SysOM-AI：面向生产环境人工智能训练的持续跨层性能诊断

郑宇晟*，[2]，毛文安*，[1]，程舒怡[1], 冯福秋[1], 李光水[1], 廖兆岩[1],黄永卓[1], 肖振伟[1], 李玉清[1], Andi Quinn†，[2]，马涛†，[1][1]阿里巴巴集团，中国[2]加州大学圣克鲁兹分校，美国

arXiv:2603.29235v1 [cs.PF] 31 Mar 2026

摘要

生产规模的人工智能训练中的性能诊断颇具挑战，因为细微的操作系统级问题可能引发连锁 GPU 延迟和网络性能下降，导致数千个 GPU 的训练效率降低。现有性能分析工具仅覆盖单个系统层，开销高得难以接受（10–30%），或缺乏持续部署能力，因此需要耗时数日的人工分析。我们认为，要弥补这一差距，必须借助操作系统级插桩和分层差分诊断，实现持续的跨层可观测性。我们介绍 SysOM-AI，这是一套生产环境可观测性系统，通过自适应混合栈展开和基于 eBPF 的跟踪，持续整合 CPU 栈性能分析、GPU 内核跟踪和 NCCL 事件插桩，开销不到 0.4%。SysOM-AI 已在阿里巴巴部署一年多，覆盖超过 80,000 个 GPU，帮助诊断了 94 个已确认的生产问题，并将诊断时间中位数从数天缩短至约 10 分钟。

CCS 概念

• 计算机系统组织 → 异构（混合）系统；• 软件及其工程 → 软件性能。

关键词

性能诊断、人工智能训练、eBPF、栈展开、性能分析

1 引言

大规模人工智能训练作业日益占据现代生产数据中心的主导地位，通常会同时消耗数千个 GPU。在如此规模下，由操作系统级调度、GPU 热行为或网络争用引起的微小性能低效，都可能逐步演变为严重的速度下降，从而影响整体训练吞吐量 [24]。以往的特征分析研究表明，此类问题是训练作业低效的主要来源之一，而诊断通常需要数天时间 [11]。诊断并缓解这些问题仍然是一项重大的运维挑战。

当前的性能分析工具主要聚焦于单层可见性（表 1）：GPU 专用性能分析器（例如 NVIDIA Nsight [20]）会监控 GPU 执行，但会带来 10–30% 的开销 [19]；面向 CPU 的工具（例如 Strobelight [34]，async- profiler [22]）能够高效捕获调用栈，但无法观测 GPU 或分布式 NCCL 事件；而框架级性能分析器（例如 PyTorch Profiler [26]）则忽略了关键的操作系统级行为。由于开销过高或系统覆盖范围不完整，这些工具无法在大规模生产环境中持续运行。因此，操作员必须手动关联来自多个来源的碎片化数据，大幅延长了诊断时间。

我们认为，持续的跨层可观测性（同时捕获并关联 CPU、GPU 和网络活动）对于识别生产环境人工智能训练中性能下降的根因至关重要。基于这一认识，我们开发了 SysOM-AI，这是一套生产环境性能分析系统，利用操作系统级 eBPF 插桩，以轻量级方式常驻采集 CPU 调用栈（包括完整的内核调用链）、GPU 内核耗时和 NCCL 集合通信事件。中央分析服务会关联所有秩的这些信号，并执行分层差分诊断：对比异常秩与正常秩的 GPU、CPU 和操作系统性能分析数据，以系统地定位根因所在层；或与历史基线对比，以发现一致性性能下降。要实现生产规模的部署，必须应对多项挑战，包括通过自适应混合栈展开尽可能降低性能分析开销、实现集中式符号解析以避免节点内存压力，以及在没有调试符号的情况下开发框架无关的 NCCL 跟踪。

SysOM-AI 已在阿里巴巴生产环境的集群中部署一年多，覆盖超过 80,000 个 GPU 和 10,000+ 个节点，每日处理 ~400TiB 的性能分析数据。在本文报告的六个月评估期内，它帮助诊断出 94 个经确认的性能问题（GPU 热降频、网卡软中断争用、VFS 锁争用、日志开销），这些问题未被现有的单层工具检测到，并将诊断时间中位数从数天缩短至约 10 分钟。

本文作出以下贡献：

（1）我们设计并实现了 SysOM-AI，这是一套面向生产环境人工智能训练的持续跨层性能分析系统，以低于 0.4% 的开销整合了 CPU、GPU 和 NCCL 可观测性（§3–4）。（2）我们提出了一种分层差分分析方法，通过比较不同秩之间的情况（针对拖尾进程），以及与历史基线进行比较（针对一致性性能下降），系统地隔离根因，并使用每个通信组的 *CPU 基准线* （§3.1）。（3）我们通过 eBPF 实现了自适应混合栈展开、集中式符号解析和框架无关的 NCCL 跟踪，从而以低开销实现常开式性能分析（§3.3–3.2）。

