降低了 30.4% 的平均延迟，同时比 Linux 资源管理系统降低了 40.7%。此外，在性能违例时间占比上，KFCSys 较 PARTIES 减少了 10%。因此，在硬件或软件支持不可用的情况下，KFCMetric 基于内核函数调用频率特征，为性能监测与资源管理提供了一种新颖且稳健的解决方案。

关键词 云计算；性能指标；资源管理；数据中心；内核插桩

中图法分类号 TP316

DOI: 10.7544/issn1000-1239.202440970 **CSTR:** 32373.14.issn1000-1239.202440970

云计算数据中心的建设运维成本高企，因此一直以来提升资源利用率[1]、“降本增效”都是云服务提供商尤为关注的话题。通常情况下，单一应用程序难以充分利用服务器的所有资源，特别是如购物网站、在线聊天软件等在线延迟敏感类应用。例如，根据 Cast AI[2] 的报告，AWS 和 Azure 的平均资源利用率仅为 11%。因此，云服务商通常会采用多应用混合部署的方式，将在线延迟敏感型应用与 CPU 利用率高的如 MapReduce 等离线批处理应用混合部署，拉升集群的整体利用率。然而混合部署却给在线应用带来了严重的性能干扰问题，在离线应用共享底层服务器的 CPU、内存和 I/O 等关键资源，不可避免地在共享资源上产生严重的资源竞争，进而导致在线应用的延迟增加。

为了解决这一问题，云服务提供商首先需要一套性能监测机制，能够实时监控云应用的运行状态。当检测到性能下降时，能够迅速诊断当前的资源瓶颈并进行资源重新调度，以保障云应用的性能。然而，要想实现准确的云应用运行时性能监控并非易事。通常情况下，特别是公有云环境，出于安全隐私等方面因素考虑，云服务提供商不能对上层云应用的应用层性能（响应延迟等）进行直接监测，云应用对于云服务提供商就是一个黑盒，云服务提供商只能通过采集底层应用运行时数据，“猜测”或“估算”上层应用的运行时性能，这就为云服务商提出了极大的挑战。只有准确地判断出上层应用何时发生性能干扰以及为什么发生性能干扰，才能精准地进行资源分配和调度决策，保障应用服务质量。

现有的性能监控方法主要可分为 3 类：1）基准测试[3-4] 通过执行一系列标准化任务或操作，评估系统或应用程序在特定工作负载下的表现，通常关注响应时间、吞吐量等关键性能指标。基准测试能够提供系统在特定条件下的性能表现，但其适用范围通常局限于特定工作负载或场景，难以全面反映复杂环境中混合部署应用的资源竞争和动态变化。因此，基准测试无法准确捕捉实际运行中的多种资源争用情况。2）模型类指标[5-6] 通过构建应用的性能模型，分析应用在运行时可能产生的压力以及在不同压力条件下的性能表现。该方法基于一系列假设和简化，推测系统在更大规模环境中的表现。然而，随着机器配置复杂化、应用数量和干扰类型的多样化，模型的精度可能受到影响，导致其在大规模数据中心环境中的应用变得困难。3）实时指标[7-8] 通过实时采集软硬件性能指标来评估系统性能，相较于基准测试和模型类指标，实时指标能更好地应对动态变化的运行环境。然而，这种方法的采集依赖于特定软硬

件机制的支持，例如 CPI[2,7] 依赖于硬件性能计数器，而端到端延迟[9] 则依赖于应用层主动提供接口。因此，针对复杂的运行环境，我们试图探索一种新的评估指标，该指标不依赖于软硬件支持，能够适应更广泛的应用场景。

本文提出了 KFCMetric（metric based on kernel function call frequencies），一种基于内核函数调用频率的性能监测指标。根据经验，内核中存在大量的快慢速路径，这些路径与资源状态密切相关。于是，我们通过内核跟踪工具扫描内核函数的调用与返回，发现某些内核函数的调用频率与应用延迟高度相关，并且通过分析内核调用频率的变化，能够间接反映硬件资源的波动。基于 KFCMetric，我们构建了资源管理系统 KFCSys。该系统通过监测内核函数调用频率的动态变化来指示性能下降，并识别资源瓶颈。在检测到干扰时，系统会自动调用相应的资源隔离技术，以保障应用的服务质量。

然而，实现该目标面临 3 个关键挑战：

1）庞大的内核搜索空间。Linux 内核包含数万个函数，且它们之间存在复杂的调用关系。因此，从这些错综复杂的调用关系中精准识别与性能相关的内核行为是一个具有挑战性的任务。

2）应用资源敏感的差异性。不同应用程序由于功能需求和实现逻辑的差异，其内核函数调用的频次和模式存在显著差异。因此，需要设计通用的性能指标，以适应多种类型的应用，并通过统一的阈值定义降低开发者和监控人员的干预成本，从而提升方案的通用性和易用性。然而，设计通用指标以满足不同应用需求的多样性仍然是一个复杂的任务。

3）构建内核行为与资源联系。某些硬件资源的处理不经过内核，例如现代处理器中的 LLC（last-level cache）缓存缺失处理完全由硬件自行完成，这使得通过内核行为反映硬件状态变得异常困难，从而增加了构建内核行为与底层硬件状态关联模型的复杂性。

为解决挑战 1，即内核的搜索空间庞大，KFCMetric 采用静态内核跟踪技术捕获内核函数调用图，并结合随机森林算法筛选出最具特征的内核分支，从而有效缩小分析范围并减少开销。针对挑战 2，即应用资源敏感的差异性，KFCMetric 通过方差倍数化方法量化数据点偏离正常值的程度，提供统一的性能评估方式，简化了配置过程并降低了复杂性。为应对挑战 3，即内核无法直接感知硬件压力类型，KFCMetric 通过实时监控内核分支调用频率的变化，间接反映硬件资源的波动，及时识别由硬件资源竞争引发的性能问题。

通过筛选一批与应用性能和资源瓶颈具有较高相关性的内核函数，KFCMetric 能够实时感知应用程序的运行状态，并精确定位性能瓶颈的根源。然后根据受干扰应用的实时状态，动态反馈式调整不同应用的资源分配，保障延迟敏感型应用的延迟和服务质量。

实验结果表明，在干扰检测方面，对于简单负载压力类型的检测，KFCMetric 的准确率达到了 96.7%。与 CPI 相比，KFCMetric 检测干扰的准确率提高了 31.40%。基于 KFCMetric 指标构建的资源调度框架 KFCSys，凭借 KFCMetric 在性能指示和资源瓶颈定位方面的优势，其平均尾延迟相比于 PARTIES 提升幅度达到 30.4%。此外，KFCSys 相较于 PARTIES[9] 违例率降低了 10%。

本文的主要贡献包括 3 个方面：

1）低依赖性能监控方法。本文提出了一种无需硬件支持或软件层指标的性能监控方案，通过内核关键函数调用频次占比来指示性能下降。在一个内核版本上进行一次静态分析后，后续无需离线分析即可实现高效监控。

2）高精度资源瓶颈类型推断。通过提取不同干扰状态下的内核函数调用图，识别与资源状态相关的关键分支，并利用随机森林筛选，显著减少内核搜索空间中的函数范围，快速定位性能瓶颈。

3）统一性能指标。为了避免为每个应用程序单独配置性能监控参数的复杂工作，我们采用偏差倍数法来统一衡量各个指标偏离正常范围的程度，实现了不同类型应用的统一性能评估框架。

1 相关工作

当多个应用共享同一台服务器的内存、CPU 和网络等资源时，它们之间不可避免地会发生资源竞争。这种竞争可能导致应用程序的响应延迟增加，特别是对于延迟敏感型应用而言，这可能影响其满足服务水平协议（service level agreement, SLA）。为了有效管理数据中心资源并优化性能，需要建立一种机制，能够实时监控应用性能，并在性能下降时触发资源重新调度。性能指标在这一过程中起着至关重要的作用。本节将探讨不同的性能评估方法，并根据性能指标的不同构成进行分类讨论。我们将现有的性能评估方法分为三大类：基准测试、模型类指标和实时类指标。

1.1 基准测试

基准测试[10][11]是评估系统或组件性能的一种标准方法。通过执行一系列标准化任务或操作来评估应用在特定场景下的性能，其评估维度包含响应时间、吞吐量[12]、资源使用量(CPU[13-14]、内存[15]、I/O[16-17]和电力[18-19])和并发能力等。基准测试能够分析应用在不同负载条件下的性能表现，帮助识别潜在瓶颈和优化机会，从而辅助性能调优和资源管理[20]。例如，BigDataBench[21]提供了一个全面的大数据和AI工作负载基准测试套件，专注于评估相关应用的性能，涵盖多种数据类型和处理技术。Perfkit Benchmarker[22]由Google公司开发，其测试结果使得用户可以在不同的云服务和配置之间进行有效的性能比较。DeathStarBench[23]则专注于微服务和云原生应用，通过提供多种用例的基准测试来评估社交网络、媒体服务和在线零售等应用的性能。

然而，基准测试的设计通常基于特定的场景和工作负载，未充分考虑混合部署场景下资源竞争对应用的影响，因此难以准确反映在真实环境场景下的应用的性能。此外，基准测试依赖于静态的预定义测试集，无法捕捉系统机器配置、混部应用类型和负载的动态变化，例如，资源需求的激增。因此，基准测试适用范围通常局限于特定工作负载或场景，难以全面反映复杂环境中混合部署应用的资源竞争和动态变化。

1.2 模型类指标

模型类指标通过构建模型来描述应用程序在运行时产生的干扰以及在承受干扰时的性能表现。考虑的资源维度包括CPU计算资源、内存带宽、缓存和网络带宽等。例如，Bubble-Up[24]和iBench[25]通过构建压力与服务质量(quality of service, QoS)之间的对应关系，利用压力性能曲线来描述各应用程序可容忍的压力水平及其对系统造成的压力，从而预测在混合部署场景下各个应用的性能表现。Paragon[26]评估应用在运行时对系统会造成多大的干扰，以及自身在多种共享资源中可以容忍多大的干扰。此外，Paragon还考虑了服务器配置对应用性能的影响。在应用正式运行之前，在2个服务集群上进行短时间的运行，以对新部属的应用进行异构性和干扰分类。

为了避免因环境差异而导致的模型不准确，部分研究采用沙箱、克隆或请求复制的方式进行建模分析。例如，Quasar[27]在Paragon的基础上进行了资源的垂直扩展和水平扩展，测量在同一节点上分配更多资源(如CPU核心和内存)对性能的影响，以及增减节点数量对性能的影响。Quasar使用沙箱收集运行时数据，并通过协同过滤算法中的奇异值分解(singular value decomposition, SVD)和乘积量化(product quantization, PQ)重构，填充性能矩阵中的缺失值，从而预测每个节点的性能。DeepDive[28]利用代理在沙箱中复制客户端请求来模拟应用执行，进而推断应用的正常和异常状态。WSMeter[29]构建了一个小型集群以评估集群中应用的性能。CLITE[30]利用贝叶斯优化和资源等价类特性，探索多个资源维度的配置，例如计算核心数量、共享缓存空间、内存带宽、内存容量和磁盘带宽，从而提供优化的资源分配建议。

模拟类指标[24-25]通过性能压力模型或提前运行的方式来模拟程序的真实运行，考虑了混合部署场景下应用之间因资源竞争引起的性能下降。一些研究[27]也考虑了机器的资源状态以及异构配置对性能的影响。对于云应用而言，其鲁棒性相较于基准测试更强。然而，模拟类指标通常需要耗时的离线训练来构建压力性能模型，且随着应用数量和场景复杂度的增加，这一离线分析的开销会呈指数级增长。另一方面，模拟类指标主要是考虑到混部应用的类型、机器的配置等因素，但在动态变化的环境中，面对突发负载等无法预见的情况时，模型指标的准确性往往难以保证。

