计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(11): 2826−2842, 2025

基于动态插桩的网络功能软件性能测量方法

贾 茹[1,2] 姜海洋[1] 李振宇[1] 谢高岗[3]
[1](中国科学院计算技术研究所 北京 100190)
[2](中国科学院大学 北京 100049)
[3](中国科学院计算机网络信息中心 北京 100083)
(jiaru@ict.ac.cn)

Dynamic Instrumentation Based Performance Measurement for Network Functions Software

Jia Ru[1,2], Jiang Haiyang[1], Li Zhenyu[1], and Xie Gaogang[3]
[1](*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)
[3](*Computer Network Information Center*, *Chinese Academy of Sciences*, *Beijing* 100083)

Abstract Softwareization of network function (NF) provides flexibility for the implementation and deployment of new network applications. However, duo to more complex program structure and running environment compared with NF hardware, NF software introduces various performance issues, such as, short-term throughput anomalies and long-tail delays, and degrades user experience. Once NF performance problem occurs, it is necessary to quickly locate problematic modules and determine the cause of the problems through performance measurement. Facing to complex operating environments of NF, increasingly expanding code size, and diverse root causes of problems, coarse-grained performance measurement cannot meet the requirement of problem location and analysis. More efficient fine-grained NF performance measurement is necessary. For the two types of widely used NF performance measurement methods: sampling-based and instrumentation-based, we first prove through actual measurement analysis that, the sampling-based performance measurement method is not suitable for fine-grained NF performance measurement, and the instrumentation-based method will generate a large amount of additional measurement overhead, affecting the measurement results. To this end, we propose a function-level dynamic instrumentation method that combines dynamic library piling and function-level fast breakpoints. Compared with static instrumentation, dynamic instrumentation can execute instrumentation on demand in runtime. It is more suitable for use in the production environment. Our dynamic instrumentation method reduces the instrumentation overhead by an average of 70% compared with baseline fast breakpoints. On this basis, we design and implement the packet-level NF performance measurement method LProfile, based on lightweight probes and storage optimization. Compared with TAU, a general-purpose performance measurement tool, LProfile reduces the single-point measurement overhead by 82%.

Key words network function softwareization; performance measurement; dynamic instrumentation; fast breakpoint; lightweight probe

收稿日期: 2024-04-01; **修回日期**: 2025-02-24

基金项目: 国家重点研发计划项目(2022YFB2901305); 国家自然科学基金项目(62072430)
This work was supported by the National Key Research and Development Program of China (2022YFB2901305) and the National Natural Science Foundation of China (62072430).

通信作者: 谢高岗(xie@cnic.cn)

摘　要　网络功能（network function, NF）软件化为新型网络场景和应用的实现与部署提供灵活性. 然而，相较于专有硬件，网络功能软件的程序结构和运行环境更复杂，导致短时吞吐异常、长尾时延等各种性能问题，影响用户体验. 当出现性能问题时，需要快速通过性能测量，定位问题所在模块，确定问题产生的原因. 面对NF软件复杂的运行环境、日益膨胀的代码规模、问题根因的复杂多样等问题，粗粒度性能测量已经无法满足性能问题定位和分析的需求，急需高效的细粒度NF软件性能测量方法. 当前NF软件性能测量主要分为基于采样和基于插桩2类方法. 通过实际测量分析证明了基于采样的性能测量方法不适用于细粒度NF软件的性能测量，而基于插桩的方法可以满足细粒度测量的功能需求，但会产生大量的额外测量开销，影响测量准确度. 为此，提出了动态库插桩和函数级快速断点相结合的函数级动态插桩方法. 和静态库插桩相比，动态库插桩可以在NF软件运行过程中实时按需打桩，解决了静态库打桩的灵活性问题；和传统快速断点相比，函数级快速断点的插桩开销平均降低了70%. 在此基础上，设计并实现了数据包级的NF软件性能测量方法LProfile，基于轻量化探针和存储优化化等技术进一步减少测量开销. 对比基线方法TAU，LProfile降低了82%的单点测量开销.

关键词　网络功能软件化；性能测量；动态插桩；快速断点；轻量化探针

中图法分类号　TP393

DOI: 10.7544/issn1000-1239.202440228　　**CSTR:** 32373.14.issn1000-1239.202440228

随着新型网络场景和网络应用的不断出现，传统基于专有硬件的网络中间件（middlebox），面临开发周期长、灵活性差等问题. 学术界和工业界开始将原本由硬件承载的网络功能（network function, NF）软件化[1−2]. 软件化后的网络功能不再依托于专有硬件，而是以运行网络功能的网络功能软件（NF软件）的形式存在.NF软件可以在通用或者专用平台上部署，部署规模可按需灵活调度；同时具有研发周期短、升级快捷与成本低等优势. 但与专有硬件相比，软件实现的网络功能代码逻辑和运行环境更加复杂，引入了性能和稳定性的新挑战[3−4].

NF软件在实际运行过程中，可能会面临短时吞吐异常、长尾时延[5]等各类性能问题，这些性能问题会严重破坏服务级别协议 (service-level agreement, SLA) 和用户体验. 在生产环境中，当出现性能问题时，需要快速通过性能测量定位问题所在模块来分析问题产生的原因. 由于NF软件复杂的运行环境和日益膨胀的代码规模，其性能问题产生的原因复杂多样，如何有效地进行性能测量，已成为NF软件性能诊断和优化过程中的核心问题[6]. 高效的细粒度的性能测量方法可以大幅提升性能诊断的效率.

目前，针对NF软件性能测量的研究工作大多是基于队列监控的，即对出入流量和队列等资源进行实时监控和测量，测量结果用于分析和定位引起性能问题的异常流[6−9]，这类方法无法定位NF内部导致异常的模块或代码块. 针对NF软件内部的性能测量，部分研究使用静态代码和仿真环境近似测量NF软件的性能[10−11]，然而日益复杂的CPU架构使得仿真很难接近真实数据，无法准确反映实时性能问题[11].

工业界大多直接使用基于CPU的硬件性能计数器（performance monitoring counters, PMC）的性能测量工具[12]，对NF软件进行细粒度的性能测量. PMC可以从硬件层面对运行在CPU上的进程进行性能数据测量，记录其在运行过程中触发的性能事件的次数，如指令执行、Cache未命中等. 根据其运行模式的不同，衍生出了两大类通用性能测量方法：

1）基于采样的性能测量方法，如Linux Perf[13]，Intel VTune[14]等. PMC运行在采样模式下，当性能事件发生的次数达到阈值时产生中断，中断处理器进行性能数据读取，并写入缓存供用户态工具使用. 测量工具对原数据进行分析并给出性能报告. 此类方法引入的全局开销相对较低且简单易用，目前使用得最为广泛. 然而由于硬件采样机制的限制，无法实现数据包级性能追踪.

2）基于插桩的性能测量方法，如Score-P[15]，TAU[16]，ADEL23[17]等. PMC运行在计数模式下，仅在硬件寄存器中统计性能事件发生的次数. 用户态测量工具实现测量探针，在探针中主动调用PMC API进行数据采集和记录，并通过插桩技术将探针注入到被测程序中. 该类方法具有测量的粒度和范围可控的优势，可以实现数据包级细粒度的性能测量和追踪，但现有方法会引入大量的测量开销. 本文评测了Score-P和TAU这2种典型的基于插桩的方法的单函数测量开销，如表1所示.

Table 1 Overhead Comparison of Common Measurement Tools and High Performance NF Software

表 1 常见测量工具与高性能 NF 软件的耗时对比

软件类型	应用场景	耗时/μs
高性能 NF 软件	NAT（5 000 Mbps）	4.43
	FW（1 500 Mbps）	5.04
性能测量工具	Score-P[15]	6.31
	TAU[16]	1.21

以高性能 NAT(Cisco VPP[18])为例, NF 软件本身每次数据包处理的耗时约为 4.43 μs, 而 Score-P 的单函数测量开销为 6.31 μs, 假设该函数在每次数据包处理过程中执行 1 次, 那么仅仅这一个函数的测量开销就已经是原数据包处理耗时的 142%. 实际上, 在一次完整数据包处理过程中, 需要经历几十甚至上百次函数执行, 过大的单函数测量开销会严重干扰被测程序的性能行为.

本文面对基于插桩的方法开销大的挑战, 进一步对其开销的构成进行深入分析和评估. 该类方法的开销主要由 2 个部分构成: 探针本身的性能数据收集和记录操作; 用于注入探针的插桩技术引入的开销. 基于此, 从探针轻量化和插桩技术优化这 2 个角度出发, 设计并实现了一种基于动态插桩的数据包级 NF 软件性能测量方法 LProfile.

本文创新性地提出了动态库打桩和函数级快速断点相结合的函数级动态插桩方法, 在保证易用性和可测范围的同时, 大幅减小插桩开销. 在探针自身的优化方面, LProfile 使用用户态指令替代系统调用级的 PMC API, 大幅降低性能数据读取开销; 同时, 简化性能数据记录, 仅保留核心信息, 并优化性能记录的存储机制. 应用上述优化, LProfile 可以在保持低开销和低干扰的前提下, 对 NF 软件的数据包处理进行性能测量和追踪. 最后, 在真实转发场景下, 对现有工具和 LProfile 进行了详细的实验对比, 实验结果展现了 LProfile 的有效性.

本文的主要贡献有 3 点:

1)从 NF 软件性能测量的需求出发, 通过原理分析和真实场景验证, 对现有两大类性能测量方法进行了评估. 提出基于采样的性能测量方法不适用于细粒度 NF 软件性能测量.

2)设计并实现了动态库打桩和快速断点相结合的函数级动态插桩方法. 与传统动态插桩相比, 插桩开销平均降低了 70%, 同时解决了静态插桩操作复杂、难以应用的问题.

3)利用函数级动态插桩方法, 设计并实现了数据包级的 NF 软件性能测量方法 LProfile.通过轻量化探针和存储优化等技术, 与 TAU 相比, 将单点测量开销降低了 82% 以上.

1 NF 软件性能测量研究现状

1.1 NF 软件性能测量的挑战和需求

本文的目标是对 NF 软件本身进行性能测量, 深入探究 NF 软件内部的性能行为和性能问题, 所以首先对 NF 软件的数据通路进行简要的介绍, 并探讨影响其性能的因素.