*同等贡献（共同第一作者）。†通讯作者：boyu.mt@taobao.com, aquinn1@ucsc.edu。

表 1：与人工智能分析工具的比较。✓= 支持，✗= 不支持。"内核" = 操作系统内核级信号。"范围"= 单个进程与集群范围。

Tool	Py	C++	内核	范围	GPU	开销
MegaScan [32]	✓	✗	✗	Proc	✗	Low
MegaScale [14]	✗	✗	✗	Proc	✓	Low
DeepSpeed Prof. [18]	✓	✗	✗	Proc	✓	Med
Nsight Systems [20]	✗	✓	✗	Proc	✓	High
Strobelight [34]	✗	✓	✗	Proc	✗	Low
async-profiler [22]	✗	✓	✗	Proc	✗	Low
SysOM-AI	✓	✓	✓	集群	✓	Low

在这些工具中，没有任何一种会收集中断处理或调度器活动等内核级操作系统信号。SysOM-AI 增加了具备操作系统内核可见性的持续集群范围可观测性。

（4）我们报告了为期一年的生产环境部署情况，覆盖 80,000+ 个 GPU，并通过端到端案例研究展示了对现有工具无法发现的问题进行跨层诊断的实例（§5）。

2 背景与动机

我们首先介绍分布式人工智能训练的性能特征，然后通过一起生产环境事故说明 SysOM-AI 的设计动机。

2.1 人工智能训练性能特征

现代大规模人工智能训练采用同步数据并行或流水线并行，每次迭代都包含前向传播、反向传播、通过集合通信（例如全规约、ReduceScatter）进行梯度规约，以及参数更新 [14]。由于集合通信会阻塞，通信组中最慢的秩决定了所有秩的迭代时间。即使每个秩只有很小的延迟，在大规模场景下也会累积：对于一个包含 1,000 个 GPU、每次迭代耗时 100 ms 的作业，单个秩每次迭代慢 0.3 ms，每天就会浪费超过 70 GPU小时。以往研究表明，这类性能问题十分常见，根因多种多样，涉及 GPU 硬件、操作系统内核、网络和应用程序运行时 [11, 24], ，因此若无法同时获取所有层的性能分析数据，就很难确定究竟是哪一层出了问题。

2.2 动机示例

来看一个生产环境集群中的场景：在一个包含 8 个秩的通信组中，排名 4 的秩每次进入 NCCL 集合通信都晚 0.6 ms，拖慢了整个通信组。操作员的第一反应是检查 GPU 利用率，但 nvidia-smi 显示所有秩的利用率都为 100%，且 GPU 内核耗时相差不到 1%。网络监控未发现丢包或延迟异常。随后，操作员附加了 CPU 性能分析器（perf），但由于 asm_common_interrupt 编译时未启用帧指针，基于 FP 的栈展开在中断边界处截断了调用栈，导致无法看到其下方的内容。操作员因错误归因，发现某个无关函数的 CPU 时间异常偏高，随后花了两天时间追查这一错误线索。

实际根本原因是网卡软中断（NET_RX_SOFTIRQ）绑定到了与 NCCL 通信线程共享的 CPU 核心上。要诊断这一问题，需要具备三项现有工具无法同时提供的能力：(1) 跨秩比较，以发现 GPU 性能分析数据相符，但 CPU 性能分析数据存在差异；(2) 准确的内核调用栈，以揭示从 asm_common_interrupt 到 do_softirq 再到 napi_gro_receive 的完整中断链；以及 (3) 始终开启的采集，因为该问题间歇性出现，事后连接性能分析器也无法复现。我们在 §5.4（案例 2）中完整详述了这一案例。

2.3 挑战

这个示例揭示了现有工具中的四个根本性不足，SysOM-AI 针对每项不足都设计了专门的机制。

C1：缺乏跨层诊断方法。现有工具各自针对单一层运行：Nsight Systems [20] 分析 GPU 内核，Strobelight [34] 分析 CPU 调用栈，而网络计数器则跟踪链路级统计数据。没有工具能够系统地比较 GPU、CPU 和操作系统层的性能分析结果，从而确定哪个层导致了性能下降。操作员必须手动关联碎片化数据，因此诊断通常需要数天。SysOM-AI 通过分层差分诊断（§3.1）解决这一问题。

C2：与框架耦合的 NCCL 插桩。跨秩诊断需要逐集体操作计时，以检测拖尾进程。MegaScan 等先前工具通过在框架层对 NCCL 进行插桩来实现这一点，这使其与 Megatron 的代码路径紧密耦合；切换框架就需要重新插桩。此外，生产环境 NCCL 构建版本发布时不包含调试符号，因此难以识别通信组并匹配集合通信实例。SysOM-AI 通过基于 eBPF 的框架无关 NCCL 跟踪（§3.2）解决这一问题。

C3：在生产环境可接受的开销下，调用栈仍不准确。即使在单个层内，CPU 分析器生成的数据也不可靠。由于使用 -fomit-frame-pointer 编译的函数（GCC/Clang 在 -O2 下的默认设置）会导致调用栈被截断或归因错误，基于 FP 的栈展开在生产环境 AI 工作负载中的栈帧准确率较低；在一个生产环境案例中，这导致 50% 的 CPU 采样被错误归因到同一个函数（§5.2）。基于 DWARF 的栈展开虽然准确，但每次采样的开销显著更高，在 100 Hz 下会消耗 CPU 核心数个百分点，因此无法用于始终开启的场景。SysOM-AI 通过自适应混合 FP+DWARF 栈展开解决了这一问题（§3.3）。