1.3 实时类指标

实时类指标通过收集运行时数据来评估应用程序的性能，主要分为应用层性能指标和操作系统/硬件层性能指标2类。应用层性能指标，如尾延迟[31-32]、每秒操作次数和吞吐量，在能够从云环境中获取时通常被视为理想的性能指标[33-34]。例如，Heracles[33]和PARTIES[9]利用这些指标对应用进行资源配置并进行性能优化，IX[35]则以吞吐量和延迟作为性能指标进行调度优化。然而，由于安全性和隐私等原因，云服务提供商无法直接监控上层云应用的应用层性能。对于云应用，云服务提供商只能将其视为一个“黑盒”，无法访问具体的应用层来运行数据。因此，云服务提供商只能通过收集底层应用的运行时数据推测或估算上层应用的性能表现。

因此，部分工作[36]利用操作系统和硬件层面的实时类指标来推测上层应用的性能表现。例如，每周期指令数(instructions per cycle, IPC)、每秒指令数(instructions per second, IPS)或每秒百万指令数(millions of instructions per second, MIPS)等，这些指标是衡量处理器效率、吞吐量和响应能力的重要指标。例如，CPI^{2}收集性能监控单元(performance moni-

toring unit, PMU)的每指令周期(cycles per instruction, CPI), 通过比较集群中节点之间的 CPI 数据来判定性能异常节点, 而无需事先构建性能模型。

还有一些研究[37-38]通过监测特定的硬件或软件特性来定位所竞争的资源。例如, Alita[8]通过监测锁定频率、LLC 容量分配和每个核心的温度来反映性能。DeepDive[28]在沙箱中进行离线分析, 并通过监控一些底层的指标(如 cpu_unhalted, inst_retired, retired_load, L2_ifetch 等)来评估运行时性能。FlARE[39]将多个系统级或作业级指标(如 CPU 利用率、重试指令数和 LLC 未命中率)整合为高层级指标, 但该方法需要收集数十种甚至上百种不同类型的指标, 这不仅对软硬件支持提出了较高要求, 还需在运行时将这些指标聚合为单一信号或少量关键信号, 以确保调度器能够基于一致的指标信息做出准确决策, 避免因信号不一致而导致调度失败。

实时类指标在应对动态负载和复杂线上运行环境时展现了较强的适用性。应用层指标能够最直观地反映应用性能。然而, 在云环境中, 服务提供商通常难以直接获取应用层数据, 往往依赖底层指标来推测或估算上层应用的性能。但底层指标的使用需要底层硬件(如 PMU)的支持, 且由于底层硬件可能对某些类型的干扰不敏感, 这会导致较高的误报率。此外, 利用底层指标定位资源瓶颈通常需要采集、分析和整合多维度的资源数据, 显著增加了分析的复杂性。

尽管现在已有大量的性能评估方案, 但目前仍然缺乏一种无需依赖大量离线分析、既高效实用又能精准反映资源瓶颈的性能指标。基准测试通常基于特定的场景和工作负载, 无法应对复杂的云应用混部场景。模拟类指标根据离线分析或者模拟执行来评估应用在运行时的性能。但是在实际生产环境中, 应用程序通常只部署 1 次, 无法通过请求复制或预执行等方式进行性能评估, 从而使模拟指标难以适用于云环境中。为此, KFCMetric 提出了一种基于内核层实时指标的性能监控方法, 有效避免了对 PMU[40-41]等硬件性能监控工具的依赖。通过内核跟踪工具, KFCMetric 可以实时捕获应用程序的内核行为监测性能, 并精确定位性能瓶颈。

2 内核级性能指标

数据中心一般将应用分为 2 类: 一类是尽可能优先保障的延迟敏感(latency-critical, LC)应用; 另一类是尽力而为(best-effort, BE)的次优先级应用。KFCSys 基于 KFCMetric, 通过将内核函数调用频次作为性能指标来监测 LC 应用的性能, 并根据监测结果重新分配资源。如图 1 所示, KFCSys 架构包含 4 个模块: 静态分析、内核插桩、性能监测和资源调度。静态分析模块利用内核跟踪工具构建内核函数调用图, 并通过比较不同干扰场景下各应用的调用图来提取关键分支。内核插桩模块利用动态插桩技术对这些分支相关的内核函数进行插桩, 在应用运行时监测父函数与子函数的行为, 从而获取各关键分支中子函数与父函数的调用占比。这些分支占比数据通过 BPF 缓冲区传递给用户层, 用于性能监测。在运行时, 性能监测模块根据预先分析的内核分支特征以及 KFCSys 通过方差倍数法统计当前分支占比偏离正常值的程度, 来检测当前是否发生性能下降, 如果发生性能下降, 则进一步诊断当前的资源瓶颈。最后, 资源调度模块根据性能监测模块的性能指标调用相应的子控制器对资源进行重新分配, 从而保障在 LC 应用满足 SLO 的前提下提升资源利用率。

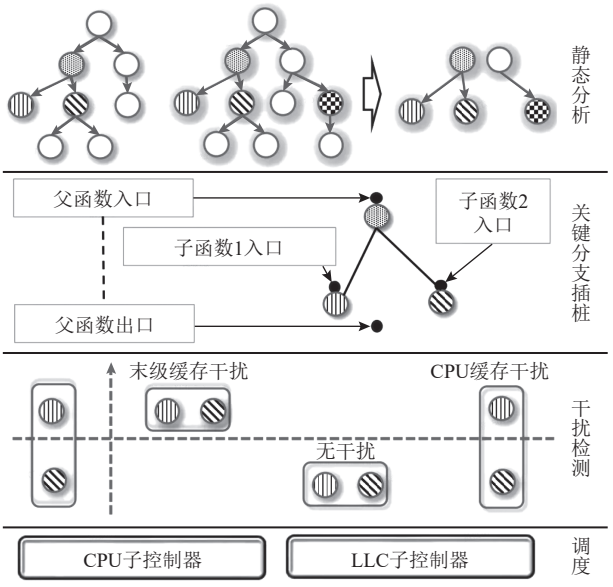


Fig. 1 Architecture of KFCSys

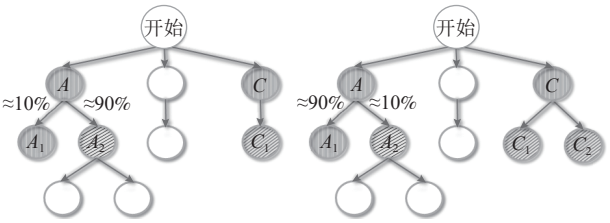
图 1 KFCSys 架构

KFCMetric 的核心思想是通过分析内核函数调用分支占比, 间接反映硬件资源的波动。具体而言, 它构建了内核关键分支调用行为与底层资源之间的映射, 在运行时通过监测这些分支的变化, 实时反映应用性能。一旦监测到性能下降, KFCSys 便通过所构建的映射精准定位资源瓶颈, 从而指导资源调度。

2.1 静态分析

为了从大量内核函数中提取出与性能密切相关的关键函数, 我们的基本思路是首先利用静态内核

跟踪工具(如 Ftrace[42])对应用程序进行内核函数调用跟踪,记录内核函数的调用与返回事件,并基于此构建内核函数调用图,如图 2 所示为一个简化示意图。



(a) 无干扰 (b) CPU干扰

注: 百分数为父函数调用各子函数的比例。

Fig. 2 Illustration of critical call branches in kernel

图 2 内核关键调用分支示意图

在该调用图中,我们重点关注函数间的调用关系及其调用频次。通过对比不同资源干扰场景下的函数调用图,我们能够找出最具特征的内核函数调用分支,并将其作为指导调度决策的依据。例如,图 2 中阴影节点表示在不同场景下调用频次差异显著的函数,我们将其定义为 KFCMetric 关键分支。假设函数 A_1 代表函数 A 的慢路径调用,函数 A_2 则代表快路径调用。在 CPU 资源紧缺时,慢路径调用频率相比无干扰状态时增加了约 80%。具体而言,无干扰状态下,函数 A_1 调用次数仅占函数 A 总调用次数的 10%,而在干扰条件下,该比例可能迅速上升至 90%。

2.1.1 关键分支筛选

在静态分析阶段,我们首先利用内核跟踪工具构建内核函数调用图。随后,通过对比不同干扰场景下函数调用图的差异,识别出对底层资源状态敏感的父子函数调用。为了精准筛选内核关键分支,我们进一步使用随机森林算法,以函数调用占比(子函数调用次数相对于其父函数调用次数的比例)作为输入特征,评估各调用路径对系统性能的影响。具体筛选过程关注 2 个指标:一是稳定性,即在资源状态稳定时,函数调用占比的波动不超过 10%。二是差异性,即当底层资源状态发生变化时,函数调用占比的波动超过 10%,体现出明显的资源敏感性。

那么,我们最终确定了 KFCMetric 需要插桩的 3 个关键内核分支,如表 1 所示。其中,前 2 个分支父函数为 *select_idle_sibling*,分别调用了子函数 *select_idle_cpu* 和 *cpus_share_cache*。第 3 个内核关键分支父函数为 *try_to_wakeup*,其调用子函数 *ttwu_do_activate*。

KFCMetric 通过静态分析获得的 3 个关键分支都是内核调度代码的关键路径。下面我们将从内核代码内部深入探究这 3 个关键分支在发生 CPU 干扰

Table 1 Critical Kernel Function Calls for KFCMetric

表 1 KFCMetric 的关键内核函数调用

内核函数调用	功能描述
select_idle_sibling ⇒*select_idle_cpu*	慢速路径调 *select_idle_cpu* 扫描整个 LLC 域空闲核
select_idle_sibling ⇒*cpus_share_cache*	该调用用于判断 2 个 CPU 核是否共享 LLC,辅助快速选核
try_to_wake_up ⇒*ttwu_do_activate*	加锁维护目标 CPU 核的运行队列,修改目标核的运行任务

或 LLC 干扰时的调用频次是否会有明显变化,从而验证静态分析得到的关键路径的合理性。

1)函数 *try_to_wake_up* 关键分支

本节分析函数 *try_to_wake_up* 及其子函数如何反馈当前系统的压力状态。函数 *try_to_wake_up* 的功能是根据任务的可唤醒状态和唤醒标志来唤醒指定线程,并为其选择合适的运行队列。该函数通过调用子函数 *ttwu_do_activate* 负责完成唤醒任务的具体过程,包括将任务插入到指定队列,以及抢占正在执行的任务。在函数 *try_to_wake_up* 的逻辑中,只有函数 *ttwu_queue* 会调用函数 *ttwu_do_activate*。如图 3 所示,函数 *ttwu_do_activate* 的调用取决于第 4 行的条件判断是否为真,即函数 *ttwu_queue_wakelist* 返回的布尔值。

```
1  static void ttwu_queue(struct task_struct *p, int cpu,
       int wake_flags) {
2      struct rq *rq=cpu_rq(cpu);
3      struct rq_flags rf;
4      if (ttwu_queue_wakelist(p, cpu, wake_flags))
5          return;
6      rq_lock(rq, &rf);
7      update_rq_clock(rq);
8      ttwu_do_activate(rq, p, wake_flags, &rf);
9      rq_unlock(rq, &rf);
10 }
```