如图 1 所示, NF 软件的数据通路由多个数据包处理模块组成, 模块之间通过队列(queue)等数据结构传递信息. 数据包接收模块从网卡的接收队列中取出数据包, 放入内存的数据包缓存中. 之后数据包进入解析器(parser), 解析器对数据包的首部进行解析, 执行合法性验证、IP 分片处理等. 解析器处理后, 将合法的数据包传递给下一个模块, 即规则匹配. 根据 NF 的功能需求, 不同类型的规则匹配将被执行, 例如防火墙(firewall, FW)、IPsec 安全网关的访问控制规则匹配, 以及路由、NAT 的 IP 最长前缀规则匹配等. 规则匹配模块对数据包进行了分类, 不同类型的数据包对应不同的后续处理, 例如重写、复制、丢弃或是转发. 对于需要被转发的数据包, 规则匹配模块确认其输出端口, 并传递到对应的转发模块中.最后, 转发模块根据网卡物理端口的处理要求, 将数据包写入到网卡输出队列中, 等待网卡发出.

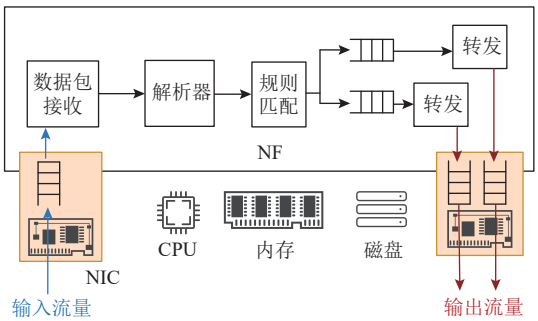


Fig. 1 Data path of NF software

图 1 NF 软件的数据通路

与传统硬件相比, NF 软件的运行环境和性能问题更加复杂[19]. 纵观整个数据包处理过程, 本文将影响 NF 软件的性能因素归为 4 类:

1)模块间的资源竞争. 一个 NF 通常由多个模块组成, 这些模块可能运行在同一个线程上, 也可能并行地运行在多个线程上. 线程内各模块竞争内存和

cache 等资源，线程间进一步竞争 CPU 和锁等资源. 资源的竞争加剧了 NF 软件性能的不稳定，也使得性能测量和分析更加复杂.

2）数据包处理算法的设计实现问题. 数据包处理算法，尤其是各类规则匹配算法的软件实现，一直是网络领域研究的一大热点，通常也是 NF 软件的主要性能瓶颈. 规则匹配算法在不同的数据包和不同的规则集上，性能行为会有较大的差异[20].

3）队列导致的异常传递[6]. 以长尾时延为例，某一数据包的处理时间过长，会使后续若干个数据包长时间排队等待，时延异常问题从单一数据包传递给了后续的多个数据包.

4）操作系统和运行在同一核上的其他进程与 NF 软件实例间的资源竞争. 与独占硬件资源的硬件网络中间设备不同，操作系统和其他运行在服务器上的进程不可避免地会与 NF 软件实例产生资源竞争，进一步复杂化了 NF 软件的性能问题.

通过对 NF 软件的数据通路和性能问题的分析，本文提出了 NF 软件性能测量的 3 个需求：

1）NF 软件的性能影响因素复杂，且与底层硬件资源的使用强相关，仅仅使用队列信息或是流的测量，不足以进行性能问题的深入定位，NF 软件性能测量需要更详细和更底层的性能指标.

2）以 NF 为整体的全局性能数据无法反映 NF 软件内部不同模块和不同函数的性能行为，不利于 NF 软件本身的性能分析和诊断. NF 软件的性能测量需要函数级的测量粒度.

3）NF 软件的性能在不同数据包上存在差异，数据包的大小、内容会明显影响各类数据包处理算法的性能表现，NF 软件的性能测量需要能够分辨这种差异性.此外，NF 软件的性能问题还存在传递性，一个数据包上性能异常通常会影响后续多个数据包的处理.NF 软件性能测量需要实现数据包级的性能追踪，来准确定位性能问题.

此外，随着网络功能虚拟化（network function virtualization，NFV）的普及和飞速发展，5G 等新兴场景不断涌现，为 NF 软件提出了更多、更复杂的功能需求，使得 NF 软件代码飞速膨胀. 本文所使用的 NF 软件编程框架 Cisco VPP 19.08 中，总计包含 2 377 个代码文件、80 893 个函数、45 万行代码. 面对庞大的代码量和复杂的代码结构，依托于经验和手动修改源码的传统性能测量方法难以应用于 NF 软件性能测量. 这对 NF 软件性能测量提出了第 4 个需求，即灵活性和可用性的需求，测量方法需要适用于各类复杂的软件编程框架，避免在测量过程中对 NF 软件进行手动修改.

通用的性能测量方法已经可以方便地测量进程执行过程中的底层硬件资源的使用情况，且易于使用，目前被工业界广泛应用于 NF 软件的性能测量. 但 NF 软件程序和通用计算程序在性能指标上有很大的区别，其性能测量的需求也因此不同. 通用计算程序或高性能计算（HPC）程序的目的是使用尽可能少的资源尽快地完成计算任务. 它们的输入数据集通常是庞大但有限的，核心性能指标大多为全局指标，如任务完成耗时、内存占用率和并行化指标. 以 HPC 常见应用中的气象数值预报[21] 为例，气象数值预报需要面对大量的气象观测数据，求解上百万个自由度的控制偏微分方程组，并在有限的时间内给出预测结果. 其核心评估指标是在整个数据集上完成全部运算的总体开销. 而 NF 与通用计算程序和 HPC 不同，NF 被要求提供长期的、稳定的网络服务，如路由、防火墙、VPN 等. 它持续地接收数据包，面对的是持续的、相对无限的流量. 在评价 NF 软件的性能时，除了高吞吐、低时延等整体性能指标外，服务质量的稳定性和可靠性同样重要[22]. 近几年在线会议、直播、虚拟现实等实时交互应用大规模兴起. 这些应用都需要时延敏感型的底层网络服务，进一步加剧了对 NF 软件的数据包级性能指标的要求[23]. 数据包级的性能测量需求为 NF 软件性能测量带来了极大的挑战，它需要性能测量方法实时追踪数据包处理过程，而不是仅仅统计全局性能信息.

1.2 相关工作

NF 软件相关的性能测量研究主要有队列监控、静态代码分析和 NFV 系统级测量 3 种，目前这 3 类研究均无法实现数据包级和函数级的细粒度 NF 软件性能测量. 而在通用程序和 HPC 的性能测量领域，函数级甚至指令级的基于 PMC 的性能测量方法已经被大量研究，工业界也已经应用于 NF 软件性能测量.

1）NF 软件专用的性能测量方法

①基于队列的 NF 软件性能测量. 目前，部分 NF 软件性能测量工作集中在基于队列的性能监控和测量[6–9]. 队列的存在使得一个数据包或流上的异常，会持续影响后续的若干个数据包或流. 这一类工作对出入 NF 的队列进行实时监控，测量队列内排队长度，记录队列内流的信息，主要应用于异常流的定位. 由于测量并没有深入到 NF 内部，无法定位 NF 软件内部导致异常的模块或代码块.

②基于静态代码分析的 NF 软件性能测量与分

析. Pedrosa 等人[10]引入静态代码分析技术, 在离线环境下遍历所有可能的 NF 执行分支, 生成用于线上压力测试的对抗性工作负载(adversarial workloads)或攻击流量. StateAlyzr[24]使用静态代码分析来计算 NF 软件的流状态和全局状态. 文献 [25−27] 的研究工作使用静态代码分析来进行 NF 软件代码的正确性验证. 以上工作均无法直接地反映 NF 软件实时运行时的性能问题, Bolt[11]在静态代码分析的基础上, 进一步引入了仿真执行, 对 NF 软件的性能进行离线分析和预测. CPU 架构的复杂性和多样性, 使得基于仿真的工作很难接近真实数据, 无法准确反映实时性能问题.

③ NFV 系统级性能测量工作. 目前大量研究工作集中于 NFV 系统级的实时性能测量, 即多个 NF 软件实例间的性能测量. 它们的主要目标是定位整个系统维度上哪个网络节点(即 NF 软件实例)是主要的性能瓶颈. NFVPerf[28]对所有输入和输出的数据包进行镜像和实时监控, 定位 VM-to-VM 通信路径上的瓶颈. PerfSight[19]提取典型软件数据面的关键组件, 通过组件间队列监控流量, 定位瓶颈组件. Pfitscher 等人[29]测量每个 NF 软件实例的 CPU 使用率、CPU 活跃时间和队列信息, 定位 NFV 服务链上的问题 NF. 和这些工作相比, 本文主要关注单一 NF 软件实例的性能测量.

2)基于 PMC 的通用性能测量方法

PMC 是 CPU 硬件中的一组用于追踪硬件性能事件的寄存器. 其可测量的性能指标包括被执行的指令数、运行所需的 CPU 时钟周期数(单位为 cycle)、各级 cache 的 miss/hit、TLB 的 miss/hit 等. 所有基于 PMC 的性能测量方法, 也都支持这些细粒度硬件性能指标, 本文简称为 PMC 指标.

PMC 提供 2 种基本的测量模式: 采样模式和计数模式. 采样模式下, PMC 的处理逻辑被配置为: 硬件性能事件每触发 N 次, 产生一个“overflow”中断, 内核中断处理器负责性能样本的采集, 测量工具负责从内核中取出样本并进行分析, 这也是基于采样的性能测量方法. 受限于采样模式, 该类方法无法实现数据包级的性能测量和追踪.

计数模式下, PMC 仅简单地统计指定性能事件的触发次数, Linux 提供相应的 API 来直接读取 PMC 寄存器数值, 进而衍生出了基于插桩的性能测量方法. 即用户态工具自行实现测量探针, 在探针中主动调用 PMC API 进行数据采集和记录, 并通过插桩方法将探针注入到被测程序中. 该类方法提供了函数级和数据包级性能测量的可能, 但现有方法存在测量开销大和功能支持有限的问题. 表 2 给出了常见的基于 PMC 的性能测量方法.