C4：符号解析在大规模场景下失效。生产二进制文件通常会剥离调试符号，并将其单独存储。在每个性能分析节点上加载完整符号表（大型库的符号表大小为 600 MB–1GB）会导致内存不足故障；而当符号表稀疏时，节点端的最近地址匹配会产生错误的归因（§5.2）。SysOM-AI 通过集中式的基于 Build-ID 索引的符号解析解决了这一问题（§3.4）。

3 系统设计

为应对 §2.3 中指出的挑战，SysOM-AI 将基于 eBPF 的常驻数据采集与集中式分析流水线相结合。图 1 展示了该架构。

总体而言，SysOM-AI 分三个阶段运行：（1）在每个节点上通过基于 eBPF 的探针持续进行数据采集；（2）集中式符号解析与聚合；（3）分层差分诊断，精准定位系统各层的根因。我们首先介绍诊断方法（§3.1），然后介绍三项支撑机制：

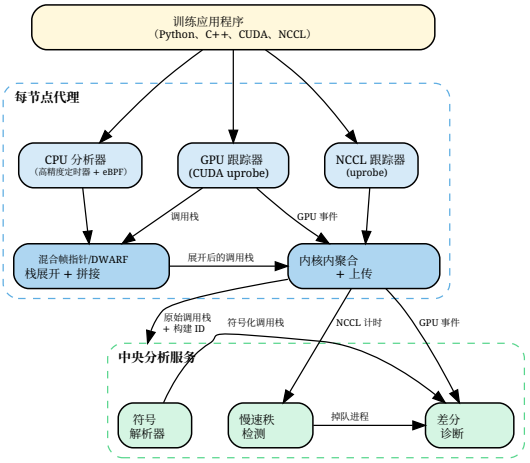


图 1：SysOM-AI 架构。每个节点都运行一个代理，用于收集 CPU 调用栈（高精度定时器 +eBPF）、GPU 内核计时数据（ CUDA uprobe）和 NCCL 集合通信事件（uprobe）。调用栈通过自适应混合帧指针/DWARF 方式展开，在 Python 和 C++ 帧之间拼接，在内核中聚合，随后上传到中央服务进行符号解析、慢速秩检测和分层差分诊断。

NCCL 可观测性（§3.2）、自适应栈展开（§3.3）和集中式符号解析（§3.4）。

3.1 分层差分诊断

CPU 基准线。我们将**CPU 基准线** 定义为：在滑动窗口内，基于最近 W次迭代（默认 $W = 100$）计算出的每个通信组的基线 CPU 性能分析。对于通信组 g中的每个函数 f，我们计算所有秩中该函数 CPU 占比 μ_f^g的均值和标准差 σ_f^g。当某个秩 r中的任一函数超过 $\mu_f^g + k\sigma_f^g$（默认 $k = 2$）时，就将该秩标记出来。基准线是在组内所有 秩上同时计算的；无需事先将秩划分为“健康”和“不健康”两类。拖后者属于统计离群值：在包含 N个秩的情况下，单个异常秩只会使 μ偏移 $1/N$，并使σ略微增加，因此该离群秩上的函数仍会超过阈值。对于$N \geq 8$ （通信组的典型大小），单个拖尾进程对组统计数据的影响是有限的（§5.4）。

慢速秩检测。使用 §3.2 中描述的 NCCL 插桩，SysOM-AI 会记录每次集合通信的主机端进入时间戳和 GPU 端持续时间。跨秩时钟对齐利用集合通信的屏障语义：由于所有秩都必须进入并退出，因此最早的进入时间和最晚的退出时间为对齐跨进程时间戳提供了天然的参考点。如果某个秩在W次迭代的滑动窗口内，其进入时间超过 $\mu + k\sigma$（默认值为 $k = 2$），则会将该秩标记为拖尾进程（§5.4）。

跨秩差分诊断。检测到拖尾进程后，SysOM-AI 会生成逐层差分性能分析数据：

(1) **GPU 差分**：比较 GPU 内核执行时间。所有内核都出现一致的减速，说明可能存在硬件问题（热节流、内存错误）。特定内核出现减速，则可能是软件问题（算子级变更）。(2) **CPU 差分**：如果 GPU 性能分析数据一致，则比较 CPU 火焰图。新增的热点函数或特定路径耗时增加，表明可能存在主机端干扰（中断、调度、输入/输出）。(3) **操作系统差分**：如果应用层 CPU 性能分析数据一致，则比较操作系统子系统信号：中断计数（/proc/interrupts）、调度器延迟（sched_stat 跟踪点）以及 NUMA 迁移事件。由于这些事件持续时间短且频率高，可能不会出现在采样火焰图中。

时间基线比较。 未检测到拖尾进程但绝对迭代时间增加时，SysOM-AI 会将当前的分组火焰图与存储在集中式日志服务中的历史基线进行比较。相较于基线，CPU 占比增加超过 δ（默认 0.5%）的函数会被标记为性能退化候选项。这补充了跨秩分析：跨秩比较用于识别哪个秩存在异常，而时间比较用于识别性能何时发生变化，以及哪条代码路径导致了变化。