Fig. 3 Code snippet of the kernel function *ttwu_queue*

图 3 内核函数 *ttwu_queue* 代码片段

当目标 CPU 为远端且处于空闲时,函数 *ttwu_queue* 会先尝试通过函数 *ttwu_queue_wakelist* 直接对远端 CPU 发送 IPI 以请求其自行唤醒操作本地运行队列;但若远端 CPU 此时处于非空闲状态,则 IPI 无法带来性能提升,此时函数 *ttwu_queue* 会在本地 CPU 调用函数 *ttwu_do_activate*,对目标运行队列加锁操作。当 CPU 干扰严重时,目标核空闲的可能性显著降低,导致函数 *ttwu_queue_wakelist* 返回“true”的概率降低,从而使函数 *ttwu_do_activate* 的调用占比上升。因此,一旦监测该内核分支的调用占比,即可判定应用可能正受到干扰。

2)函数 *select_idle_sibling* 关键分支

本节分析函数 *select_idle_sibling* 及其子函数在不

同类型干扰下的调用行为，图 4 直观地展示了 2 个关键分支在不同干扰场景下的调用占比。我们将 LC 应用 xapian 分别与 CPU 干扰（深灰色）和 LLC 干扰（浅灰色）混合部署，并以纵坐标显示这 2 个函数的调用次数占函数 *select_idle_sibling* 调用次数的百分比。从图 4 中可以看出，在 CPU 干扰阶段，函数 *cpus_share_cache* 的调用占比降低，而函数 *select_idle_cpu* 的调用占比显著上升；在 LLC 干扰阶段，函数 *cpus_share_cache* 和函数 *select_idle_cpu* 的调用占比均显著上升，并且在 2 次相同实验中均呈现一致特征，说明这一现象并非偶然。接下来，我们将深入分析其原因。

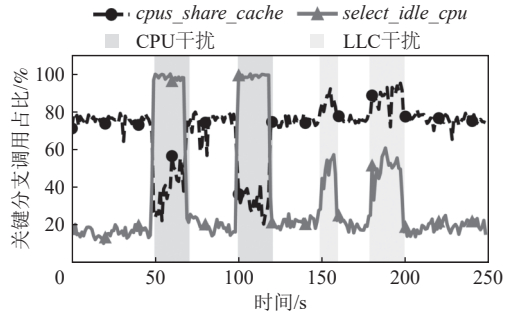


Fig. 4 Variation trends of call proportion of critical branches

图 4 关键分支的调用占比变化趋势

函数 *select_idle_sibling* 代码如图 5 所示。该函数通过快速选核路径检查目标核（*target*）和任务的旧核（*prev*）是否为空闲。如果 *target* 或任务的 *prev* 共享 LLC 且为空闲，快速路径将直接返回（第 2~6 行）到父函数，无需要调用函数 *select_idle_cpu*（第 18 行）来扫描整个 LLC 域。LLC 域指的是共享最后一级缓存的一组 CPU 核。

如果这些快速路径未能找到空闲核，并且该函数中其他的快速选核策略也全部失败（第 2~16 行），则调用函数 *select_idle_cpu* 遍历整个 LLC 域中的所有核以寻找空闲 CPU。因此，在 CPU 干扰与 LC 应用混合部署的情况下，快速路径更容易失效，导致函数 *select_idle_cpu* 调用占比增加。函数 *cpus_share_cache* 用于判断 2 个核是否共享 LLC 缓存，并在代码中多次辅助快速选核逻辑（第 5, 11, 16 行）。

然而，函数 *cpus_share_cache* 分支的调用频次取决于原来所使用的 *prev* 和 *target* 之间的负载平衡。具体而言，函数 *select_idle_sibling* 的传入参数 *target* 由函数 *wake_affine* 决定，该函数在 *target* 和 *prev* 之间选择空闲核。如果二者都非空闲，函数 *wake_affine* 会根据负载值选择负载较小的一方。在 CPU 干扰严重时，由于各核的负载趋于均衡，函数 *wake_affine* 更倾向于选

```
1  static int select_idle_sibling(p, prev, target) {
2      if ((available_idle_cpu(target)||sched_idle_cpu(target))
           && asym_fits_cpu(…))
3          return target;
4      // if the previous CPU is cache affine with target.
5          if (prev!=target&&cpus_share_cache(prev, target))…
6          return prev;
7      …
8      // Check a recently used CPU as a idle candidate:
9      if (recent_used_cpu != prev &&
10         recent_used_cpu != target &&
11             cpus_share_cache(recent_used_cpu, target) &&…
12     …
13     // Enabling hyperthreading, if no cores are idle, check
               prev is cache affine with target.
14     if (sched_smt_active()) {
15         has_idle_core=test_idle_cores(target, false);
16         if (!has_idle_core &&cpus_share_cache(prev, target)
            ) {…} }
17     …
18     i=select_idle_cpu(p, sd, has_idle_core, target);
19     …
20     return target;
21 }
```

Fig. 5 Code snippet of the kernel function *select_idle_sibling*

图 5 内核函数 *select_idle_sibling* 代码片段

择 *prev*，这会导致 *prev* 与 *target* 相等的概率增加（第 5 行），从而减少函数 *cpus_share_cache* 的调用频次。

在 LLC 干扰的场景下，由于共享 LLC 的核更容易出现 *target* 和 *prev* 同时非空闲的情况，快速选核路径较难满足条件，导致函数 *cpus_share_cache* 的调用概率上升。同时，快速选核失败的概率增大，促使调度逻辑遍历整个 LLC 域寻找空闲核，从而增加了函数 *select_idle_cpu* 的调用频率。因此，数据分析表明，当应用受到不同类型干扰时，函数会根据干扰的类型呈现出不同的调用特征。进而，可以通过静态分析构建内核行为与底层资源状态之间的映射关系。

2.1.2 插桩方案

KFCMetric 通过 eBPF（extended Berkeley packet filter）动态内核插桩技术，对内核函数调用进行监控，从而实现对关键性能指标的采集与分析。eBPF 允许在 Linux 内核中安全地执行自定义代码。KFCMetric 通过自动化流程，依据预设模板生成 eBPF 程序，如图 6 所示，这些程序能够精确捕捉特定父子函数调用对的调用频率。

KFCMetric 的内核插桩需要一个如表 1 所示的配置文件作为输入，该文件指定了需要追踪的父函数（*caller*）和子函数（*callee*）。根据这些配置信息，KFCMetric 为特定的父函数和子函数生成相应的 eBPF 程序，并通过插桩的方式将代码嵌入到内核中。

父函数的插桩位置设置在其入口和出口处，主要用于记录父函数的调用频次和调用栈帧。具体而言，程序通过监测进程号来跟踪目标进程。如果进

```
1  BPF_HASH(IN_CALLER, u32, u64);
2  BPF_HASH(CALLER_COUNT, u32, u64);
3  BPF_HASH(CALLEE_COUNT, u32, u64);
4
5  int trace_CALLER_entry(struct pt_regs *ctx) {
6      u64 tgid_pid=bpf_get_current_pid_tgid();
7      u32 tid=tgid_pid >> 32;
8      if (tid==TARGET_PIDS) {
9          IN_CALLER.update(&tid, &tgid_pid);
10         u64 zero=0;
11         u64 *c=CALLER_COUNT.lookup_or_init(&tid, &zero);
12         (*c)++;
13     }
14     return 0;
15 }
16
17 int trace_CALLER_exit(struct pt_regs *ctx) {
18     u64 tgid_pid=bpf_get_current_pid_tgid();
19     u32 tid=tgid_pid >> 32;
20     if (tid==TARGET_PIDS) {
21         IN_CALLER.delete(&tid);
22     }
23     return 0;
24 }
25
26 int count_CALLEE(struct pt_regs *ctx) {
27     u64 tgid_pid=bpf_get_current_pid_tgid();
28     u32 tid=tgid_pid >> 32;
29     if (tid==TARGET_PIDS && IN_CALLER.lookup(&tid)) {
30         u64 zero=0;
31         u64 *count=CALLEE.lookup_or_init(&tid, &zero);
32         (*count)++;
33     }
34     return 0;
35 }
```

Fig. 6 Code snippet of eBPF instrumentation template

图 6 eBPF 插桩模版代码片段

程号符合目标条件，则递增父函数的调用计数器。当父函数执行结束时，程序会从哈希表中移除该进程号，以确保后续对子函数调用是由该父函数发起的。

子函数的插桩旨在统计子函数的调用频次，其插桩位置设置在子函数的入口处。插桩代码首先确认当前线程是否为目标线程，并检查哈希表以判断子函数的调用是否由父函数触发。如果条件满足，则表明父函数正在执行，此时递增子函数调用频次的计数器。通过这种方式，仅统计由指定父函数触发的子函数调用，从而避免其他路径对子函数调用导致的统计误差。

通过 eBPF 的动态插桩，KFCMetric 获取了关键分支中每一对子函数调用频次与父函数调用频次的占比。这部分数据在运行时以字符串形式通过管道传递至性能监测模块，用于诊断和识别性能瓶颈。

2.2 性能监测

KFCMetric 的性能监测通过识别内核异常行为来实现，其核心是监控内核函数关键分支调用占比是否显著偏离正常值范围。如图 7 所示，在 3 维空间中展示了关键分支调用占比在无干扰、CPU 干扰和 LLC 干扰场景下的分布情况。在这些场景下，可以直接观察到指标数据在同一类型干扰下的聚集特性。因此，基于指标与资源状态之间的映射，我们可以推断当前应用是否受到干扰，以及当前干扰类型是 CPU 干扰还是 LLC 干扰。

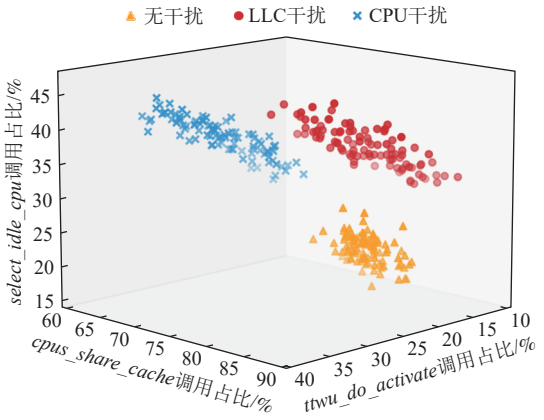


Fig. 7 Distribution of the call proportion of critical functions

图 7 关键函数的调用占比分布

在图 8 中，我们将进一步评估这一特性在不同应用场景下的可扩展性。图 8 显示了 6 个应用依次与 CPU 干扰(深灰色阴影)和 LLC 干扰(浅灰色阴影)混合部署后，各关键分支的百分比变化。

从图 8 中可以看出，当应用与 CPU 干扰混部时，内核关键分支函数 *ttwu_do_activate* 和函数 *select_idle_cpu* 的调用占比上升，而函数 *cpus_share_cache* 占比则下降。相反，当应用与 LLC 干扰混合部署时，3 个指标均上升，特别是函数 *cpus_share_cache* 的上升尤为显著。这些趋势与我们之前的分析一致。

因此，KFCMetric 基于内核关键函数特征设计性能监控与资源动态调整的算法。算法 1 分为预处理阶段和监测阶段。在预处理阶段，我们收集应用在不受干扰的场为预处理阶段和监景(但应用独占运行，无混部其他应用)下的内核各关键分支调用占比数据，以此作为基准性能特征。监测阶段会基于预处理阶段的数据计算运行时数据的偏差程度，并根据计算结果来判定当前的性能水平。一旦监测到性能下降，就利用指标来判定当前的性能瓶颈。