Table 2 Common PMC-Based Performance Measurement Tools

表 2 常见的基于 PMC 的性能测量工具

类型	测量工具/方法	性能指标	数据包级性能追踪	测量开销
基于采样的方法	Perf[13]	PMC 指标、内核事件	不支持	低
	Intel VTune[14]	PMC 指标、内核事件	不支持	低
	HPCToolkit[30]	PMC 指标、内核事件	不支持	低
基于插桩的方法	HPCToolkit[30] Intel VTune[14]	IO、内存使用量	仅系统库函数	中
	Callgrind[31] DynamoRio[32]	仿真后的硬件指标	支持	中
	Score-P[15] TAU[16]	PMC 指标、内核事件	支持	高
	LProfile(本文)	PMC 指标、内核事件	支持	较低

HPCToolkit[30]和 Intel VTune 仅支持对系统库函数的性能测量, Callgrind[31]和 DynamoRio[32]实现的是仿真环境的测量, 与真实环境相比存在偏差. Score-P 和 TAU 支持更为全面的函数级性能测量, 但均会引入巨额的测量开销. 设计一种新的性能测量方法, 支持对 NF 软件进行全面的函数级性能测量的同时, 大幅降低了其测量开销, 是本文的研究目标.

1.3 NF 软件性能测量: 采样还是插桩

为了对 NF 软件本身进行细粒度性能测量, Perf, Intel VTune 等基于采样的通用性能测量方法目前已被工业界使用. 它们对 NF 软件的全局性能影响相对较小且简单易用, 不需要对 NF 软件进行手动修改. 而基于插桩的性能测量方法, 因其额外开销较大且功能支持有限, 尽管在 HPC[33−34]和分布式系统[35−36]的性能测量中有针对性的研究, 但目前极少用于 NF 软件性能测量. 这 2 类测量方法都可以实现函数级的测量粒度. 除此之外, 如 1.1 节所述, NF 软件性能问题定位还需要数据包级的性能测量信息, 那么, 基于采样的方法和基于插桩的方法哪一种更适用于数据包级细粒度 NF 软件性能测量. 答案是基于插桩的性能测量方法. 下面将从 2 个方面进行论证.

1)基于采样的方法受限于 PMC 采样的固有机制, 无法实现数据包级性能测量. 而基于插桩的方法提供了数据包级性能追踪的可能.

从原理上，PMC 的采样是基于硬件性能事件发生次数的采样，即被测程序每触发 N 次指定的硬件性能事件就进行一次采样. 这种采样得到的结果只能体现 NF 软件在某一段时间内触发的性能事件最多的函数，即“热点”函数. 尽管可以收集采样点的调用栈信息，但不能对应到具体每次的函数执行和数据包处理，也就无法进行数据包级的性能行为追踪. 与此相比，基于插桩的性能测量方法自行构造测量探针，主动决定探针插入的位置，性能数据的读取时机和测量范围明确且可控. 探针的执行和 NF 本身的执行紧耦合，只需要恰当地构造探针和数据记录，即可对 NF 软件进行数据包级追踪.

2）尽管基于插桩的方法会引入较高的测量开销，但其对 NF 数据包级性能行为特征的干扰小于基于采样的方法.

性能测量的目的是通过测量所得的数据尽可能真实地还原被测程序的性能行为，以及尽可能保持原有的性能行为特征. 任何性能诊断测量都会引入额外的操作，这些额外操作除了其本身的开销，还会与被测程序产生资源竞争，势必会干扰被测程序的性能行为. 由于无法在不引入任何额外操作的前提下获取真实性能数据，测量行为对被测程序的干扰程度难以评估，通常直接用全局性能影响来指代[37]. 正如 1.1 节所述，NF 软件拥有数据包级的性能测量需求，因此在讨论测量行为对 NF 软件的干扰程度时，除了整体性能的影响之外，数据包级的性能行为特征是否会被破坏也同样重要. 已有研究工作[38]探讨了基于 PMC 的性能测量方法对 NF 数据包级性能行为特征的干扰程度. 由于采样的触发时间和地点是不确定的，其开销不均匀地分布在每次数据包处理过程中，严重破坏数据包级行为特征. 而基于插桩的方法使用可控的、与数据包处理紧耦合的测量探针，其开销相对均匀地分布在每次数据包处理过程中，对数据包级行为特征的干扰更小.

1.4 插桩方法的研究现状

插桩方法是基于插桩的性能测量方法的核心，决定了测量的可用性，影响测量的开销. 现有插桩方法整体上分为静态插桩和动态插桩两大类. 静态插桩包括手动修改源代码和在编译或链接阶段进行库打桩.Score-P, Scalasca[39], TAU 支持以直接修改源代码的方式插入测量探针，探针部署的灵活性大，可以进行代码行级测量，并覆盖每一个代码块. 然而目前复杂的 NF 程序通常含有上千的函数以及 10 万行以上的代码量，源代码修改的操作量巨大，在实际应用过程中并不适用. HPCToolkit[30], Intel VTune[14], Score-P[15]和部分研究工作[33−36]，对于库函数提供了库打桩的探针注入方法. 在编译或链接时，将待测函数重定向到对应探针函数，不需要额外的桩点代码，测量开销较低，但仅支持系统库函数和预定义的用户库函数.

动态插桩即动态二进制插桩（dynamic binary instrumentation, DBI），是在程序运行时对进程进行插桩，有效避免了繁复的代码修改以及重编译重链接操作，更适用于复杂 NF 软件的性能测量. 目前应用于性能测量的动态插桩技术主要有基于即时编译器和基于快速断点 2 种.

1）基于即时编译器的动态插桩技术

即时编译器（just-in-time compiler, JIT）技术是在运行时将被修改的函数或程序段编译成机器码缓存在内存中，当再次遇到该函数时，直接从缓存中执行已编译好的机器码，从而实现动态代码修改.Intel Pin[40], Valgrind[31], eBPF, Frida[41]等利用即时编译器，将测量探针动态地注入到被测程序中，不需要手动的静态重编译重链接，操作简单，但即时编译器本身会引入较高的开销，且即使没有探针修改或是测量，也需要在后台持续运行.

2）基于快速断点的动态插桩技术

快速断点（fast breakpoint）[42]是一种经典的动态插桩方法. 它使用分支指令或跳转指令替换被测程序中的指令，使程序跳转执行用户注册的插桩代码片段，完成之后再跳回到原指令流中. 跳转前后为了保证状态的一致性，需要对执行状态进行缓存和还原. 这种插桩技术在每个测量探针处额外加入多次跳转和状态保护指令，其测量开销会随探针密度的增加快速增长. 现有基于快速断点的动态插桩技术[43−45]在探针密度较大时，会严重影响 NF 软件的性能.

本文对常用的库打桩、即时编译器和快速断点 3 类插桩技术进行了实验对比，其开销和适用范围如表 3 所示. 2 种动态插桩技术引入相对较高的测量开销，但可测量的函数范围更广. 静态库打桩由于不支持内部函数，在实际 NF 软件性能测量中，会丢失部

Table 3 Comparison of Popular Instrumentation Technologies

表 3 主流插桩技术对比

插桩技术	类型	外部函数	内部函数	单点开销/cycle
库打桩	静态	支持	不支持	30
即时编译器	动态	支持	支持	100
快速断点	动态	支持	支持	60

分关键信息. 综上所述, 现有的插桩技术无法同时满足高性能 NF 软件性能测量的开销和可测范围的需求.

2 低开销函数级动态插桩方法

通过对现有的插桩方法的研究和实验, 本文发现现有的插桩方法无法同时满足可测范围和高性能 NF 软件对测量开销的需求. 为了解决现有插桩方法在 NF 软件性能测量上的问题, 本文对常见的 NF 软件/软转发编程框架进行了调研. 目前的高灵活性和拓展性的 NF 软件框架大多采用模块化架构, 基本的简单的处理操作多直接实现在框架核心代码中, 即以内部函数的形式存在. 而复杂的数据包处理操作, 则根据其操作类型, 实现在不同的外部模块中, 并封装成动态库, 在实际使用时按需加载. 这些外部模块中的函数实际以外部库函数的形式存在.

针对这一发现, 本文创新性地提出了动态库打桩和函数级快速断点相结合的函数级动态插桩方法. 对不同类型的函数和函数调用方法采取不同的插桩方式, 在满足可测范围的同时, 尽可能降低开销. 即, 对于外部库函数, 提出了动态的库打桩方法; 对于内部函数, 对现有快速断点进行了针对性的优化, 提出了函数级的低开销快速断点.

2.1 动态库打桩

库打桩在 Linux 链接器中已经有稳定的实现和应用. 其原理是, 给定一个目标函数, 实现一个原型相同的包装(wrapper)函数, 通过链接时的符号重定向, 截获程序对目标函数的调用, 取而代之地执行自行实现的包装函数. 目前, Linux 编译器支持在 3 个阶段进行打桩:

1)编译时打桩. 需要直接修改目标程序源代码.

2)链接时打桩. 即静态连接器的“-wrap f”选项, 在链接时, 将对符号 *f* 的引用解析成对符号 *__wrap_f* 的引用, 需要重链接.

3)程序启动时打桩. 通过设置系统环境变量, 使程序在动态链接阶段优先搜索指定的带有包装函数的共享库, 从而替换对目标函数的调用.

以上 3 种实现, 均是在程序运行前的一次性修改, 无法在 NF 软件长期稳定的运行过程中实时按需进行打桩.

那么, 在程序稳定运行过程中, 函数是如何被调用的. 对于内部函数, 函数调用指令 CALLQ 指向的地址通常为目标函数的实际地址, 如果要进行重定向, 需要找到目标函数所有的调用点来修改 CALLQ 指令的目标地址. 在基于 C/C++的 NF 程序中, 函数指针被大量使用, 难以找到所有的调用点. 但是对于外部库函数, 其动态链接时的符号解析需要符号重定位(relocation)和延迟绑定(lazy binding), CALLQ 指令的目标地址实际指向的是一个 PLT(procedure linkage table)桩函数. 如图 2 所示, 程序调用外部库函数 *foo* 时, 实际 CALLQ 指令会指向函数 *foo@plt*, 这个函数只有 3 行指令. 在第 1 次被调用时, 第 1 行跳转指令对应的跳转, 在经过 GOT 表后, 实际指向该 PLT 桩函数的第 2 条指令. 第 2 条和第 3 条指令调用动态链接器, 完成外部库函数的符号解析. 解析后其第 1 条指令的跳转地址将被替换为 *foo* 的实际地址. 并且在之后的执行中, 后 2 条指令将不会被执行.