3.2 框架无关的 NCCL 可观测性

上述诊断流水线依赖逐集合通信计时数据。在生产环境中获取这些数据需要解决三个问题。

框架无关追踪。 SysOM-AI 不在框架层对 NCCL 进行插桩（§2.3，C2），而是在库边界拦截 NCCL 调用，从而实现框架无关追踪，能够在 Megatron、DeepSpeed、ms-swift 等框架中以相同方式运行。

无需调试符号的通信组识别。通信组识别通常需要借助调试信息解析 ncclComm 结构，而生产环境中的 NCCL构建版本缺少这些信息（§2.3，C2）。SysOM-AI 会根据已知的版本特定偏移量预先解析 NCCL结构布局（目前支持 NCCL 2.14–2.21 和 ACCL），因此无需运行时调试信息即可识别通信组；代价是当 NCCL 内部布局发生变化时需要进行配置更新。

集合通信实例分离。在一个通信组内，可能同时有多个集合通信处于执行中。标准方法使用 ncclComm 中的 opCount字段，但对于点对点操作，该计数器位于 GPU内存 中，读取开销较大。SysOM-AI 则利用时间重叠：识别不同秩上在时间上重叠的操作，并将其归为同一集合通信实例，从而避免访问 GPU内存。

3.3 自适应混合帧指针+DWARF栈回溯

准确的 CPU 火焰图是 §3.1 中差分诊断的核心。关键在于，在典型的二进制文件中，大多数函数都可以通过基于 FP 的栈展开正确处理，即使用 -fno-omit-frame-pointer 编译，或以其他方式保留帧指针约定的函数。在我们的生产二进制文件中，约 20% 的函数需要使用 DWARF，主要是那些经过会省略帧指针的优化编译的 C++ 库。

算法 1 自适应混合栈展开

要求： 寄存器：PC、SP、帧指针
确保： 调用栈 S
$S \leftarrow [\,]$
当PC 位于映射的可执行区域**时执行**
  $m \leftarrow$ 获取标记(构建 ID(PC)，偏移量(PC))
  如果 m =未标记 **则**
    $(pc', sp', fp') \leftarrow$ UnwindFP展开算法(PC，SP，帧指针)
    如果ValidateCallerPC(pc', sp', PC) **则**
      设置标记(PC，帧指针)
    else
      $(pc', sp', fp') \leftarrow$ UnwindDWARF展开算法(PC，SP)
      设置标记(PC，dwarf)
    结束 if
  否则，如果 m =帧指针 **则**
    $(pc', sp', fp') \leftarrow$ UnwindFP展开算法(PC，SP，FP)
  else ▷ m = dwarf
    $(pc', sp', fp') \leftarrow$ UnwindDWARF展开算法(PC，SP)
  结束如果
  S.追加(pc')
  (PC，SP，FP) $\leftarrow (pc', sp', fp')$
结束循环

SysOM-AI 会针对每个函数学习所需的方法并缓存决策，从而将 DWARF 的开销分摊到一段时间内。

栈展开方法标记。每个函数由其构建 ID 和代码偏移量标识。SysOM-AI 维护一个哈希映射 Map<(BuildID, Offset) → Marker>，其中 Marker ∈{unmarked, fp, dwarf}。算法 1 描述了栈展开过程。

验证。ValidateCallerPC 函数通过查询进程的内存映射（/proc/[pid]/ maps），检查展开后的返回地址 pc' 是否位于映射的可执行区域内。具体而言，它会验证两个条件：(1) pc' 位于某个可执行 ELF 段的 [start, end) 范围内；(2) 栈指针 sp' 单调递增（即调用栈正向展开）。如果任一检查失败，则帧指针结果无效，通常是因为函数在编译时使用了 -fomit-frame-pointer，导致帧指针寄存器保存的是通用值，而非调用者的帧地址。

标记的收敛与稳定性。在我们的生产环境工作负载中，大多数不同的调用点标记会在第一个性能分析窗口内收敛。仅在罕见代码路径（错误处理、初始化）中遇到的函数，首次遇到时会回退到 DWARF，此后则会缓存结果。

标记是稳定的，因为函数的帧指针行为在编译时就已确定，运行时不会改变。两个例外是动态加载库（dlopen）和即时编译代码，其中的函数最初没有标记，并经历相同的收敛过程；我们将在 §4 中介绍检测机制。

成本分析。由于大多数函数使用缓存的帧指针路径，只有少数函数需要 DWARF，因此稳态下每次采样的开销接近纯帧指针回溯。DWARF 栈帧通常出现在调用栈较高的位置，并且每次采样贡献的帧数较少，因此在实际应用中进一步降低了开销。我们在 §5.1 中对此进行了量化。