算法 1. 性能监控与资源动态调整。

① 收集 10 s 数据并计算各指标的均值和方差；

② 归一化指标得到 $ratio_1$, $ratio_2$, $ratio_3$；

③ while 监控开启

④ 从 *eBPF map* 中读取指标；

⑤ if $ratio_1 > M_1$

⑥ if $ratio_2 > M_2$ & $ratio_3 < -M_3$

⑦ 正在遭受 CPU 干扰，调整 CPU 资源；

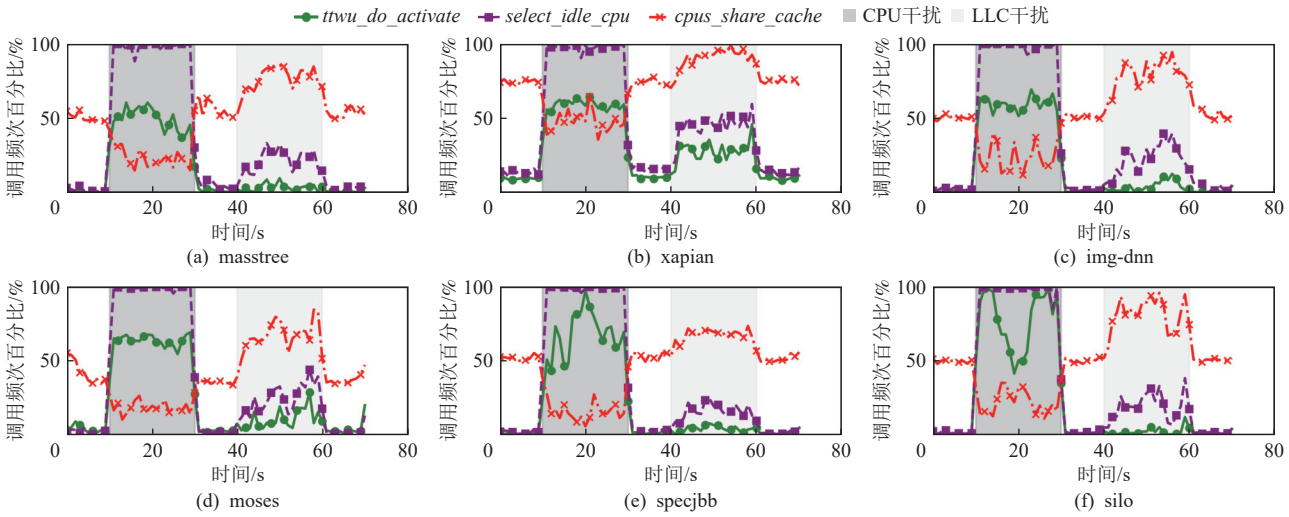


Fig. 8 Proportion of key branch calls of each kernel in the mixed deployment of six delay-critical applications

图 8 6 个延迟敏感型应用混部时各内核关键分支调用占比

⑧ 设置间隔 $stall = 5$;
⑨ else if $ratio_2 > M_2$ & $ratio_3 > M_3$
⑩ 正在遭受 LLC 干扰，调整 LLC 资源;
⑪ 设置间隔，$stall = 5$;
⑫ else if $ratio_1 < M_1$
⑬ if $stall > 0$
⑭ $stall--$;
⑮ else if $stall == 0$
⑯ 恢复资源至默认状态;
⑰ end if
⑱ end if
⑲ end if
⑳ end while

eBPF 监测的内核数据存储到 BPF 缓冲区中，并可以被应用层读取。监测模块在应用层以固定间隔时间循环读取缓冲区内容，并将每个分支中子函数调用频次与父函数调用频次的占比按时间维度存入数组，每个分支对应一个独立的数组。循环读取的间隔根据系统灵敏度要求设定，本文中设置为 1 s。在预处理阶段，系统将每个指标的关键数据存储到对应的数组中，记为 $Metric_{\text{profile}}$，预处理时间设置为 10 s。预处理结束时计算每个数组的均值和方差，分别记为 $\mu\left(Metric_{\text{profile}}\right)$ 和 $\sigma(Metric_{\text{profile}})$。

$$ratio = \frac{Metric_{\text{runtime}} - \mu(Metric_{\text{profile}})}{\sigma(Metric_{\text{profile}})},$$

其中，$Metric_{\text{runtime}}$ 表示在运行时采集到的 3 个分支的占比，$ratio_1$ 和 $ratio_2$ 分别对应函数 *select_idle_sibling* 调用函数 *select_idle_cpu* 和函数 *cpus_share_cache* 的频率的偏差倍数，而 $ratio_3$ 则对应函数 *try_to_wakeup* 调用函数 *ttwu_do_active* 的频率相对于均值偏差倍数。随后，运行时数据将根据均值和方差来衡量计算各关键指标偏差倍数，阈值参数 M_1, M_2, M_3 可根据实际需求灵活配置。例如，M_1 用于判断系统是否出现性能下降，数值越大，系统对干扰的容忍度越高。M_2 和 M_3 则对应 CPU 干扰 与 LLC 干扰的识别阈值，其设置通常根据无干扰状态下指标的偏差波动范围设定。以 M_2 为例，若无干扰情况下相关指标的偏差一般在 3 倍标准差以内，则将 M_2 设置为 5 可有效区分是否存在干扰。具体取值依据实际机器配置进行调整，在实验环境中，我们分别将 M_1, M_2, M_3 设置为 10, 3, 1。

这些指标直接用于指示应用的运行时性能状态以及当前系统的所竞争资源类型。当 $ratio_1$ 超过阈值 M_1 时，系统判断关键应用受到干扰，并进行资源重分配。此时，如果资源未初始化，将调用函数 *init_resource* 进行资源初始化；接着，根据静态分析所提取的关键分支调用特征，通过 $ratio_2$ 和 $ratio_3$ 判断干扰类型：当 $ratio_2 > M_2$ 且 $ratio_3 < -M_3$ 时，表明 LC 应用正在遭受 CPU 干扰；当 $ratio_3 > M_3$ 时，表示存在 LLC 干扰。

为了避免频繁调整引发的性能波动和资源重分配带来的性能开销，KFCMetric 设置了调整间隔，并通过变量 *stall* 进行控制。当 $ratio_1$ 恢复至低于 M_1 时，KFCMetric 判定 LC 应用的运行状态正常，并开始递减计数器 *stall* 的值，直到计数器归零，此时资源配置将完全恢复至默认状态。恢复阶段开始后，BE 应用可以共享 LC 应用的资源，以提高整体资源利用效率。

2.3 资源调度

KFCMetric 的资源调度模块的调度策略是优先满足 LC 应用的资源需求。在 LC 性能稳定时，KFCMetric

允许 BE 应用使用剩余资源,以提高整体资源利用率;当检测到资源竞争时,调度模块采用资源隔离技术以避免资源争用。具体而言,CPU 资源是通过 Cgroup 的 Cpuset 机制重新绑定 CPU 核心,LLC 资源则利用 Intel 的资源管理技术(resource director technology, RDT)进行动态调整。

如图 9 所示,我们将应用分为 LC 组和 BE 组。在没有发生干扰的情况下,BE 和 LC 应用共享 CPU 和 LLC 资源。当检测到 LC 应用因 BE 应用引发的 CPU 资源争用而受到干扰时,系统通过 Linux Cgroup 的 Cpuset 隔离核心,使 LC 应用能够使用更多的核心,以确保其资源需求。Cpuset 通过配置文件定义任务组的 CPU 核心集合,并依托内核调度器限制任务的运行范围。通过动态更新配置文件,系统能够实时调整 CPU 分配,核心的添加或移除将即时生效。

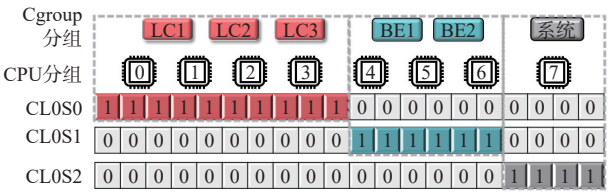


Fig. 9 Resource isolation mechanism of CPU and LLC

图 9 CPU 和 LLC 资源隔离机制

当 LLC 缓存发生资源竞争时,系统通过 Intel 的缓存分配技术(cache allocation technology, CAT)[43-44]为 LC 应用隔离更多的缓存资源。CAT 将缓存资源划分为不同的服务类别(class of service, COS),并通过 RDT 技术将 LLC 缓存分为多个块,每个块由 1 个比特位表示,进程仅能访问位掩码中对应的缓存块。同时,CAT 通过 CLOSID 限制进程对缓存的访问。这种方法有效地隔离了 LC 和 BE 应用的缓存资源,避免了相互干扰。当系统检测到 LC 应用的性能稳定且持续一段时间后,KFCMetric 将会允许资源共享,即 BE 组的应用可以使用 LC 组应用剩余的资源,从而确保系统资源得到高效利用,提升系统整体吞吐能力。

3 实验评估与分析

在实验中,我们从 4 个方面评估 KFCMetric。

1)正确率。与现有性能指标相比,KFCMetric 的正确率可以从 2 个方面进行评估:干扰检测正确率(实验 1)和瓶颈识别正确率(实验 2)。

2)可扩展性。在不同云环境配置、内核版本和应用场景下,KFCMetric 的可扩展性(实验 3)。

3)干扰场景性能。在干扰环境下,评估 KFCSys 相比先进资源管理框架所带来的性能提升(实验 4)。

4)运行时开销。使用 eBPF 插桩时,测量 KFCMetric 引入的运行时性能开销(实验 5)。

3.1 实验配置

1)机器配置。实验在一台配置了 2 颗 Intel Xeon E5-2630 v4 处理器的服务器上进行,总计提供 20 个物理核心,支持高效并行计算。服务器配备 64 GB 主存和 50 MB 共享 LLC 缓存。KFCMetric 利用处理器支持的 Intel 资源管理技术(resource director technology, RDT)[44]缓存分配技术 CAT 对 LLC 缓存进行动态分配,以满足不同函数的资源需求。实验运行环境为 Ubuntu 20.04 操作系统,Linux 内核版本是 5.15.0 以及 Docker[45]版本为 20.10.14。为验证可扩展性,实验同时在阿里云 ECS 实例上部署,配置为 8 vCPU(Xeon Platinum 8582C),16 GB RAM,运行 Ubuntu 24.04(Linux 6.12.0 内核)。

2)实验对照。在对比实验中,我们在干扰检测方面选择了 CPI 作为比较指标;在资源瓶颈识别方面,由于本文主要关注 CPU 与 LLC 干扰,因此选取了 CPU 资源利用率以及 LC 应用与 BE 应用的 LLC Cache Miss 作为对比工作;在资源管理方面,我们比较了 DefaultSys 与 PARTIES 这 2 个资源管理框架。

① CPI。CPI 是衡量 CPU 性能的重要指标,用于 Google 实际生产环境中[7, 36]。

② PARTIES。PARTIES 框架的优势在于直接使用应用延迟作为性能指标,这使得它无需进行额外的离线性能分析。此外,由于 PARTIES 直接基于应用层端到端的延迟,因此在检测性能下降时,其准确性要显著高于其他基于间接推测方法。

③ KFCSys。KFCSys 基于 KFCMetric 进行性能推断,由于缺少应用层信息,其主要依赖内核指标(关键内核函数调用频率)进行性能推断,因此需要通过在预处理阶段(10 s 基准采集窗口)建立关键分支调用的基准占比。这种额外的基准采集步骤使得 KFCSys 与 PARTIES 的比较是相对公平的。