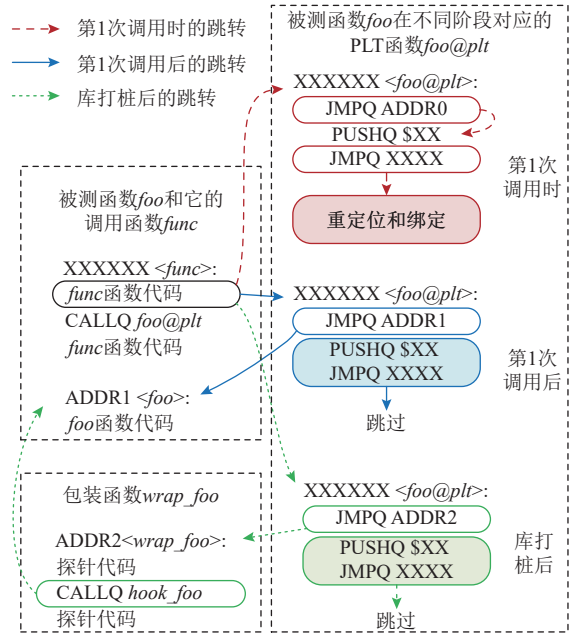


Fig. 2 Dynamic linking and library interposition of library function

图 2 库函数的动态链接和库打桩

本文将链接时库打桩的思路与外部函数的动态链接过程相结合, 提出了动态库打桩, 其核心思路是直接修改 PLT 桩函数和动态符号表来完成重定向. 以目标函数 *foo* 为例, 具体步骤为:

1)预先实现与目标函数 *foo* 原型相同的包装函数 *wrap_foo* 和用于保存原目标函数的空符号 *real_foo*, 将它们单独封装在共享库中.

2)在程序稳定运行过程中, 通过进程地址空间内的二进制代码解析, 获取目标函数 *foo* 和其对应的包装函数 *wrap_foo* 的实际载入地址, 用 *wrap_foo* 的实际载入地址替换掉目标函数 *foo* 对应的 PLT 桩函数的第 1 行跳转目标, 使所有对 *foo* 的调用都被重定

向到 *wrap_foo*.

3)为了保证包装函数内能正确执行目标函数，包装函数构造时对目标函数的调用实际为对空符号 *hook_foo* 的调用. 在完成第 2 步的替换后，将原目标函数的实际地址写入到 *hook_foo* 对应的 GOT(global offset table)表项中，即 *hook_foo* 实际变为原目标函数.

此外，因为被测程序被视作黑盒，所以无法直接获知所有目标函数的原型. 本文利用反编译技术，直接从二进制代码中解析目标函数的函数原型，并通过自动化程序生成带有测量探针的包装函数. 尽管难以准确获知每个参数的数据类型，但可以明确知道参数个数. 而 Linux 编译时，函数参数的处理仅取决于参数个数，与参数的类型无关，只需要知道目标函数的参数个数和是否拥有返回值，即可保证状态的一致性.

通过动态库打桩，可以在程序运行过程中，实时按需地对外部库函数进行打桩，且打桩后与静态打桩相比，没有添加额外的跳转或状态保存操作，理论上，其开销与静态库打桩相同.

2.2 函数级快速断点

对于不能通过动态库打桩进行插桩的内部函数，本文使用快速断点技术进行插桩. 传统快速断点是向目标程序的指定位置(本文称之为桩点)插入断点，执行额外的代码. 为了测量和计算目标函数每次执行完整的执行开销，对于每个目标函数，有 2 段额外代码需要在不同的位置被执行，即函数入口处的测量探针的执行和函数出口处探针的执行.

图 3 给出了使用传统快速断点对目标函数进行插桩的过程. 每一段额外的代码都需要封装在一个完整的断点中，2 个测量探针分别对应图中的 2 个断点. 断点 1 直接插入到目标函数的第 1 条指令处，而断点 2 则插入到目标函数的返回指令 RETQ 前. 下面以断点 1 为例，描述其运行原理.

1)为了使程序的执行转移到断点上，使用跳转指令 JMPQ 替换掉桩点处的 N 条指令. 这里 N 的数值取决于不同平台上跳转指令的长度.

2)被插入的额外代码段，即测量探针的执行，可能会破坏原执行过程中的全局状态，如堆栈状态和寄存器的值. 为了不破坏原程序的执行结果，全局状态的一致性必须被保证. 因此，在断点中，测量探针调用前后，需要执行全局状态的保护，即全局状态保存和还原. 同时，为了临时保存状态，还需要分配额外的状态空间，即状态空间管理.

3)为了保证目标函数的正确执行，需要执行被替换掉的 N 条指令. 然后通过跳转指令回到桩点处，继续执行目标函数后续的代码.

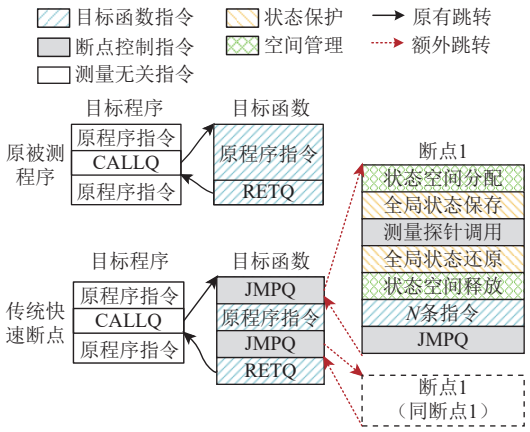


Fig. 3 The principle of traditional fast breakpoint

图 3 传统快速断点原理

如图 3 所示，对于一个目标函数，要完成入口和出口处的 2 次探针插入，总计加入 4 次额外的跳转、2 次状态空间管理、2 次全局状态保护.

通过观察发现，打桩的模式非常固定，即在每个被测函数的入口和出口处分别插入一个探针. 针对这一固定的模式，本文提出了函数级快速断点，核心思路是对原本的入口和出口处的 2 个快速断点进行合并. 对于每一个目标函数，只需要 1 个函数级断点. 与传统快速断点相同，将目标函数的前 N 条指令替换为跳转指令，使原程序的执行转移到断点上. 断点内不再只执行 1 个测量探针，而是执行全部的 2 个探针，如图 4 所示. 同样，为了保证状态一致性，断点首先分配状态空间，对全局状态进行保存，然后调用第 1 个测量探针. 和传统快速断点不同的是，第 1 个探针执行后，对状态进行还原，但不释放状态空间，并且不再只执行目标函数被替换掉的 N 条指令，而是除

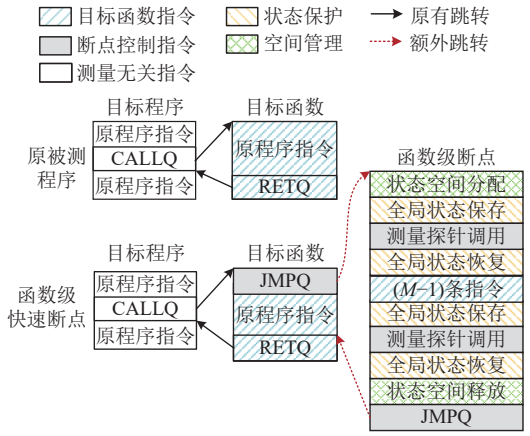


Fig. 4 The principle of function-level fast breakpoint

图 4 函数级快速断点原理

了 RETQ 以外的所有指令. 假设目标函数总计包含 M 条指令, 那么断点就需要执行 $(M-1)$ 条指令. 接下来, 第 2 个探针需要被调用, 由于之前分配的状态空间没有被释放, 这里直接进行复用. 在同样的状态保存、探针调用、状态还原过程后, 状态空间才被释放. 最后, 断点结束不再跳回到目标函数的入口处, 而是直接跳转到其出口处, 即 RETQ 指令前.

通过这种优化, 对于每个目标函数, 仅插入了 1 个断点, 总计需要 2 次额外的跳转、1 次状态空间管理和 2 次全局状态保护. 与传统快速断点相比, 节省了 2 次额外的跳转和 1 次状态空间管理.

3 基于动态插桩的数据包级 NF 软件性能测量方法

在低开销函数级动态插桩方法的基础上, 本文设计并实现了基于动态插桩的数据包级 NF 软件性能测量方法 LProfile, 其基本原理如图 5 所示.

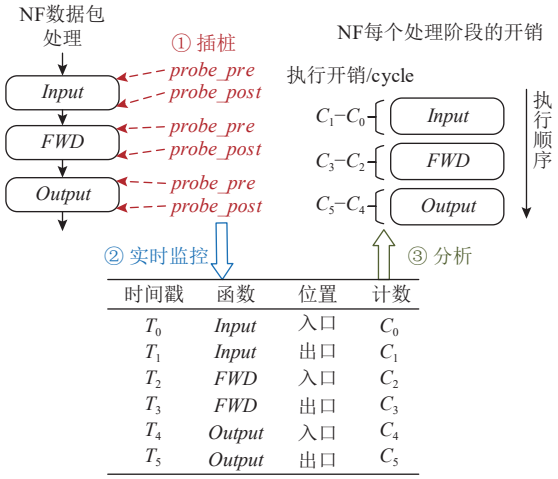


Fig. 5 Dynamic instrumentation based NF software performance measurement method

图 5 基于动态插桩的 NF 软件性能测量方法

在每个目标被测函数的入口和出口处, 分别插入一个测量探针, 测量探针的执行和 NF 实际数据包处理紧耦合. 探针触发时, 同时收集 PMC 性能数据和触发点的执行时序信息, 并通过后期离线数据处理和分析, 计算 NF 每个数据包处理过程中不同阶段的处理开销.

如第 1 节所述, 现有的基于动态插桩的方法会引入大量的测量开销, 可能严重破坏 NF 软件本身的性能行为.尽可能地降低测量开销是本文的主要目标之一. 通过对现有基于插桩的测量工作的深入分析, 本文将测量开销分解为插桩开销、单一探针执行开销和测量探针部署密度这 3 个部分, 其中探针部署问题本文并不涉及.为了解决现有方法的高开销和高干扰问题, LProfile 对插桩方法和轻量化探针 2 个部分进行了设计与实现的优化, 其中插桩方法的设计已在第 2 节中进行了描述.

3.1 整体架构

LProfile 的整体架构如图 6 所示. LProfile 的基础是函数级动态插桩库和轻量级测量探针库. 其中函数级动态插桩库由动态库打桩和函数级快速断点 2 部分组成, 均在开源动态插桩库 Dyninst[44] 的基础上实现. 轻量级测量探针库主要由 PMC API 和数据记录存储机制组成, 3.2 节将详细描述其设计和实现. 轻量级探针库中的 PMC 数据读取和数据记录的写入共同构造出测量探针, 其余部分和函数级动态插桩库一起, 组成 LProfile 监控进程. 实际测量的流程如下:

1)LProfile 监控进程根据用户指定的 NF 软件实例和被测函数确认对应的 NF 进程, 解析进程对应的进程地址空间, 定位被测函数. 根据被测函数的类型, 执行动态库打桩或函数级快速断点, 将测量探针插入到被测函数中.