3.4 集中式延迟符号解析

即使栈展开准确无误，也必须对原始地址栈进行符号化，才能生成可读的火焰图。

问题。生产环境人工智能二进制文件（例如盘古分布式存储客户端、PyTorch 共享库）通常会被剥离，调试符号仅存放在单独的文件中。在每个性能分析节点上加载完整符号表（大型内部库的符号表大小为 600 MB–1GB），当同时对多个进程进行性能分析时会导致内存溢出。此外，当符号表稀疏时，节点端采用"最接近的较低地址"进行符号匹配会产生错误归因；我们在 §5.2 中记录了一个具体案例，其中 50% 的 CPU 采样被错误归因。

设计。节点代理会收集原始地址栈，以及每个已加载二进制文件的构建 ID。上传时，代理会根据构建 ID 检查中央符号仓库中是否已有对应的符号文件；如果没有，则提取并上传调试符号。中央解析器使用完整符号表执行符号查找，通过 $O(\log n)$ 查找准确获取函数名，无需将整个文件加载到内存中。在我们的部署中，单个区域的代码库中存储了超过 170,000 个不同的构建 ID。

4 实现

节点代理使用 C（eBPF 程序）和 Rust（用户空间守护进程）实现；中央服务使用 Go 实现。

eBPF 程序与代理通信。代理使用三种 eBPF 程序类型：(1) 附加到高精度定时器的 perf_event，用于周期性 CPU 采样（可配置范围为 10–999 Hz；默认值为 99 Hz，以避免与定时器中断发生锁步混叠）；(2) 针对 CUDA 运行时函数（cuLaunchKernelEx）的 uprobe，用于拦截 GPU 内核启动；(3) 针对 NCCL 入口点（ncclAllReduce、ncclReduceScatter 等）的 uprobe，用于集合通信跟踪。为在内核中聚合数据，eBPF 程序会对每个堆栈跟踪进行哈希，并在 BPF 哈希映射中递增相应调用栈的计数器。用户空间守护进程每 5 秒清空一次该映射，与逐次采样流式传输相比，数据量可减少 10–50×。节点代理通过 Unix 域套接字与训练应用程序通信。应用程序启动时会注册其进程 ID 和 NCCL 通信器信息；代理利用这些信息，将 eBPF 收集的调用栈与应用程序级元数据（作业 ID、秩、通信组）关联起来。这种设计无需修改训练脚本中的任何代码；只需设置一个环境变量（SYSOM_SOCK_PATH）即可。

DWARF 预处理。eBPF 程序的调用栈限制为 512 字节，且不支持动态内存分配，因此无法完整解释 DWARF 调用帧信息。SysOM-AI 通过两阶段方案解决这一问题。在代理启动时（阶段 *1*），用户空间守护进程会解析每个已加载二进制文件的 .eh_frame 段，并为每个帧描述条目（FDE）提取规范帧地址（CFA）规则（寄存器 + 偏移量）、返回地址偏移量和程序计数器范围。随后，这些信息会编译为一个已排序数组，并加载到 eBPF 映射中。使用 DWARF 表达式（而非简单寄存器+偏移量规则）的 FDE 会被标记为"复杂"，并通过用户空间回退进行栈展开；我们的生产二进制文件中绝大多数 FDE 都使用简单规则。在运行时（阶段 *2*），eBPF 栈展开器会对 FDE 数组执行二分查找（$\lceil \log_2 M \rceil$ 次迭代，适用于 M 个条目；通常为 16 个 $M \approx 50,000$），并通过一次内存解引用计算 CFA 和返回地址，完全符合 eBPF 的限制。每个二进制文件的预处理会增加 ~200 毫秒

的启动时间。对于动态加载库（dlopen），代理会每隔 5 秒轮询 /proc/[pid]/maps，以检测新的映射，并增量预处理新库的 .eh_frame 段。即时编译代码（例如由 torch.compile 生成的代码）通过 perf_event_mmap 通知进行检测；由于其帧布局可能不符合标准 ABI，因此会保守地将 JIT 编译的函数标记为 dwarf。多个 CPU 核心可能同时遇到同一个尚未标记的函数；我们在标记映射上使用原子比较并交换操作，使并发竞争最终收敛到相同的标记值。

多运行时调用栈拼接。 人工智能训练调用栈横跨 Python 和原生 C++ 帧。对于 CPython 进程，代理通过 _PyRuntime 全局符号和线程局部存储偏移定位 PyThreadState，随后遍历帧链（对于 Python 3.10 及更早版本，使用 PyFrameObject.f_back；对于 3.11+，使用 _PyInterpreterFrame），以提取代码对象指针和行号。原生 C++ 帧使用混合 FP/DWARF 栈展开器。随后，以线程的原生栈指针为连接点，将两者拼接为统一调用栈。此方法无需修改 Python 解释器，可用于未经修改的生产环境部署。

数据管道与符号管理。 从 ELF .note.gnu.build-id 节中提取构建 ID。调试符号通过分块传输（每段 64 MB）上传，以限制节点上的峰值内存占用。代码库采用紧凑的二进制格式，其中偏移量和符号数据节由头部索引，支持 $O(\log n)$ 查找，而无需加载整个文件。数据每 30 秒批量上传至集中式日志服务（SLS）。中央服务在数分钟内处理摄取的数据，计算各组聚合值并执行慢任务检测。每个节点代理的常驻内存占用为 ~200MB，其中包括 eBPF 映射和预处理后的 DWARF 表。目前，该系统已在多个集群中的 10,000+ 个节点上部署，覆盖 80,000+ 张 GPU 卡，日均数据总量为 ~400TiB。NCCL 跟踪与框架无关，并已在生产环境中的 Megatron（每日 357 个任务）、DeepSpeed（51 个）和 ms-swift（113 个）上完成验证。