④ DefaultSys。此外,DefaultSys 本身是基于 Linux 操作系统标准的资源管理框架,并未针对应用感知进行特别优化。在 CPU 调度层面,依赖 CFS(completely fair scheduler)的完全公平时间片分配;在 LLC 管理层面,采用完全共享的 LLC 策略。

⑤系统层指标。本实验采用 CPU 资源利用率和 LLC Cache Miss 作为系统层指标。CPU 资源利用率是经典的干扰检测指标,通过设定一个安全阈值(例

如 70%)来判断 LC 应用性能是否下降，数据采集由 Perf 工具完成。LLC Cache Miss 作为常用性能指标，已被 Caladan 和 FLARE 等系统采用，本实验通过 PQoS 工具采集相关数据。

我们将应用分为 LC 应用和 BE 应用。BE 应用是利用 Stress-ng[46] 工具来模拟实际运行环境中的干扰。其中，CPU 干扰通过同时启动多个 CPU 密集型进程来大量消耗 CPU 资源实现；而 LLC 干扰则由多个缓存抖动触发。LC 应用选用了 Tailbench[47] 基准测试工具，其中包括 img-dnn, xapian, silo, specjbb, masstree, moses 这 6 个应用，并额外增加了 redis 和 nginx 这 2 个常见应用，具体如表 2 所示。

Table 2 Description of Latency-Critical Application

表 2 延迟敏感型应用描述

应用	应用描述	规模	工作负载
img-dnn	图像识别系统	毫秒级	Tailbench
xapian	全文检索引擎	毫秒级	Tailbench
moses	机器翻译系统	毫秒级	Tailbench
specjbb	Java 基准测试	毫秒级	Tailbench
masstree	KV 键值存储库	微秒级	Tailbench
silo	内存数据库	微秒级	Tailbench
redis	内存数据库	毫秒级	redis-benchmark
nginx	Web 服务器	毫秒级	Apache Benchmark

3.2 正确率评估实验

作为性能指标，KFCMetric 在指示应用性能水平和定位资源瓶颈方面的准确性至关重要。为此，我们在图 10 中展示了 LC 应用分别混部 CPU 干扰(深灰色阴影)和 LLC 干扰(浅灰色阴影)后的各项指标变化趋势。图 10 中①展示 6 个应用的每秒第 95 百分位延迟，明显看出 LC 应用性能均出现了不同程度的下降。由于各应用对资源干扰敏感性不同，其性能下降幅度也存在差异。图 10 中②展示关键分支调用占比的变化趋势，我们基于函数 *ttwu_do_activate* 调用占比计算性能指标，并定义性能指标偏离正常值超过 10 个方差即视为性能下降。图 10 中③展示了 KFCMetric 对干扰检测的推断结果，当检测到性能下降时，KFCMetric 随即产生预警信息。图 10 中④则进一步展示了 KFCMetric 对当前性能干扰类型的具体推断结果。

实验 1. 干扰检测正确率评估。为比较 KFCMetric 与传统性能指标的干扰检测能力，我们在图 11 展示了性能指标 CPI、CPU 利用率和缓存不命中率的变化趋势，并通过干扰检测的正确率评估这些指标的有效性。

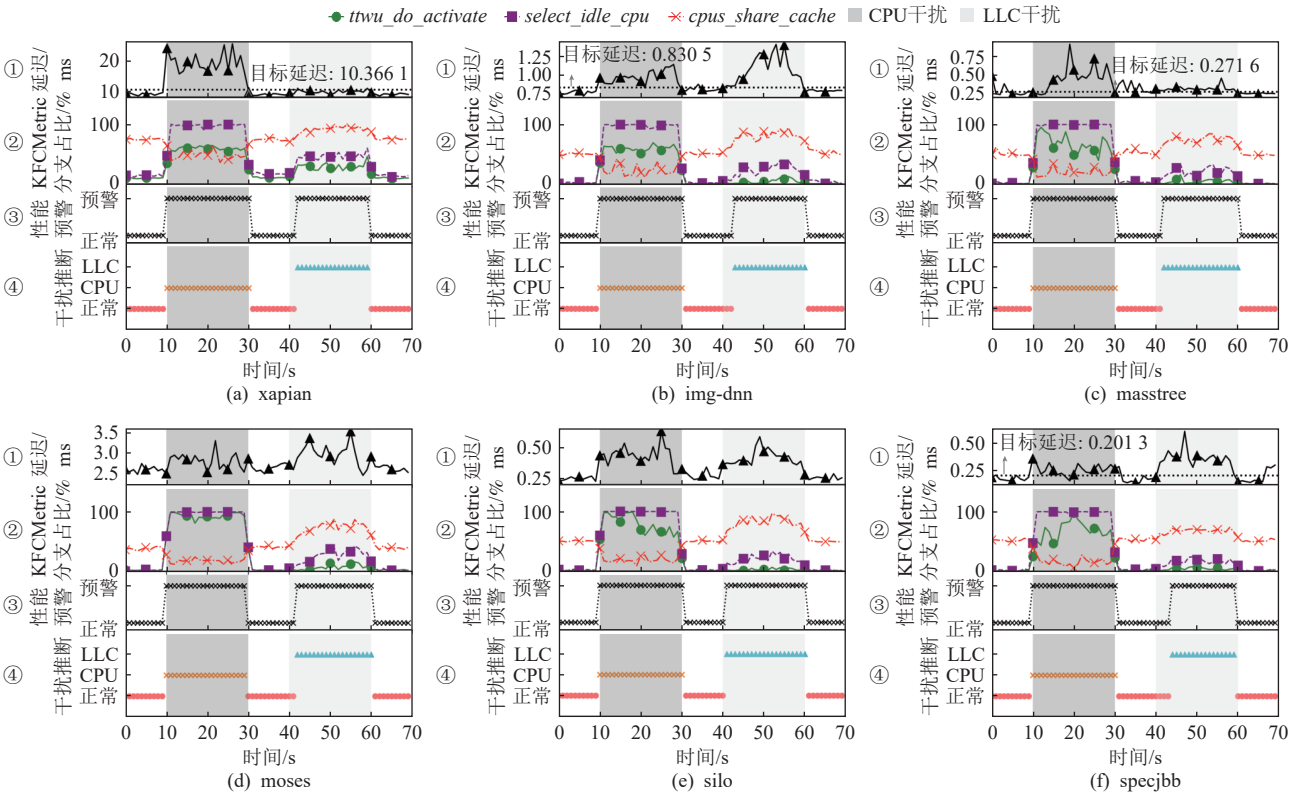


Fig. 10 Variation trends of latency, KFCMetric, interference detection results, and inference on resource bottlenecks

图 10 延迟、KFCMetric、干扰检测结果及资源瓶颈推断的变化趋势

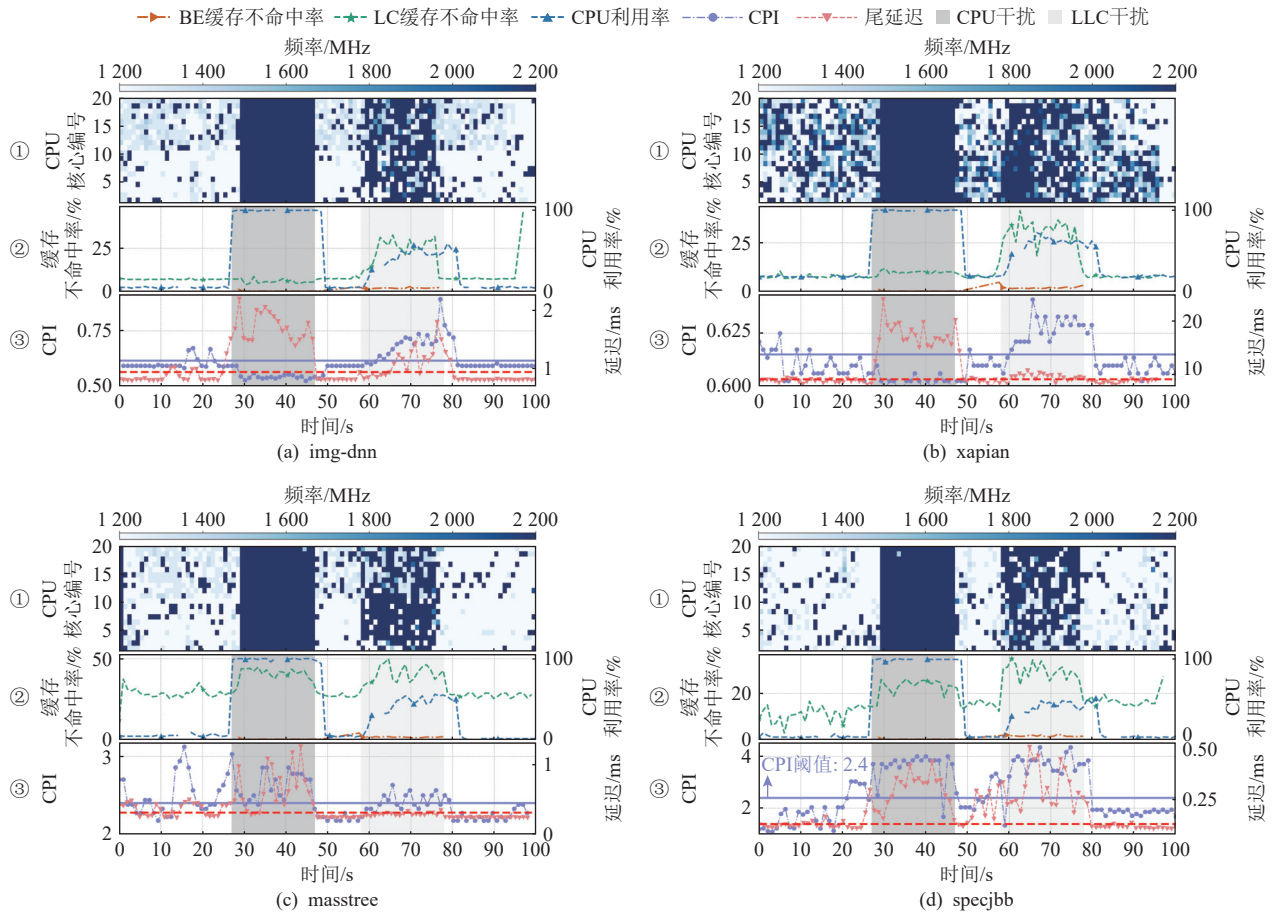


Fig. 11 Variation trends in LLC Cache Misses, CPU utilization, CPI, and core frequencies

图 11 缓存不命中率、CPU 利用率、CPI 和核频率的变化趋势

具体而言, 我们在图 11 中通过设定 CPI 阈值来衡量其性能干扰检测的正确率。由于 CPI 在不同应用之间差异显著, 例如 img-dnn 的 CPI 约为 0.6, 而 masstree 和 specjbb 则在 2 左右, 因此需要依据经验为每个应用分别设定阈值(图 11 中③实线水平线所示)。同时, 我们根据经验设定了目标延迟(图 11 中③虚线水平线所示), 代表应用性能的理想状态。基于这些 SLO 设定, 我们统计了以 CPI 为性能指标的干扰检测正确率, 我们定义当应用延迟超过其正常值 10% 即判定为性能下降。此时若 KFCMetric 通过分析关键分支调用占比来判定当前关键应用的性能已经下降, 则为正确检测; 若未检测出, 则记为漏报。当性能未下降, CPI 未超过阈值则记为正确检测, 但如果 KFCMetric 发出了报警, 则记为误报。我们在表 3 中统计了 CPI 与 KFCMetric 作为性能指标的正确率、误报率和漏报率来衡量 KFCMetric 的正确率。