2)LProfile 监控进程通过 PMC API, 开启对 NF 进程的 PMC 测量.

3)测量探针作为 NF 进程的一部分, 随 NF 进程的执行进行性能数据的收集, 并将数据记录写入到共享内存中. LProfile 监控进程在 PMC 测量开启后, 将不断轮询共享内存, 从中取出数据记录, 并写入文件.

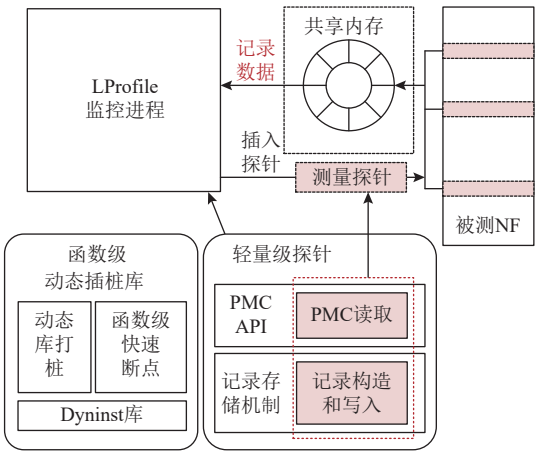


Fig. 6 Overall architecture of LProfile

图 6 LProfile 整体架构

3.2 探针轻量化

测量探针主要有 2 个需要执行的核心操作: 一是数据记录所需信息的获取, 包括 PMC 硬件性能数据的读取以及当前时序和调用信息的读取; 二是数据

记录的存储. LProfile 分别从这 2 方面对探针进行轻量化.

1)数据记录所需信息的获取

在 PMC 数据读取方面，现有方法大多直接使用 Linux 自带的系统调用级 PMC API. 由于系统调用的每次执行都涉及中断处理，其单次开销极大，并不适用于高性能 NF 软件程序. PAPI[46] 对用户态指令的使用进行了封装，提供完整的 PMC API，但由于实现过于繁复，单次 PMC 读取开销约 150 cycle. 通过调研发现，Linux 本身提供了用户态指令 rdpmc 直接读取 PMC 寄存器，该指令可以在 20 cycle 左右完成 1 次读取，但它缺乏配套的 PMC 管理 API. LProfile 直接使用用户态指令进行 PMC 数据读取，并实现了配套的管理 API.

在时序信息方面，LProfile 直接使用 PMC 中的 CPU 时钟周期数作为时间戳，它本身也是性能测量的重要指标. 在调用信息上，同样使用最精简的记录方法. 在第 1 次执行测量时，LProfile 对整个被测程序进行 1 次完整的扫描，统计所有可测量的函数，并给每个函数赋予独立的函数 ID. 对于外部函数，需要为每个目标函数构造一个包装函数，包装函数内仅包含目标函数前的测量探针函数、目标函数和目标函数后的测量探针函数 3 个函数调用. 目标函数 ID 和相对位置(前或后)会作为参数传入测量探针中，保证探针执行时能准确获取位置信息. 对于内部函数，在构造断点时，同样也可以明确获知每个探针所对应的目标函数和相对位置. 调用栈信息则可以通过执行时序后期离线进行还原，如图 5 所示.

2)高性能数据记录存储

探针读取到性能数据后，需要实时地将数据写入到文件中，以进行后续的离线分析. 如果在每次探针执行时都进行文件读写操作，对于高性能 NF 程序来说，其开销是不可接受的. 利用本地缓存和共享内存相结合的方式，LProfile 实现了高性能的性能数据记录存储. 监控进程在执行完插桩后，会额外运行一个独立于被测进程的数据存储进程，并与被测进程共享一块由无锁环形队列管理的内存. 被测进程端，探针负责按需将数据记录放入环形队列，数据存储进程则周期性地从环形队列中取出数据记录并写入到文件，整个流程均是无锁的. 为了进一步进行优化，LProfile 还设计实现了本地缓存机制，即在被测端，探针不需要每构造一个记录就写一次共享内存，而是将记录先放入本地缓存中，只有当本地缓存已满时，才将数据写入共享队列. 为了尽量避免多线程同步问题，本地缓存使用线程本地存储(thread-local storage)实现，每个线程有自己的状态、函数执行计数和独立的环形队列.

通过对数据读取和存储 2 个方面的优化，LProfile 将单一探针的测量开销大幅降低，与低开销函数级动态插桩方法相结合，将现有方法单函数上千 cycle 的测量开销降低至 100 cycle 左右，极大地减轻了测量行为对高性能 NF 软件自身行为的干扰.

4 LProfile 对 NF 软件性能的干扰程度分析

正如 1.3 节所述，性能测量对 NF 软件本身的干扰是无法避免的. 测量所带来的干扰使得测量所得的数据与真实性能数据存在偏差. 使用这些数据进行分析和诊断，会影响结果的准确性. 针对数据包级的性能测量需求，进一步探讨 LProfile 对 NF 数据包级性能行为的干扰程度.

LProfile 的测量整体依托于 PMC 的计数模式，如图 7 所示. NF 进程运行过程中所有硬件性能指标的测量和基本计数工作，均由 CPU 中的 PMC 硬件完成. LProfile 负责按照用户的需求配置和启动 PMC，构造测量探针并插入到 NF 进程中，在探针中读取 PMC 寄存器计数. 类似基于 PMC 的其他性能测量方法(如 Perf)，LProfile 可测量的性能指标包括被执行的指令数、运行所需的 CPU 时钟周期数、各级 cache 的 miss/hit、TLB 的 miss/hit 等 PMC 指标.

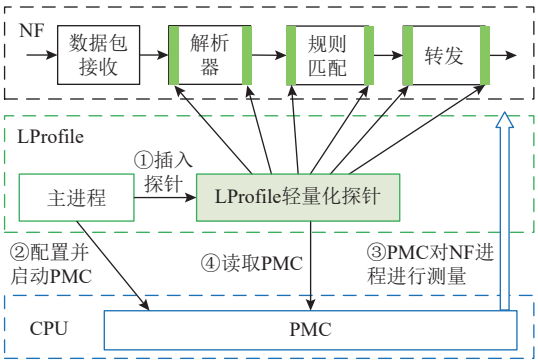


Fig. 7 The principle of LProfile

图 7 LProfile 原理

由于运行在计数模式下的 PMC 本身的测量计数开销是可以忽略不计的，对于被测 NF 软件而言，其性能所受的干扰主要来源于 LProfile 插入的轻量化探针.

以图 7 为例，LProfile 对 NF 软件的 3 个模块进行了监控和测量，即在这 3 个模块的前后分别插入了 1

个探针, 共计 6 个探针, 每个探针除了对性能数据的读取和记录外, 还包括为了插入探针而引入的插桩代码. 在每一次完整的数据包处理过程中, 假设这 3 个模块都一定且仅会被执行 1 次, 那么 6 个探针也会随着被测模块被分别执行 1 次. 在理想状态下, 探针本身开销固定, 且与 NF 进程之间不存在因资源竞争(如 cache 竞争等)导致的干扰. 那么对于每次数据包处理, LProfile 所引入的额外开销都是增量的, 仅是绝对数值上的增加, 不会破坏不同数据包处理之间或是不同函数之间的性能指标差异, 即不会干扰 NF 软件数据包级的性能行为特征.

实际场景中, 探针的执行必然会和 NF 软件的执行产生资源竞争, 从而互相干扰. 为了尽可能降低这种干扰, LProfile 设计实现了更高效的插桩和轻量化的探针, 尽可能减少探针所需要执行的操作, 尽可能保证探针本身每次执行的波动较小.

本文以防火墙为例给出了实验验证, 观察其在测量前后, 数据包级性能(逐包时延)的变化情况, 即测量行为对 NF 软件数据包级性能特征的干扰程度. 具体实验环境将在 5.1 节进行介绍, 此实验中 LProfile 测量了 5 个防火墙核心函数, 图 8 给出了实验结果. 如图 8 所示, 尽管测量前后, 平均时延从 15.7 μs 增长到了 30.5 μs, 但逐包时延的整体特征以及不同数据包之间的差异, 都能相对完整地记录下来. LProfile 实现了相对增量级的测量开销, 对 NF 软件数据包性能行为特征的干扰较小.

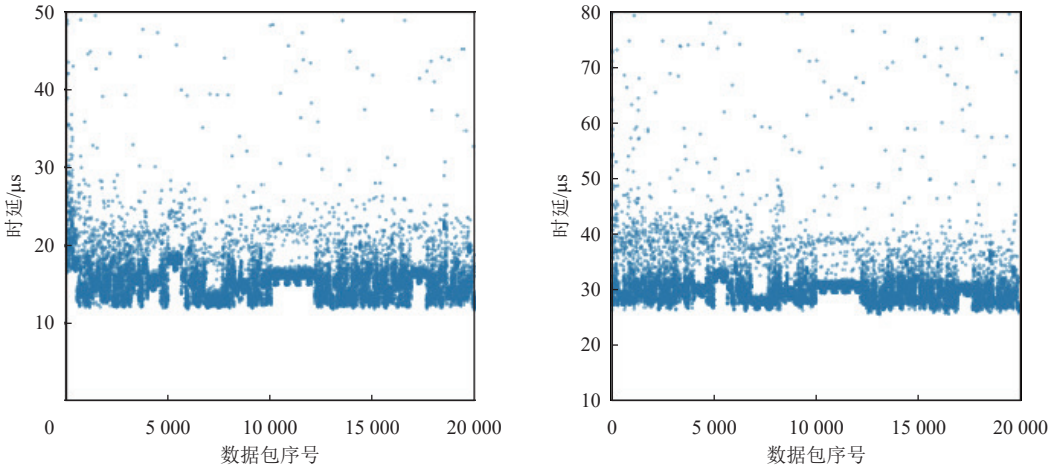


Fig. 8 Firewall packet delay before and after LProfile measurement

图 8 LProfile 测量前后防火墙逐包时延

使用 LProfile 测量所得数据进行 NF 软件性能诊断分析时, 能够更加真实地还原 NF 软件原本的性能行为特征, 使得性能问题定位更加高效.

5 实验验证

针对基于插桩的高性能 NF 软件性能测量场景, 本节分别从测量的开销分析和实际场景案例 2 个方面进行了实验验证.