5 评估

SysOM-AI 已在阿里巴巴部署一年多，覆盖 80,000+ 个 GPU 和 10,000+ 个节点，处理 ~

每天处理 400 TiB 的性能分析数据。在六个月评估期内，系统触发了 2,649 个诊断事件（图2）：606 个 GPU 硬件问题、269 个操作系统干扰问题（诊断时间中位数为 10 分钟）、320 个网络问题（中位数为 30 分钟），以及通过基于日志的SOP规则匹配识别出的 1,454 个软件问题（中位数为 1 分钟）。其中，94 个被确认为需要基于性能分析进行根因分析的跨层事件。

我们从四个维度评估该系统：训练开销（§5.1）、栈回溯准确率（§5.2）、符号解析（§5.3），以及生产环境中的端到端案例研究（§5.4）。

5.1 训练吞吐量开销

设置。 我们测量了在 2× NVIDIA A100-80GB GPU 上使用 PyTorch 2.1、NCCL 2.18、序列长度 1500 和批量大小 8 训练 Llama-3.2-1B-Instruct 时对吞吐量的影响。报告值为 20 个训练步数的平均值（第 21–40 步，即经过 20 步预热后）。“采样率”

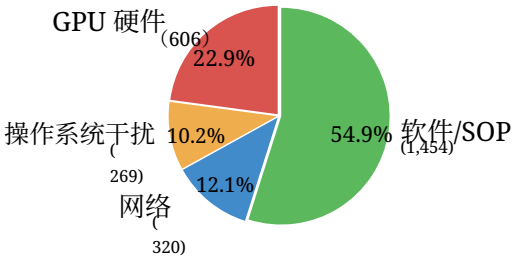


图2：六个月评估期内，按根因类别划分的 2,649 个诊断事件分布。

表 2：不同采样率下的训练吞吐量开销。基线吞吐量：15.11 次迭代/秒。

采样率	性能分析期间	性能分析之后
1%	−0.06%	−0.20%
10%	−0.33%	−0.20%
20%	−0.33%	−0.26%
40%	−0.72%	0%
80%	−1.25%	−0.20%
100%	−1.72%	−0.26%

表示触发完整堆栈采集的高精度定时器时钟节拍（频率为 99 Hz）的比例。

表 2 展示了结果。在生产环境默认设置下（10% 的采样率，相当于 ~10 samples/s），开销为 0.33%。即使在 100% 采样率下（99samples/s），开销也仅为 1.72%。这种低开销得益于三个因素：(1) 通过 BPF 映射在内核中聚合调用栈，消除了逐次采样的用户空间切换；(2) 混合栈展开器避免了对大多数栈帧使用 DWARF 带来的开销；(3) 符号解析延迟至中央服务执行，而非在节点上进行。

停止性能分析后，吞吐量恢复至基线（偏差为 0–0.26%，处于 0% 采样时观察到的 0.33% 基线测量噪声范围内）。

5.2 栈展开准确性

图 3 比较了生产环境 AI 工作负载中不同栈展开配置的栈帧准确率。仅基于帧指针的栈展开准确率仅为 ~5%，因为大多数生产环境二进制文件（Python、C++ 使用 -O2 编译）都会省略帧指针；只有 Go 二进制文件会始终保留帧指针。SysOM-AI 的混合栈回溯将准确率提高到 70%，而添加集中式符号解析（可消除节点端符号表稀疏所导致的错误归因）后，准确率进一步提升至 95%。

5.3 符号解析

符号误归因案例。 在生产环境中的分布式存储工作负载中，节点端符号解析导致函数 pangu_memcpy_avx512（地址为 0x11c6aa0，由于符号条目稀疏，覆盖了 18MB 的地址范围）吸收了超过 50% 的 CPU 采样，从而生成了一张虚假的递归火焰图（图 4）。通过使用完整符号表进行集中式解析

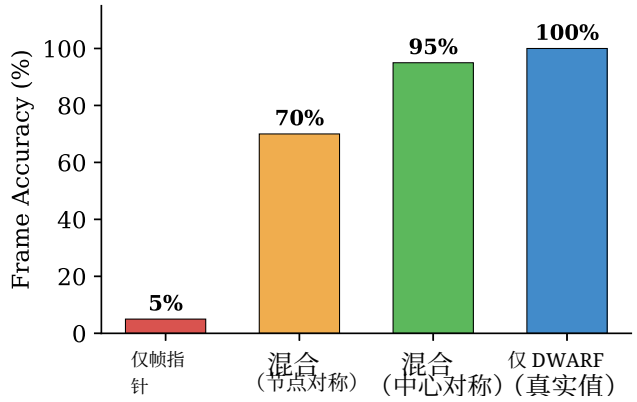


图 3：生产环境 AI 工作负载中的栈回溯栈帧准确率。由于帧指针普遍缺失，仅 FP 的准确率只有 5%。SysOM-AI 采用混合栈回溯并结合集中式符号解析，准确率达到 95%。