尽管 CPI 指标在特定场景下表现出较高正确率, 例如在 img-dnn 的 LLC 阶段和 masstree 的 CPU 干扰阶段, 其变化趋势与应用延迟高度一致, 但即使我们已经针对各个应用精细调整了 CPI 阈值以优化其性能指示效果, KFCMetric 在各应用上的干扰检测正确率依然更高。具体而言, CPI 在 img-dnn 和 xapian 应用上的干扰检测正确率分别只有 62.89% 和 54.64%, 其主要原因是在混合 CPU 干扰的情况下, CPI 指标反而比正常情况更低, 无法有效识别性能下降。这是由于在 Intel Turbo Boost 模式下, 当 CPU 检测到有较高的计算需求或特定干扰情况(如资源竞争、负载增加)时, 会自动提高核心频率(如图 11 热力图所示), 以尝试抵消性能下降的影响。

Table 3 Accuracy Comparison of KFCMetric and CPI for Interference Detection

表 3 KFCMetric 与 CPI 干扰检测正确率比较 %

应用	CPI 指标			KFCMetric 指标		
	正确率	误报率	漏报率	正确率	误报率	漏报率
img-dnn	62.89	6.19	30.93	94.29	2.86	2.86
xapian	54.64	12.37	32.99	74.29	0.00	25.71
masstree	79.38	12.37	8.25	80.00	14.29	5.71
specjbb	79.38	6.19	14.43	85.71	10.00	4.29

因此，在干扰检测方面，KFCMetric 较 CPI 具有更高的正确率，最高可达 31.40 个百分点的差值。另外，CPI 只能用于指示性能，没有办法用来指示当前受到的干扰类型，因此，我们在下面引入 CPU 资源利用率和缓存不命中率来衡量 KFCMetric 在干扰类型诊断方面的正确性。

实验 2. 资源瓶颈指示精确度评估。实验主要评估 KFCMetric 与传统的资源利用率指标（包括 CPU 利用率和缓存不命中率）在资源瓶颈诊断中的表现。

传统的资源利用率指标，如 CPU 利用率和 LLC 缓存不命中率，提供的是一种聚合层面的资源占用情况。这类指标在单一干扰条件下可以直观反映系统中资源的紧缺状态，但它们存在以下 2 个局限：尽管传统指标能够大体上反映出当前紧缺的资源类型，但它们也存在不足之处。首先，如图 11(c)所示，对于 masstree 这类自身缓存不命中率较高的应用，单纯依赖缓存不命中率指标难以准确判断其性能状态。如果以缓存不命中率作为性能指标将会导致严重的误报。另一方面，在实际环境中，应用往往同时面临来自 CPU 和缓存等多维度的干扰，图 11 中在 LLC 干扰的阶段，img-dnn, xapian, specjbb 的 CPU 资源利用率也会超过 20%，另外 specjbb 在 CPU 干扰阶段的缓存不命中率也会达到 25%。因此，单一指标很难确定这种多维度资源影响下的瓶颈状态。

KFCMetric 能够高准确度地识别应用所争抢的资源。从图 10 中④可以看出，在干扰阶段的漏报主要是因为性能水平被判定为无干扰，从而不会进行进一步的资源瓶颈的诊断。如果按照在相应的干扰阶段，直接判定为对应的瓶颈，那么 img-dnn, xapian, silo, masstree, specjbb, moses 的资源瓶颈正确率分别是 95.7%, 97.1%, 98.6%, 97.1%, 94.3%, 97.1%。平均的资源瓶颈识别率为 96.7%。

3.3 可扩展性评估实验

为了评估 KFCMetric 方案的可扩展性，本实验从跨内核版本的适应性、新分支的有效性和延迟敏感型应用场景 3 个维度展开评估。如图 12 所示，所有实验均在公有云环境下基于 Linux 6.12 内核版本完成。首先，我们评估了 KFCMetric 关键分支在跨内核版本与配置环境下的适应性。4 组实验结果表明，在已分析的 Linux 5.15 版本的 3 个 KFCMetric 关键内核函数 *select_idle_cpu* 分支在 4 个应用场景中的调用频率特征依旧符合预期：当存在混部 CPU 干扰或 LLC 干扰时，该分支的调用占比均上升。其他分支也得到了部分实验验证，例如，在 img-dnn 应用中，函数 *ttwu_do_activate* 关键分支呈现类似表现；而在 silo 中，*ttwu_do_activate* 关键分支在 CPU 干扰下调用频次降低、在 LLC 干扰下调用频次上升，这些均印证了 KFCMetric 在不同内核版本和系统环境下均能有效指示当前资源瓶颈。

其次，我们评估新筛选出的 KFCMetric 关键分支的有效性。由于 KFCMetric 依赖于内核版本，在新环境部署时需要重新进行静态分析以提取新的关键分支。通过静态分析，我们进一步筛选出由函数 *try_to_wake_up* 调用函数 *set_task_cpu* 分支的占比，并验证其在更多路径下对干扰类型的识别能力。在 img-dnn, silo, nginx, redis 等应用中，函数 *set_task_cpu* 分支同

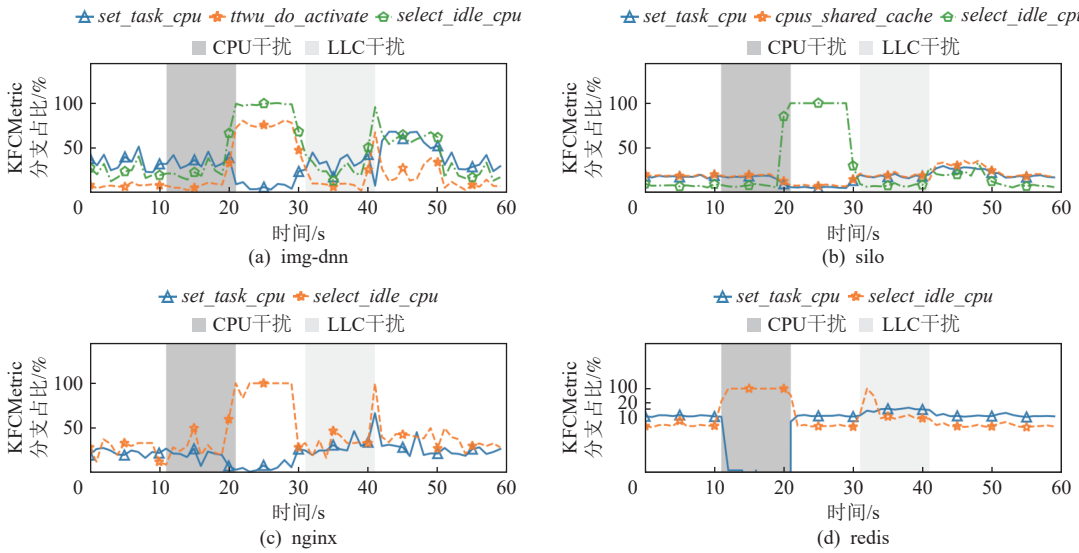


Fig. 12 Variation trends of KFCMetric in public cloud environments

图 12 KFCMetric 在公有云环境下的变化趋势

样与应用的资源类型密切相关: 在受到 CPU 干扰时, 其调用占比下降, 而受 LLC 干扰时调用占比上升, 从而可以用于识别资源瓶颈类型。该结果表明, 我们提出的静态分析方法能够跨内核版本有效识别具备资源类型敏感性的关键分支。

最后, 为进一步验证方案在不同应用类型下的扩展性, 我们在新版本公有云环境中又测试了 nginx 和 redis。nginx 是一种高性能 Web 服务器, 而 redis 是一种内存数据库。函数 *set_task_cpu* 与函数 *select_idle_cpu* 这 2 个关键内核分支进行资源瓶颈诊断的结果显示, KFCMetric 在 nginx 和 redis 上的诊断准确率分别达到了 88.14% 和 93.22%。这些数据充分证明, KFCSys 方案在不同应用场景下均表现出优异的扩展性和跨环境适应能力。

3.4 性能评估实验

为评估 KFCMetric 在资源管理中的效果, 我们基于该指标实现了运行时资源管理系统 KFCSys, 并与 DefaultSys 和 PARTIES 进行对比。实验中采用控制变量法, KFCSys 与 PARTIES 使用相同的资源隔离方案, 均通过 Cgroup Cpuset 进行 CPU 核心的分配, 通过 CAT 技术实现 LLC 缓存的划分。但二者的区别在于性能指标不同: PARTIES 使用的是尾延迟, 且由于缺乏资源瓶颈信息, 其资源调整采用轮转置换方案。

如图 13 所示, 每个实验时长为 50 s, 在第 20 秒~40 秒引入了 CPU 干扰负载。在资源干扰初期, 由于缺乏性能分析数据, KFCMetric 未能提前预测干扰, 导致尾延迟出现剧烈波动。在检测周期内(本实验为 1 s), KFCSys 能够及时识别出 CPU 资源瓶颈, 并立即调用 CPU 子控制器进行资源隔离, 优先保障 LC 应用性能, 从而迅速降低尾延迟。在干扰阶段的资源隔离窗口内, 这种机制持续有效, 有效避免了应用间资源争抢, 保证了较低的尾延迟; 但窗口结束后, 如果干扰仍在持续, 隔离机制解除, 尾延迟可能再次出现波动。这种周期性波动可通过缩短监测窗口来缓解。

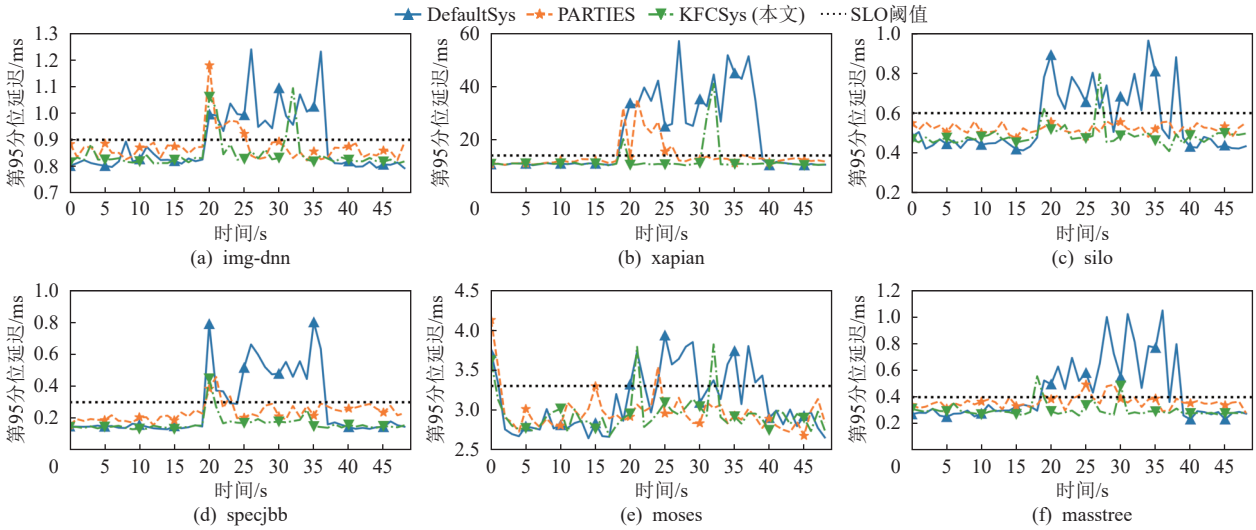


Fig. 13 Performance improvements with DefaultSys, PARTIES, and KFCSys

图 13 DefaultSys, PARTIES, KFCSys 下的性能提升

在最大延迟方面, 我们通过测量混部场景中 LC 应用的最大尾延迟来评估性能, 如图 14 所示。由于 3 个系统均为实时性能监测框架, 资源调度动作是在延迟已升高后才进行的, 因此均无法避免干扰初期的初期的延迟升高, 这导致所有系统的最大尾延迟较高。