5.1 实验环境

为了评测 NF 软件实际运行时性能测量的影响和作用, 本文在通用高性能服务器上搭建了真实的单点转发环境, 网络拓扑如图 9 所示. Linux 服务器的 CPU 为 Intel Xeon® Platinum 8160 2.10 GHz, 内存为 128 GB, 并使用 Mellanox Technologies MT27800 Family 高性能网卡. 2 台服务器中, 1 台运行发包器 PktGen, 1 台运行 NF 软件转发程序(VPP), 通过 2 对物理 NIC 端口直连.

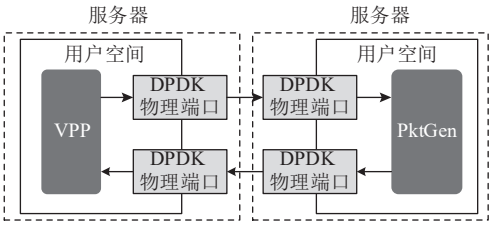


Fig. 9 Experimental topology

图 9 实验拓扑

这里所用的发包器基于 DPDK PktGen 开发, 它本身可以以链路速度进行收发包测试, 测量 NF 的实时吞吐和平均时延, 但无法记录所有单包的时延结果. 为了更直观地观察性能测量对 NF 数据包处理的

影响，本文对其进行了修改，加入了单包时延的测量和统计. 实验所用的 NF 软件全部基于 Cisco VPP 高性能软件转发架构，并选取了其中的 3 个 NF、防火墙、NAT 和 IPsec 安全网关来评估测量的开销. 其中，进一步选取了防火墙作为详细开销分析的对象.

常见的基于插桩的性能测量工具中，Score-P 需要修改编译链接选项来完成插桩. 由于 VPP 所使用的复杂的 CMAKE 结构，暂时无法使用 Score-P 进行性能测量. 因此，在对 NF 整体的测量开销分析中，仅使用了基于动态插桩的 TAU 进行对比.

5.2 开销分析

对基于插桩的性能测量方法的开销进行细粒度的分析和实验验证. 正如第 3 节所示，其开销主要来源于插桩引入的开销和探针本身的运行开销 2 个部分. 下面分别对这 2 个部分进行讨论，并最终对整体开销进行实验评估.

5.2.1 插桩开销

现有的插桩方法很多难以在 VPP 中进行应用，为了更全面地比较插桩方法的开销，这里使用了一个拥有 1 000 个函数的简单测试程序进行实验. 本文分别用 Linux LD wrap 静态库打桩、Intel Pin(即时编译器)、Dyninst(快速断点)和本文提出的函数级动态插桩方法 LProfile 对被测程序进行插桩，其中 Intel Pin 和 Dyninst 需要手动在函数入口和出口处分别打桩. 本次实验只关心插桩本身带来的性能影响，插入的代码段为一个无任何操作的空函数.

对于不同大小的被测函数，其单次执行的测量开销如图 10 所示. 可以清晰地看出，Intel Pin 的开销远大于其他方法，且会受被测函数影响，这是因为 Intel Pin 本身的效率与被测函数大小强相关. 而 Linux LD wrap 和 Dyninst 对于单一桩点所插入的额外操作都是定量的，与被测函数本身无关，Linux LD wrap, Dyninst, LProfile 的插桩开销基本不受被测函数大小的影响.

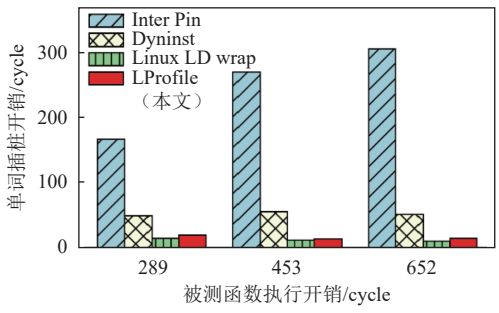


Fig. 10 Single instrumentation overhead of measured functions with different sizes

图 10 不同大小的被测函数的单次插桩开销

总体对比而言，同样是插入定量的额外操作的 Linux LD wrap, Dyninst, LProfile, 由于 Dyninst 要引入更多跳转和状态空间管理操作，其开销在 50 cycle 左右. 而 Linux LD wrap 作为静态插桩技术，每次使用都需要对 NF 软件程序进行重编译和重链接，使用复杂. LProfile 使用动态插桩技术，不需要对 NF 软件程序进行任何修改和操作，更加简单易用. 而与 Dyninst 相比，LProfile 将插桩开销平均降低了 70%.

5.2.2 探针开销

本文对 PMC 性能数据的读取方法进行了开销评测，其结果如表 4 所示. 目前大部分工具所使用的 Linux Perf event API, 因为其本身是系统调用级的，其开销接近 1 000 cycle. 基于用户态指令开发的 PAPI, 因其功能更加复杂全面，如 PMC 的多路复用、跨平台的数据修正和校对功能，其开销大于直接使用用户态指令 rdpmc 的开销.

Table 4 Overhead of Reading PMC Performance Data

表 4 PMC 性能数据读取开销

PMC API	单次读取开销/cycle
用户态指令	19
PAPI 快速读取	150
Linux Perf event API	720

本文所设计的轻量化探针直接使用用户态指令，尽可能地精简探针内操作和性能记录内容，使其开销大幅降低. 通过手动将测量探针写入到防火墙的添加规则(Insert)、查找匹配(Search)和删除规则(Delete)这 3 个操作中，验证了 LProfile 轻量级测量探针的开销. 如表 5 所示，通过精简用户态指令和探针操作，LProfile 将探针大小压缩到了 350 cycle 左右. 更进一步，通过对性能记录存储机制的优化，探针被压缩到了 144 cycle 左右，与 PAPI 单一的 PMC 性能数据读取操作的开销近似.

Table 5 Overhead of Single Probe of LProfile

表 5 LProfile 单探针开销 cycle

被测函数	原函数开销	探针开销（无存储优化）	探针开销（存储优化）
Insert	1 539	367	149
Search	42	328	138
Delete	2 485	365	145

5.2.3 整体测量开销

在真实转发场景下，本文分别使用 TAU 和 LProfile

对 NF 软件进行性能测量, 评估测量行为引入的额外开销. 这里, 为了避免 TAU 直接使用 Linux Perf event API 而引入大量系统调用, 对其开启了 PAPI 优化, 即让 TAU 只用 PAPI 库完成 PMC 性能数据的测量采集.

如图 11 所示, 对于单函数单次测量, LProfile 的单次开销(平均 195 cycle)远小于 TAU 的单次开销(平均 1 086 cycle). LProfile 的开销大小符合上文对开销组成的各个部分分别测量的开销结果, 而 TAU 即使在 PAPI 的优化下, 其整体开销也在 1 000 cycle 以上. 与 TAU 相比, 通过探针轻量化和低开销函数级动态插桩等优化, LProfile 将单点测量开销降低了 82%.

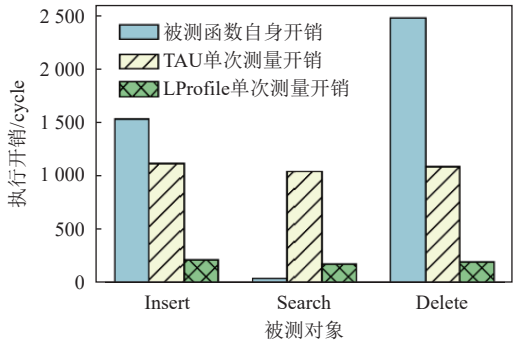


Fig. 11 Single measurement overhead of single function

图 11 单函数单次测量开销

导致测量开销差异的原因, 除了插桩方法和 PMC 的使用差异, 还有探针操作的优化问题:

1)TAU 的时间戳获取直接使用 Linux GCC 库的系统时间函数, 而 LProfile 使用 PMC 中的 CPU 时钟周期数硬件事件;

2)TAU 中使用复杂的栈结构记录和还原被测函数的调用关系, 而 LProfile 直接利用数据记录生成时的时序和函数信息, 通过离线分析还原调用关系, 不影响实时测量;

3)TAU 没有对数据记录的存储进行优化, 而 LProfile 引入了共享内存和本地缓存相结合的存储优化机制.

图 12 展示了随着探针密度的增加, 测量对防火墙(FW)、NAT 和 IPSec 整体性能的影响. 这里的探针密度是指平均每个数据包的处理过程中, 测量探针执行的次数. 在实际 NF 数据包处理过程中, 为了提高运算效率, 大多采用批处理进行性能优化. 一次数据包处理函数的执行, 如防火墙中的规则匹配, 可能会处理多个数据包, 一次探针的执行可能对应多个数据包. 从图 12 可以清晰看出, LProfile 对 3 个 NF 软件吞吐量的影响明显小于 TAU, LProfile 影响最大的 IPSec 在探针密度为 0.082 时其性能下降了 28.0%, 影响最小的 NAT 在探针密度为 0.069 时性能仅下降了 3.4%. 此外, TAU 在加入较多探针时会导致 NF 软件程序崩溃, LProfile 则没有这个问题.

5.3 案 例

本节从实际的场景出发, 验证 LProfile 如何应用于 NF 软件性能测量.

5.3.1 NF 软件性能问题的传递性

文献 [6] 表明 NF 软件中队列的存在使得 NF 软件的性能问题具有传递性, 如本文 1.1 节所述, 性能问题的传递使得问题的定位更加困难. LProfile 数据包级的性能测量方法, 能够直观快速地定位这一类问题. 本文以 VPP 的防火墙为例, 进行了性能测量和分析.

防火墙共 1 000 条五元组规则, 在低负载条件下总计处理了 100 000 个数据包. 该防火墙程序的一次完整数据包处理会从上到下依次经过表 6 所列出的 10 个模块, LProfile 对这些模块进行统一测量, 性能指标为执行开销. VPP 架构中, 模块间没有队列, 从网卡收包队列中取出数据包后, 以运行到完成(run-to-completion)的方式一次将取出的数据包处理完成, 因此, 各模块的执行次数相同, 且批处理数都小于总数据包数.