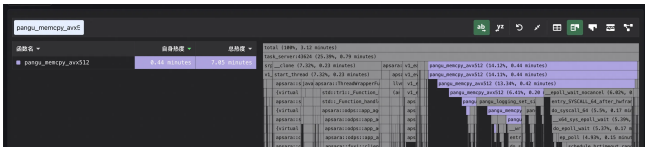


图 4：节点侧解析导致的符号错误归因。由于符号条目稀疏，函数 pangu_memcpy_avx512 吸收了许多无关函数的采样，从而产生一个虚假的热点。使用完整符号表进行集中式解析，则可正确归因这些地址。

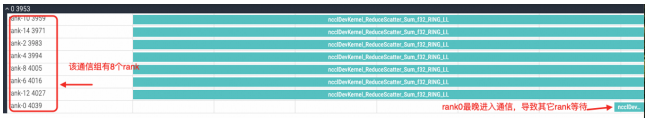


图 5：案例 1 的慢任务检测。各秩的 NCCL 集体通信进入时间显示，秩 0 最后进入，表明它是这个 8 秩通信组中的拖尾进程。

（以构建 ID ba6bc0e5... 为键），0x23XXXXXX 范围内的地址被正确归因到数十个不同的函数（PrepareWatcher::Start、IoWatcher::onReady 等），递归伪影也完全消失。

5.4 端到端案例研究

我们展示了生产环境中的五个代表性案例，以说明 SysOM-AI 如何定位不同系统层的根因。

5.4.1 案例 *1*：*GPU* 热降频。**症状**。慢节点检测在一个包含 8 个秩（秩 0–7）的通信组中标记了秩 0。与其他秩相比，秩 0 始终晚 0.4 毫秒进入 ncclDevKernel_ ReduceScatter 集合通信（图 5）。

诊断。SysOM-AI 生成了秩 0（拖尾进程）与秩 7（正常）之间的 GPU 火焰图差异，如图 6 所示。

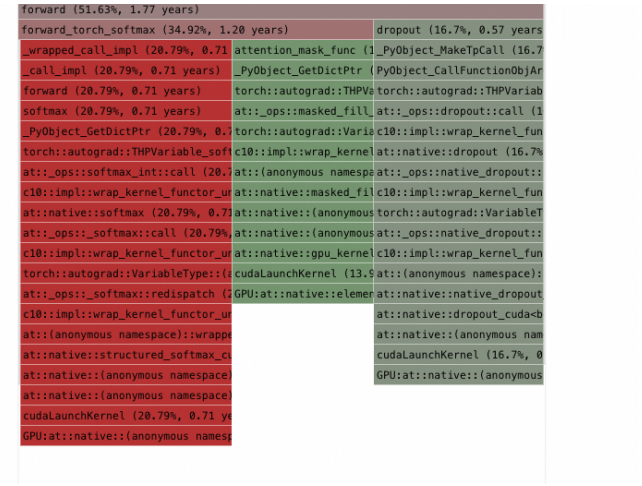


图 6：案例 1 的差分 GPU 火焰图。所有内核类型中，红色帧（拖尾秩 0）均一致宽于绿色帧（正常秩 7），这表明是全局 GPU 频率降低，而非特定操作员问题。

秩 0 上的前向传播耗用了 51.6% 的 GPU 时间（1.77 秒），而秩 7 上为 48.1%（1.65 秒）。所有内核类型均出现了同比例的减速（Softmax：20.8% 对比 19.1%；Dropout：16.7% 对比 15.3%），这与全局频率降低的情况相符，而非特定操作员的问题。

根本原因。NVIDIA DCGM 确认，由于环境温度超标，GPU 0 的频率被限制在 1,200 MHz（额定频率为 1,410 MHz）。标准利用率指标（nvidia-smi）显示所有秩的利用率均为 100%，掩盖了频率降低的情况；跨秩差分诊断（§3.1）揭示了均匀减速模式，促使操作员查看 DCGM。解决热问题后，训练吞吐量提升了 30%。

5.4.2 案例 *2*：网卡软中断争用。我们借助 SysOM-AI 的完整诊断数据，重新审视 §2.2 中的场景。

诊断。GPU 差分：SysOM-AI 对比了排名 4（拖尾进程）和排名 6（正常进程）。没有显著差异；两者的 GPU 内核耗时相差不到 1%。CPU 差分：存在显著差异（图 7）。排名 4 的 CPU 性能分析显示，其总 CPU 时间的 1.74% 用于 net_rx_action→ napi_poll → virtnet_poll → virtnet_receive 路径，完整的中断链清晰可见：asm_common_interrupt→ common_interrupt → irq_exit_rcu → do_softirq → net_rx_action（1.74%）→ napi_poll（1.74%）→ virtnet_poll（1.52%）→ virtnet_receive（1.48%）→ napi_gro_receive（0.81%）。排名 6 在此路径上的占比为 <0.1%。