具体而言, xapian, img-dnn, moses 这 3 个应用的最大尾延迟在各系统下的差异较小; 而 specjbb, silo, masstree 应用, 由于 DefaultSys 缺乏资源调控手段, 最大尾延迟较高, 这是因为 DefaultSys 在干扰阶段无法避免 BE 应用与 LC 应用之间的资源竞争。

在平均延迟方面, 基于 KFCMetric, KFCSys 的资源管理能力优于 DefaultSys 和 PARTIES。图 13 进一步量化了尾延迟的优化效果, 结果表明, 所有应用在 KFCSys 下的平均尾延迟均值最低。其中 moses 性能提升最小, 其在 DefaultSys, PARTIES, KFCSys 下的平均尾延迟分别为 3.11 ms, 2.98 ms, 2.94 ms, 相较于 DefaultSys 和 PARTIES 这 2 个系统分别提升了 5.5% 和 1.3%。

而性能提升最大的应用为 specjbb, 其平均尾延迟在 KFCSys 下仅为 0.16 ms, 显著低于 PARTIES 的 0.23 ms 与 DefaultSys 的 0.27 ms, 性能提升幅度分别

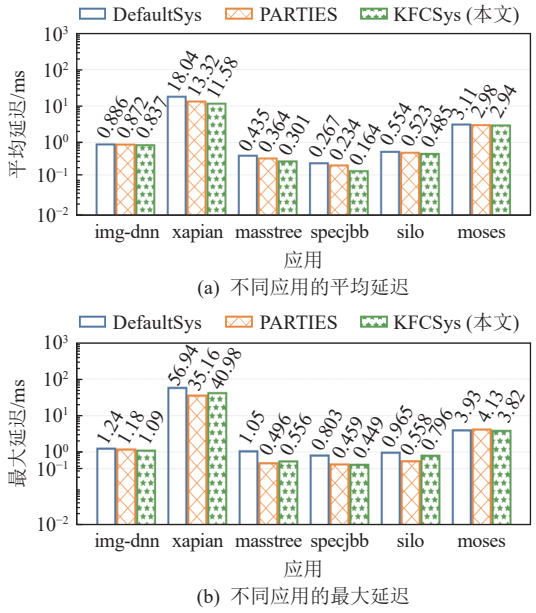


Fig. 14 Latency comparison of different resource management systems

图 14 不同资源管理系统的延迟对比

达到 30.4% 和 40.7%。因此，基于 KFCMetric 的 KFCSys 能够显著改善应用性能。

在性能恢复能力方面，PARTIES 的尾延迟恢复所需的时间和幅度均高于 KFCSys，原因是 PARTIES 采用资源轮转置换策略，其性能指标仅反映当前应用性能而不携带资源瓶颈信息。当出现干扰时，PARTIES 通过随机调整 CPU，LLC，Power 资源，以试探性方案决定资源分配。这种策略可能导致性能恢复周期较长。相较之下，KFCSys 能精准识别资源瓶颈，迅速判断 CPU 资源竞争并立即实施相应的资源调整。我们引入了资源违例率这一指标，其定义为应用尾延迟超出 SLO(service level object)阈值(如图 13 虚线所示)的累计时间占总运行时间的比例。如图 15 所示，由于干扰阶段持续 20 s，运行时间为 50 s，所以 DefaultSys 的违例率集中于 30%~40%。KFCSys 的违例率普遍低于 PARTIES，其中 silo 的违例率为 0。以 xapian 为例，KFCSys 较 PARTIES 降低了 10 个百分点的违例率。

综上所述，相较于基于实时性能指标的 PARTIES 调度框架，KFCMetric 能够精确地识别资源瓶颈，使得资源调度更快速、更高效，显著提高了混部场景下 LC 应用的性能。KFCSys 的尾延迟违例程度和违例时间均有所降低，更好地保障了应用性能满足 SLO 要求。

3.5 案例研究

KFCSys 能够准确识别干扰类型，因此其调度策略通常可以在 2 s 内恢复应用性能。图 16 展示了

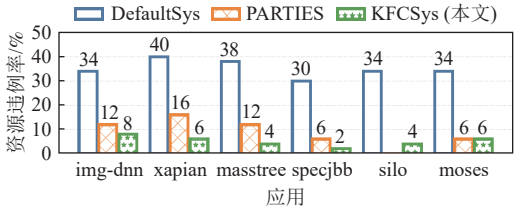


Fig. 15 Performance violation rate comparison of different resource management systems

图 15 不同资源管理系统的性能违例率对比

KFCSys 资源管理框架下，xapian 应用在混部 CPU 和 LLC 干扰场景中的调度细节。图 16 中①显示了应用的第 95 分位尾延迟，图 16 中②展示了 3 个关键分支占比的变化趋势。

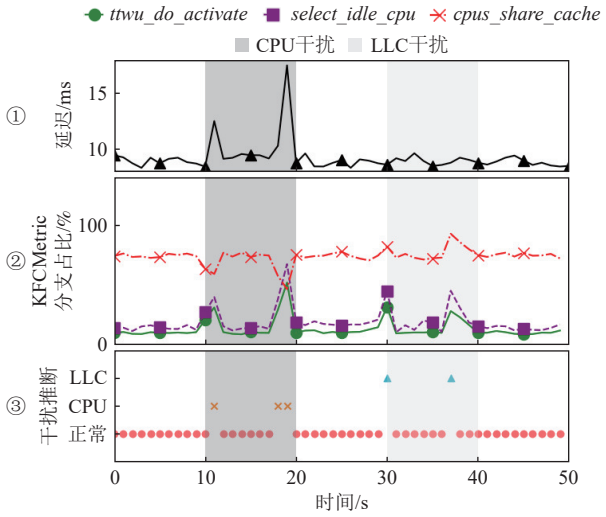


Fig. 16 Illustration of KFCSys scheduling

图 16 KFCSys 调度示意图

第 10 秒时，KFCSys 监测到函数 *ttwu_do_activate* 的调用占比显著上升，监测模块推断性能下降，并且根据另外 2 个关键函数占比的调用特征进一步推断当前为 CPU 干扰。随后，KFCSys 调用了 CPU 子控制器，通过 Cgroup 调整 CPU 核心分配。在第 12 秒的时候，性能恢复到正常水平。在接下来的 5 s 内，由于采用气泡 stall 控制，该状态会一直持续，直到第 17 秒恢复资源共享状态。然而，在第 18 秒时，KFCSys 再次监测到 CPU 干扰，并迅速重新启动隔离措施，再次成功恢复应用的正常性能。

随后，在第 30 秒，KFCSys 以函数 *ttwu_do_activate* 的调用占比为性能指标判定发生了性能波动，并结合函数 *cpus_share_cache* 占比和函数 *select_idle_cpu* 占比的上升，进一步确认缓存争用是主要性能瓶颈。为解决这一问题，KFCSys 调用了 LLC 子控制器，利用 RDT 技术将缓存块进行隔离，从而保障 xapian 应

用的缓存资源不再受到干扰应用的争用。此后,应用的性能迅速恢复至正常水平,尾延迟显著下降。

这一实验表明,KFCSys 由于有资源瓶颈信息辅助,所以能够快速精准地识别应用性能的干扰类型,并通过子控制器的针对性干预迅速恢复 LC 应用性能。这样的动态资源管理机制显著提高了资源利用效率,同时有效保障了关键应用的服务质量。

3.6 运行时开销

KFCSys 使用 eBPF 动态插桩来测量内核关键分支的占比,从而实现实时性能监控。插桩引入的性能开销取决于 eBPF 探针在应用请求处理过程中所占的时间比例。表 2 列出了所选延迟敏感型应用的基本特征,其中 masstree, silo 的请求延迟在微秒级别,明显低于其他毫秒级应用(如 img-dnn, moses, specjbb, xapian, redis, nginx)。

图 17 对比展示了在启用(KFCSys)和禁用 KFCMetric(DefaultSys)插桩情况下,不同应用的平均延迟和第 95 分位尾延迟表现。从图 17 中数据可见,插桩对延迟的影响与应用自身的延迟规模紧密相关。例如,silo 和 masstree 等微秒级应用的平均延迟分别增加了约 2.72% 和 2.35%,而 moses, img-dnn, specjbb, xapian 等毫秒级应用的平均延迟增加 1.15%~4.59%。其中,specjbb 的延迟增幅最大,为 4.59%;而 xapian 最低,仅为 1.15%。这表明虽然总体而言插桩开销较低,但对于延迟极短的微秒级应用,相对开销更为显著。

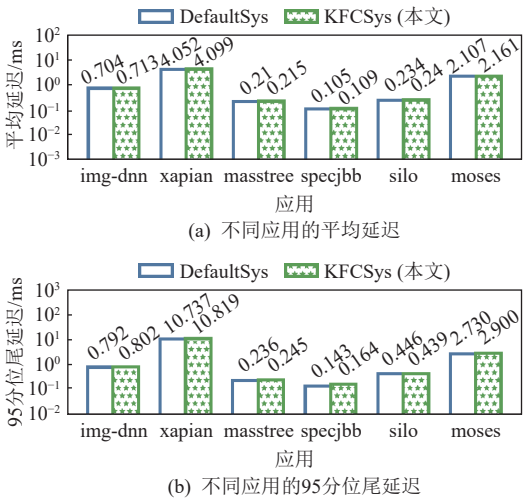


Fig. 17 Performance overhead of KFCSys instrumentation

图 17 KFCSys 插桩的性能开销

从第 95 分位尾延迟的表现来看,多数应用的变化幅度仍在可接受的范围内。具体而言,img-dnn 和 xapian 的 95 分位尾延迟几乎未见明显变化,甚至 silo 在启用 KFCSys 后 95 分位尾延迟略有降低。这说明插桩引入的尾延迟波动并不会显著加剧。

因此 eBPF 插桩所引入的运行时开销在毫秒级应用中极低,而在微秒级应用中相对更明显,但总体保持在 5% 以内。综上所述,在多应用混合部署的云环境中,KFCSys 的插桩开销相较于其他资源竞争和干扰因素导致的性能波动而言可以忽略不计,整体性能影响在实际部署场景中是可接受的。

4 总 结

本文提出了一种基于内核函数调用频率的性能监控指标——KFCMetric,旨在提升云服务提供商在多应用混合部署环境下的资源利用率并减轻性能干扰。针对传统监控方法的局限性,KFCMetric 通过实时分析内核函数的调用动态变化,能够间接反映硬件资源波动并准确识别资源瓶颈,从而为云应用提供高效的资源调度方案。与现有方法相比,KFCSys 在降低尾延迟和资源瓶颈识别方面具有显著优势,尤其在动态负载场景下展现出更高的正确率和实时性。总体而言,KFCSys 的核心贡献在于提供了一种无需依赖硬件或应用层支持的通用性能监控手段,简化了资源调度过程,提升了服务质量保障能力,特别适用于复杂且多变的云计算环境。

作者贡献声明:黄紫微负责设计与实验;姚益祁负责问题调研;包云岗负责论文整体指导;王卅完成审阅修改、项目管理和思路构思。

参 考 文 献

[1] Nathuji R, Kansal A, Ghaffarkhah A. Q-clouds: Managing performance interference effects for QoS-aware clouds [C] // Proc of the 5th European Conf on Computer Systems. New York: ACM, 2010: 237−250