从表 6 中可以看出, 开销最大的模块为 ACL 模块, 即防火墙的五元组规则匹配. 但在当前场景下, ACL 模块在总体中的占比并不大, 负责发送数据包

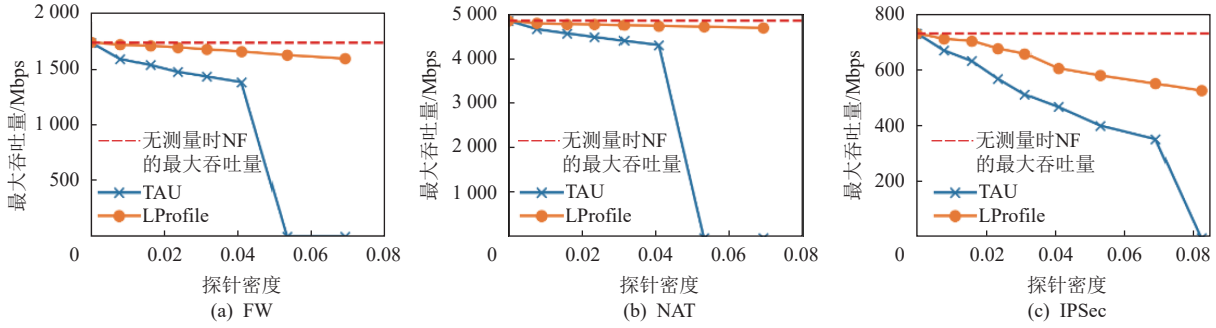


Fig. 12 Impact of the measurement on NF performance

图 12 测量对 NF 性能的影响

Table 6 Number of Executions and Average Execution Overhead of the Firewall Core Module

表 6 防火墙核心模块的执行次数和平均执行开销

模块	实际函数	执行次数	平均开销/μs
Ethernet 输入	*ethernet_input_node_fn*	67 124	0.70
L2 输入	*l2input_node_fn*	67 124	0.69
L2 功能入口	*l2_in_feat_arc_node_fn*	67 124	0.70
ACL	*acl_in_l2_ip4_node_fn*	67 124	1.33
L2 功能出口	*l2_in_feat_arc_end_node_fn*	67 124	0.63
L2 路由学习	*l2learn_node_fn*	67 124	0.65
L2 转发	*l2fwd_node_fn*	67 124	0.76
L2 输出	*l2output_node_fn*	67 124	0.77
VNet 输出	*vnet_interface_output_node*	67 124	0.84
DPDK 输出	*dpdk_dev_output*	67 124	0.92

的 VNet 输出模块和 DPDK 输出模块相对占比较大.

通过对发包器记录的数据包收发时间和 LProfile 记录的防火墙每次数据包处理的时间进行匹配，可将数据包和 LProfile 记录的 67 124 次数据包处理完全对应. 截取其中 15 次完整数据包处理，对应 19 个数据包，其测量结果如图 13 所示. 图中上半部分为 LProfile 测量所得的各模块开销，每个矩形色块表示一次函数执行，矩形的宽度等于这次执行的运行时间，可以清晰地看出每次处理各模块的时延占比. 下半部分为发包器测得的数据包收发时间，每条线段代表一个数据包，线段颜色的深浅对应时延的高低，颜色越深，时延越高. 将上下 2 部分一一对应，可以进一步展示出时延问题的复杂性.

图 13 中大部分的高时延，是由排队和批处理导致的. 以图中第 5 个数据包为例，它与后续到达的第 6 个数据包恰好被合并在同一批次处理，ACL 模块的执行因需要处理的数据包增多而明显消耗了更多时间，再与排队时间累加，造成了第 5 个数据包的高时延.

此外，由于队列的存在，时延问题存在传递性，以图中时延最高的第 14 个数据包为例. 第 13 个数据包在处理过程中，VNet 输出模块耗时异常，使其自身时延高于平均值. 而这个数据包的长时间处理，使得后续数据包，即第 14 个数据包无法被处理，排队时间大幅增加. 在等待过程中，后续数据包（即第 15 个数据包）还在持续到达，导致第 14 个后续包和第 15 个后续包被并入同一批次. 同样地，ACL 模块的执行时间增加. 以上所有耗时的增加累积在一起，导致第 14 个后续包时延严重异常. 但是，从上述分析中可以看出，这个后续包的时延问题的主因并不是它自己的处理过程，而是前序数据包异常导致的排队等待. 这种具有传递性的性能问题，现有的性能测量方法所得到的数据均无法准确分析，但 LProfile 可以帮助定位.

5.3.2 空间局部性对 NF 软件性能的影响

对于低时延高吞吐的高性能 NF 软件，输入流量的空间局部性会影响 CPU cache 的利用率，进而影响其性能. 文献 [47] 对该问题进行了深入探讨. 本节使用 DPDK 中基于五元组规则的路由转发程序（L3FWD）来验证空间局部性对 NF 软件性能的影响.

这里引用 NSDI22 中描述空间局部性程度的指标，即空间局部性指数（spatial locality factor, SLF）. SLF 是指在同一个流中背靠背到达的数据包数. 假设输入流量中共有 3 个流 *A*, *B*, *C*. 当 SLF 的值为 1 时，数据包以“*ABCABC*…”的模式到达 NF，当 SLF 的值为 2 时，数据包以“*AABBCC*…”的模式到达. 对于同一原始流量，SLF 的值仅影响流量内数据包的空间局部性，实际包含的数据包不变. SLF 的值越大，空间局部性越强.

实验选用一个 3 000 规模的五元组规则集和一组真实的输入流量. 为了控制实验变量，本文对原始流量以流为基准进行了不同空间局部性的重排序. LProfile 观察不同空间局部性程度下，L3FWD 每次数据包处理期间的指令数、L1d load（L1 数据缓存载入

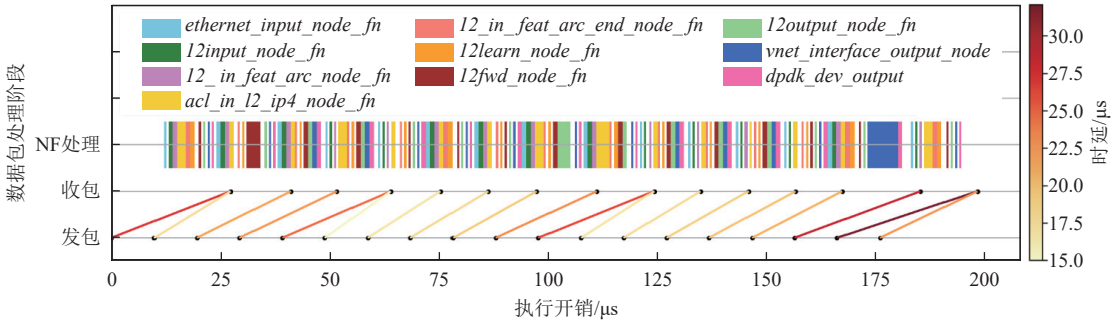


Fig. 13 Fine-grained overhead of each data packet processing of the firewall and its corresponding data packet delay

图 13 防火墙每次数据包处理的细粒度开销和其对应的数据包时延

次数)、L1d load miss(L1d 数据缓存载入未命中次数)、LLC miss(最后一级缓存未命中次数)的变化,其结果如图 14 所示. 为了更直观地观察和对比变化趋势,本文对所有指标进行了归一化处理.

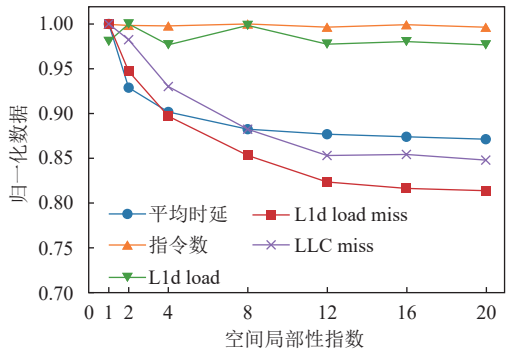


Fig. 14 Impact of spatial locality on L3FWD performace

图 14 空间局部性对 L3FWD 性能的影响

随着 SLF 值的增大,L3FWD 数据包处理的平均时延在减小,即处理性能逐渐提高. 每次处理所执行的指令数和 L1d load 次数基本稳定,而 L1d load miss 次数和 LLC miss 与平均时延相同,呈下降趋势. 实验中不同 SLF 值的输入流量都是对同一原始流量的重排序,即总体处理的数据包是相同的,整体上 L3FWD 所执行的操作也是相同的,结果中平稳的指令数和 L1d load 数印证了这一点. 在指令数和 L1d load 数稳定的情况下,L1d load miss 和 LLC miss 的下降表明了随着空间局部性的增大,cache 的利用率在提高,进而提高了 L3FWD 的性能,此结论与文献 [47] 的结论一致.

上述实验结果说明,通过对 NF 软件底层硬件性能指标,如 cache 使用情况等进行细粒度测量,LProfile 的性能测量可以分析影响性能的关键指标,更有效地应用于 NF 软件性能问题定位和分析.

6 结 论

本文从高性能网络功能软件的性能行为和性能问题出发,全面分析了 NF 软件性能测量的需求与挑战. 深度调研了现有的性能测量方法,从原理和实验验证 2 个角度,论证了基于插桩的性能测量方法更适用于 NF 软件性能测量. 从插桩优化和探针轻量化 2 个角度,设计并实现了基于动态插桩的数据包级 NF 软件性能测量方法 LProfile. 在插桩优化上,结合库打桩技术和快速断点技术,提出了低开销函数级动态插桩方法,与传统快速断点相比,将插桩开销降低了 70%. 在探针轻量化方面,通过用户态指令的使用,数据记录的精简和高效数据存储优化,实现了轻量级的测量探针. 与 TAU 相比,LProfile 在单点上测量开销降低了 82%. 本文在真实转发场景下,对 LProfile 进行了验证,并通过案例,使用 LProfile 分析 NF 软件时延问题的传递性和空间局部性对 NF 软件性能的影响.

受插桩机制所限,LProfile 无法对内联函数进行测量,而高性能 NF 程序中通常存在大量内联函数. 后续将研究解决内联函数的测量、测量探针部署、性能数据可视化等问题.

作者贡献声明: 贾茹提出了思路并完成了设计实现和论文撰写;姜海洋给出了设计和实验意见并修改了论文;李振宇和谢高岗提出了指导意见并修改论文.

参 考 文 献

[1] Gibb G, Zeng Hongyi, McKeown N. Outsourcing network functionality [C] //Proc of the 1st Workshop on Hot Topics in Software Defined Networks. New York: ACM, 2012: 73−78

[2] Baloni D, Bhatt C, Kumar S, et al. The evolution of virtualization and cloud computing in the modern computer era [C] //Proc of the 2023 Int Conf on Communication, Security and Artificial Intelligence (ICCSAI). Piscataway, NJ: IEEE, 2023: 625−630

[3] Zhou Weilin, Yang Yuan, Xu Mingwei. Network function virtualization technology research[J]. Journal of Computer Research and Development, 2018, 55(4): 675−688 (in Chinese)
(周伟林, 杨芫, 徐明伟. 网络功能虚拟化技术研究综述[J]. 计算机研究与发展, 2018, 55(4): 675−688)

[4] Mijumbi R, Serrat J, Gorricho J L, et al. Network function virtualization: State-of-the-art and research challenges[J]. IEEE Communications Surveys & Tutorials, 2016, 18(1): 236−262

[5] Kaffes K, Chong T, Humphries J T, et al. Shinjuku: Preemptive scheduling for μsecond-scale tail latency [C] //Proc of the 16th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2019: 345−360

[6] Gong Junzhi, Li Yuliang, Anwer B, et al. Microscope: Queue-based performance diagnosis for network functions [C] //Proc of the ACM SIGCOMM 2020 Conf. New York: ACM, 2020: 390−403

[7] Lei Yiran, Yu Liangcheng, Liu V, et al. PrintQueue: Performance diagnosis via queue measurement in the data plane [C] //Proc of the ACM SIGCOMM 2022 Conf. New York: ACM, 2022: 516−529

[8] Chen Xiaoqi, Feibish S L, Koral Y, et al. Fine-grained queue measurement in the data plane [C] //Proc of the 15th Int Conf on Emerging Networking Experiments and Technologies. New York: ACM, 2019: 15−29

[9] Sonchack J, Michel O, Aviv A J, et al. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow [C] //Proc of the 2018 USENIX Conf on USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2018: 823−835

[10] Pedrosa L, Iyer R, Zaostrovnykh A, et al. Automated synthesis of adversarial workloads for network functions [C] //Proc of the ACM SIGCOMM 2018 Conf. New York: ACM, 2018: 372–385

[11] Iyer R, Pedrosa L, Zaostrovnykh A, et al. Performance contracts for software network functions [C] // Proc of the 16th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2019: 517−530

[12] Zaparanuks D, Jovic M, Hauswirth M. Accuracy of performance counter measurements [C] //Proc of 2009 IEEE Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2009: 23−32

[13] Weaver V M. Self-monitoring overhead of the Linux perf_event performance counter interface [C] //Proc of 2015 IEEE Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2015: 102−111

[14] Intel Corporation. Intel VTune Profiler user guide [EB/OL]. (2022-02-06) [2024-05-22]. https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html

[15] Mey D, Biersdorf S, Bischof C, et al. Score-P: A unified performance measurement system for petascale applications [C] //Proc of an Int Conf on Competence in High Performance Computing 2021. Berlin: Springer, 2011: 85−97

[16] Shende S S, Malony A D. The TAU parallel performance system [J]. International Journal of High Performance Computing Applications, 2006, 20(2): 287−311

[17] Adel B, Martin P, Mike B, et al. Performance analysis of DPDK-based applications through tracing [J]. Journal of Parallel and Distributed Computing, 2023, 173(C): 1−19

[18] Cisco. Vector packet processing (VPP) [EB/OL]. (2022-07-12) [2024-05-22]. https://wiki.fd.io/view/VPP

[19] Wu Wenfei, He Keqiang, Akella A. PerfSight: Performance diagnosis for software dataplanes [C] //Proc of the 2015 Internet Measurement Conf. New York: ACM, 2015: 409−421

[20] Daly J, Bruschi V, Linguaglossa L, et al. TupleMerge: Fast software packet processing for online packet classification [J]. IEEE/ACM Transactions on Networking, 2019, 27(4): 1417−1431

[21] Zhao Licheng, Shen Wenhai, Xiao Huadong, et al. The application of high performance computing technology in meteorological field [J]. Journal of Applied Meteorological Science, 2016, 27(5): 550−558 (in Chinese)
(赵立成, 沈文海, 肖华东, 等. 高性能计算技术在气象领域的应用 [J]. 应用气象学报, 2016, 27(5): 550−558)

[22] Li Zhenhua, Wang Hongyi, Li Yang, et al. Cloud native reinforced design for large-scale complex terminal networks [J]. Journal of Computer Research and Development, 2024, 61(1): 2−19 (in Chinese)
(李振华, 王泓懿, 李洋, 等. 大规模复杂终端网络的云原生强化设计 [J]. 计算机研究与发展, 2024, 61(1): 2−19)

[23] Ali R, Zikria Y B, Bashir A K, et al. URLLC for 5G and beyond: Requirements, enabling incumbent technologies and network intelligence [J]. IEEE Access, 2021, 9: 67064−67095

[24] Khalid J, Gember-Jacobson A, Michael R, et al. Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr [C] //Proc of the 13th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2016: 239−253

[25] Dobrescu M, Argyraki K. Software dataplane verification [J]. Communications of the ACM, 2015, 58(11): 113−121

[26] Stoenescu R, Popovici M, Negreanu L, et al. SymNet: Scalable symbolic execution for modern networks [C] //Proc of the 2016 ACM SIGCOMM Conf. New York: ACM, 2016: 314−327

[27] Zaostrovnykh A, Pirelli S, Pedrosa L, et al. A formally verified NAT [C] //Proc of the 2017 ACM SIGCOMM Conf. New York: ACM, 2017: 141−154

[28] Naik P, Shaw D K, Vutukuru M. NFVPerf: Online performance monitoring and bottleneck detection for NFV [C] //Proc of 2016 IEEE Conf on Network Function Virtualization and Software Defined Networks. Piscataway, NJ: IEEE, 2016: 154−160

[29] Pfitscher R J, Jacobs A S, Scheid E J, et al. A model for quantifying performance degradation in virtual network function service chains [C/OL] //Proc of 2018 IEEE/IFIP Network Operations and Management Symp. Piscataway, NJ: IEEE, 2018 [2024-05-22]. https://ieeexplore.ieee.org/document/8406268

[30] Adhianto L, Banerjee S, Fagan M, et al. HPCToolkit: Tools for performance analysis of optimized parallel programs [J]. Concurrency and Computation: Practice and Experience, 2009, 22(6): 685−701

[31] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation [J]. ACM SIGPLAN Notices, 2007, 42(6): 89−100

[32] Bruening D, Zhao Qin, Amarasinghe S. Transparent dynamic instrumentation [C] //Proc of the 8th ACM SIGPLAN/SIGOPS Conf on Virtual Execution Environments. New York: ACM, 2012: 133−144

[33] Zhao Qidong, Liu Xu, Chabbi M. DrCCTProf: A fine-grained call path profiler for ARM-based clusters [C/OL] //Proc of the 2020 Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2020 [2024-05-22]. https://doi.org/10.1109/SC41405.2020.00034

[34] Lehr J-P, Huck A, Bischof C. PIRA: Performance instrumentation refinement automation [C/OL] // Proc of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems. New York: ACM, 2018 [2024-05-22]. https://dl.acm.org/doi/10.1145/3281070.3281071

[35] Ates E, Sturmann L, Toslali M, et al. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications [C] //Proc of the ACM Symp on Cloud Computing. New York: ACM, 2019: 165−170

[36] Mace J, Fonseca R. Universal context propagation for distributed system instrumentation [C/OL] //Proc of the 13th EuroSys Conf. New York: ACM, 2018 [2024-05-22]. https://dl.acm.org/doi/abs/10.1145/3190508.3190526

[37] Su Pengfei, Jiao Shuyin, Chabbi M, et al. Pinpointing performance inefficiencies via lightweight variance profiling [C/OL] //Proc of the 2019 Int Conf for High Performance Computing, Networking, Storage and Analysis. Berkeley, CA: USENIX Association, 2019 [2024-05-22]. https://dl.acm.org/doi/10.1145/3295500.3356167

[38] Jia Ru, Pan Heng, Jiang Haiyang, et al. Towards diagnosing accurately the performance bottleneck of software-based network function

implementation [C] //Proc of 2023 Passive and Active Measurement. Berlin: Springer, 2023: 227−253

[39] Geimer M, Wolf F, Wylie B J N, et al. The Scalasca performance toolset architecture[J]. Concurrency and Computation: Practice and Experience, 2010, 22(6): 702−719

[40] Luk C, Cohn R, Muth R, et al. Pin: Building customized program analysis tools with dynamic instrumentation[J]. ACM SIGPLAN Notices, 2005, 40(6): 190−200

[41] Enrique S S, Gorka G M. Detecting and bypassing Frida dynamic function call tracing: Exploitation and mitigation[J]. Journal of Computer Virology and Hacking Techniques, 2023, 19: 503−513

[42] Kessler P B. Fast breakpoints: Design and implementation[J]. ACM SIGPLAN Notices, 1990, 25(6): 78−84

[43] Bruening D L. Efficient, transparent, and comprehensive runtime code manipulation [D]. Cambridge, MA: Massachusetts Institute of Technology, 2004

[44] Buck B, Hollingsworth J K. An API for runtime code patching[J]. The International Journal of High Performance Computing Applications, 2000, 14(4): 317−329

[45] Arras P A, Andronidis A, Pina L, et al. SaBRe: Load-time selective binary rewriting[J]. International Journal on Software Tools for Technology Transfer, 2022, 24(2): 205−223

[46] Browne S, Dongarra J, Garner N, et al. A portable programming interface for performance evaluation on modern processors[J]. The International Journal of High Performance Computing Applications, 2000, 14(3): 189−204

[47] Ghasemirahni H, Barbette T, Katsikas G, et al. Packet order matters! Improving application performance by deliberately delaying packets [C] //Proc of the 19th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2022: 807−827



Jia Ru, born in 1993. PhD. Her main research interests include network function virtualization, network measurement, and performance optimization.

贾　茹, 1993 年生. 博士. 主要研究方向为网络功能虚拟化、网络测量、性能优化.



Jiang Haiyang, born in 1982. PhD, associate professor. His main research interests include software defined network, AIOps, network security, efficient data plane.

姜海洋, 1982 年生. 博士, 副研究员. 主要研究方向为软件定义网络、智能运维、网络安全、高效数据面.



Li Zhenyu, born in 1980. PhD, professor. His main research interests include network architecture and systems, and Internet measurement.

李振宇, 1980 年生. 博士, 研究员. 主要研究方向为网络体系结构与系统、网络测量.



Xie Gaogang, born in 1974. PhD, professor. His main research interests include Internet architecture and systems, and distributed computing systems.

谢高岗, 1974 年生. 博士, 研究员. 主要研究方向为互联网体系结构与系统、分布计算系统.