[2] Datacenter Dynamics. Study: Only 13% of provisioned CPUs and 20% of memory utilized in cloud computing[EB/OL]. [2024-11-28]. https://www.datacenterdynamics.com/en/news/only-13-of-provisioned-cpus-and-20-of-memory-utilized-in-cloud-computing-report/

[3] Ferdman M, Adileh A, Kocberber O, et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware[J]. ACM SIGPLAN Notices, 2012, 47(4): 37−48

[4] Hauswald J, Kang Y, Laurenzano M A, et al. Djinn and tonic: DNN as a service and its implications for future warehouse scale computers[J]. ACM SIGARCH Computer Architecture News, 2015, 43(3S): 27−40

[5] Marnic G. Bubble up: Alternative approaches to research in the

academic architecture studio[J]. International Journal of Architectural Research, 2010, 4(2/3): 61−75

[6] Delimitrou C, Kozyrakis C. Hcloud: Resource-efficient provisioning in shared cloud systems[C] //Proc of the 21st Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2016: 473−488

[7] Zhang Xiao, Tune E, Hagmann R, et al. CPI2: CPU performance isolation for shared compute clusters[C] //Proc of the 8th ACM European Conf on Computer Systems. New York: ACM, 2013: 379−391

[8] Chen Quan, Xue Shuai, Zhao Shang, et al. Alita: Comprehensive performance isolation through bias resource management for public clouds[C] //Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis(SC'20). Piscataway, NJ: IEEE, 2020: 1−13

[9] Chen Shuang, Delimitrou C, Martínez J F. PARTIES: QoS-aware resource partitioning for multiple interactive services[C]//Proc of the 24th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2019: 107−120

[10] Hsu W, Smith A J, Young H C. Characteristics of production database workloads and the TPC benchmarks[J]. IBM Systems Journal, 2001, 40(3): 781−802

[11] Xia Sibo, Ma Minghua, Jin Pengxiang, et al. Search service response time anomaly diagnosis[J]. Journal of Computer Research and Development, 2024, 61(6): 1573−1584 (in Chinese)
(夏思博, 马明华, 金鹏翔, 等. 搜索服务响应时间异常诊断[J]. 计算机研究与发展, 2024, 61(6): 1573−1584)

[12] Qu Luyi. Performance evaluation suite for distributed transaction databases [D]. Shanghai: East China Normal University, 2023 (in Chinese)
(瞿璐祎. 分布式事务型数据库的评测套件[D]. 上海: 华东师范大学, 2023)

[13] Lu Haitao, Ren Xiang, Zhong Weijun, et al. CPUBench: An open-source general CPU performance benchmark tool [J]. Microelectronics and Computer, 2023, 40(5): 75−83 (in Chinese)
(逯海涛, 任翔, 钟伟军, 等. CPUBench: 一款开放的通用计算 CPU 性能基准工具[J]. 微电子学与计算机, 2023, 40(5): 75−83)

[14] Shi Huikang, Wang Zesheng, Zhang Shizong, et al. General CPU performance benchmark research: A review[J]. Electronic Journal, 2023, 51(1): 246−256 (in Chinese)
(史惠康, 王泽胜, 张士宗, 等. 通用 CPU 性能基准测试研究综述[J]. 电子学报, 2023, 51(1): 246−256)

[15] Liu Shuai, Qiao Ying, Luo Xiongfei, et al. Key technologies in time-series databases: A review[J]. Computer Research and Development, 2024, 61(3): 614−638 (in Chinese)
(刘帅, 乔颖, 罗雄飞, 等. 时序数据库关键技术综述[J]. 计算机研究与发展, 2024, 61(3): 614−638)

[16] Liu Bingbing, Meng Xiaofeng, Shi Yingjie. CloudBM: Cloud data management system test benchmark[J]. Journal of Computer Science and Exploration, 2012, 6(6): 504−512 (in Chinese)
(刘兵兵, 孟小峰, 史英杰. CloudBM: 云数据管理系统测试基准[J]. 计算机科学与探索, 2012, 6(6): 504−512)

[17] Xia Jiandong, Wang Lechun, Gong Zhenghu. Research on computer network performance benchmarking[J]. Computer Engineering and Applications, 2000, 36(9): 31−33 (in Chinese)
(夏建东, 王乐春, 龚正虎. 计算机网络性能基准测试研究[J]. 计算机工程与应用, 2000, 36(9): 31−33)

[18] Meisner D, Gold B T, Wenisch T F. Powernap: Eliminating server idle power[J]. ACM SIGARCH Computer Architecture News, 2009, 37(1): 205−216

[19] Pelley S, Meisner D, Zandevakili P, et al. Power routing: Dynamic power provisioning in the data center[J]. ACM SIGARCH Computer Architecture News, 2010, 38(1): 231−242

[20] Jing Yinan, Zhang Hanbing, Li Zhixin, et al. Research on next-generation database test benchmarks for financial scenarios[J]. Chinese Engineering Science, 2022, 24(4): 121−132 (in Chinese)
(荆一楠, 张寒冰, 李智鑫, 等. 面向金融场景的下一代数据库测试基准研究[J]. 中国工程科学, 2022, 24(4): 121−132)

[21] Wang Lei, Zhan Jianfeng, Luo Chunjie, et al. BigDataBench: A big data benchmark suite from Internet services[C] // Proc of the 20th Symp on High performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2014: 488−499

[22] Phanekham D, Zaber M, Nair S, et al. Measuring cloud network performance with PerfKit Benchmarker [EB/OL]. (2020-01-13)[2024-12-25]. https://services.google.com/fh/files/misc/measuring-cloud-network-performance-perfkit-wp.pdf

[23] Gan Yu, Zhang Yanqi, Cheng Dailun, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems[C] //Proc of the 24th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2019: 3−18

[24] Mars J, Tang Lingjia, Hundt R, et al. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations[C] //Proc of the 44th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2011: 248−259

[25] Delimitrou C, Kozyrakis C. iBench: Quantifying interference for datacenter applications[C] // Proc of the 2013 IEEE Int Symp on Workload Characterization (IISWC). Piscataway, NJ: IEEE, 2013: 23−33

[26] Delimitrou C, Kozyrakis C. Paragon: QoS-aware scheduling for heterogeneous datacenters[J]. ACM SIGPLAN Notices, 2013, 48(4): 77−88

[27] Delimitrou C, Kozyrakis C. Quasar: Resource-efficient and QoS-aware cluster management[J]. ACM SIGPLAN Notices, 2014, 49(4): 127−144

[28] Novaković D, Vasić N, Novaković S, et al. DeepDive: Transparently identifying and managing performance interference in virtualized environments[C] // Proc of the 2013 USENIX Annual Technical Conf (USENIX ATC 13). Berkeley, CA: USENIX Association, 2013: 219−230

[29] Lee J, Kim C, Lin K, et al. WSMeter: A performance evaluation methodology for Google's production warehouse-scale computers[C] // Proc of the 23rd Int Conf on Architectural Support for Programming Languages and Operating Systems. New York : ACM, 2018: 549−563

[30] Patel T, Tiwari D. CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers[C] //

Proc of the 26th Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2020: 193−206

[31] Chen S, Jin A, Delimitrou C, et al. Retail: Opting for learning simplicity to enable QoS-aware power management in the cloud[C] // Proc of the 28th Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2022: 155−168

[32] Zheng Zibin, Ma Hao, Lyu M R, et al. QoS-aware web service recommendation by collaborative filtering[J]. IEEE Transactions on Services Computing, 2010, 4(2): 140−152

[33] Lo D, Cheng Liqun, Govindaraju R, et al. Heracles: Improving resource efficiency at scale[C]//Proc of the 42nd Annual Int Symp on Computer Architecture. New York: ACM, 2015: 450−462

[34] Haque M E, He Y, Elnikety S, et al. Exploiting heterogeneity for tail latency and energy efficiency[C] //Proc of the 50th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2017: 625−638

[35] Belay A, Prekas G, Klimovic A, et al. IX: A protected dataplane operating system for high throughput and low latency[C] // Proc of the 11th USENIX Symp on Operating Systems Design and Implementation (OSDI'14). Berkeley, CA: USENIX Association, 2014: 49−65

[36] Ren G, Tune E, Moseley T, et al. Google-wide profiling: A continuous profiling infrastructure for data centers[J]. IEEE Micro, 2010, 30(4): 65−79

[37] Ousterhout A, Fried J, Behrens J, et al. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads[C] // Proc of the 16th USENIX Symp on Networked Systems Design and Implementation (NSDI'19). Berkeley, CA: USENIX Association, 2019: 361−378

[38] Fried J, Ruan Z, Ousterhout A, et al. Caladan: Mitigating interference at microsecond timescales[C] // Proc of the 14th USENIX Symp on Operating Systems Design and Implementation (OSDI'20). Berkeley, CA: USENIX Association, 2020: 281−297

[39] Lee J, Min D, Byun I, et al. Fast, light-weight, and accurate performance evaluation using representative datacenter behaviors [C] //Proc of the 24th Int Middleware Conf. New York: ACM, 2023: 220−233

[40] Shanley T. Pentium Pro and Pentium II System Architecture[M]. Reading, MA: Addison-Wesley Professional, 1998

[41] Contreras G, Martonosi M. Power prediction for Intel XScale® processors using performance monitoring unit events[C] //Proc of the 2005 Int Symp on Low Power Electronics and Design. New York: ACM, 2005: 221−226

[42] Rostedt S. Ftrace Linux kernel tracing[C/OL] //Proc of the Linux Conf Japan. 2010[2024-11-28]. https://events.static.linuxfound.org/slides/2010/linuxcon_japan/linuxcon_jp2010_rostedt.pdf

[43] Intel Corporation. Improving real-time performance by utilizing cache allocation technology [EB/OL]. (2015-04-15) [2024-12-25]. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf

[44] Sohal P, Bechtel M, Mancuso R, et al. A closer look at Intel resource director technology (RDT)[C] //Proc of the 30th Int Conf on Real-Time Networks and Systems. New York: ACM, 2022: 127−139

[45] Sofia Z. Container technologies[J]. Hypatia, 2000, 15(2): 181−201

[46] King C I. Stress-ng[CP/OL]. 2017[2024-12-25]. http://kernel.ubuntu.com/git/cking/stressng. git/

[47] Kasture H, Sanchez D. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications[C] // Proc of the 2016 IEEE Int Symp on Workload Characterization (IISWC). Piscataway, NJ: IEEE, 2016: 1−10



Huang Ziwei, born in 1994. PhD. Student member of CCF. Her main research interest includes data center computer architecture.

黄紫微, 1994 年生。博士。CCF 学生会员。主要研究方向为数据中心体系结构。



Yao Yiqi, born in 2001. Master candidate. Student member of CCF. His main research interests include cloud computing and kernel schedulers of tasks.

姚益祁, 2001 年生。硕士研究生。CCF 学生会员。主要研究方向为云计算、内核任务调度器。



Bao Yungang, born in 1980. PhD, professor. Member of CCF. His main research interests include datacenter architecture, agile design methodology of processor chips, and ecosystem of open-source processor chips.

包云岗, 1980 年生。博士，研究员。CCF 会员。主要研究方向为数据中心体系结构、处理器芯片敏捷设计方法论、开源芯片生态。



Wang Sa, born in 1986. PhD, associate professor. Member of CCF. His main research interests include cloud computing, operating systems, and system modeling and performance analysis.

王　卅, 1986 年生。博士，副研究员。CCF 会员。主要研究方向为云计算、操作系统、系统建模与性能分析。