根本原因。网络软中断（NET_RX_SOFTIRQ）绑定到了与 NCCL 通信线程共享的 CPU 核心上，导致周期性抢占。解决方法：调整 /proc/irq/*/ smp_affinity，将网卡中断与训练核心隔离。诊断从告警到确定根本原因约耗时 10 分钟。

5.4.3 案例 *3*：*VFSDentryLockContention*。**症状**。大型训练作业中的部分节点迭代时间延长了 60%。

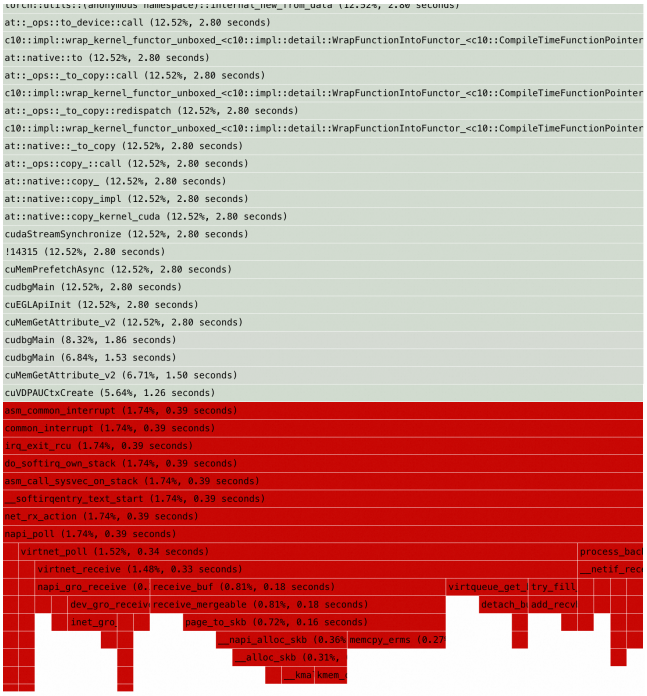


图 7：案例 2 的差分 CPU 火焰图。底部区域（红色）显示了净 _rx_action → napi_poll →virtnet_receive 中断链，该链存在于拖尾秩上，而正常秩上没有，由此确定网卡软中断争用是根本原因。



图 8：案例 3 的内核火焰图。所有采样到的 CPU 时间都处于 queued_spin_lock_slowpath 中，该路径经由 VFS 路径查找（__legitimize_path、dput、lookup_fast）到达。这种锁争用对用户态分析器不可见。

慢速秩检测标记出了受影响的秩，但 GPU 内核耗时与正常秩相差不超过 1%。

诊断。 GPU 差分未发现异常。CPU 差分显示，拖尾秩的大部分 CPU 时间都耗费在用户态分析器无法观察到的内核路径上（图 8）。SysOM-AI 通过支持完整内核栈的混合栈回溯生成的内核火焰图显示，100% 的采样时间都集中在 queued_spin_lock_slowpath 上，并汇聚于三条调用路径：（1）__legitimize_path →lockref_get_not_dead（占采样的 65%），这是惰性路径遍历期间目录项验证失败时触发的 RCU 到引用计数的回退；（2）terminate_walk → dput（34%），openat 之后释放目录项引用；以及（3）lookup_fast → unlazy_child（11%），路径解析期间进行的子目录项查找。所有路径都源自 do_sys_openat2，表明系统范围内发生了针对目录项自旋锁的 VFS 争用。

根本原因。某个系统管理组件定期在受影响的节点上执行 systemctl daemon-reload，触发了大规模目录项缓存失效。这迫使所有并发文件操作（包括训练数据加载和检查点 I/O）从无锁 RCU 路径遍历回退到引用计数路径，导致目录项自旋锁争用，阻塞了训练线程。此案例体现了跨进程操作系统级干扰：根本原因是一个无关的系统服务，而要诊断此问题，必须能够查看内核栈——现有的 AI 性能分析工具均不具备这一能力（表 1）。

5.4.4 案例 *4*：过高的用户态日志开销。**症状**。一次常规基础设施更新后，一个大规模训练作业的吞吐量下降了 10%。未检测到 GPU 级异常；所有秩的迭代时间相近（没有拖尾进程），但迭代时间整体均匀增加。

诊断。 由于没有检测到拖尾进程，SysOM-AI 使用时间基线比较（§3.1），将当前各组火焰图与更新前基线进行比较。CPU 火焰图显示，SLS::LogClient::Send → protobuf::Serialize → memcpy 中出现了一条新的热点路径。更新前基线中不存在这条路径。

根本原因。 基础设施更新将阿里巴巴 SLS（简单日志服务）客户端的日志级别从 INFO 提高到 DEBUG，新增了每次迭代记录张量统计信息的日志。序列化开销消耗了训练线程的 CPU 周期，导致所有秩的运行速度都一致变慢。解决方法：恢复原来的日志级别。由于所有秩都一致变慢，因此没有触发拖尾进程告警；常开式性能分析使时间基线比较（§3.1）得以发现这条新的热点路径。

5.4.5 案例 *5*：数据摄取瓶颈。 **症状**。一个大规模训练作业报告吞吐量下降了 30%，而模型配置和集群拓扑均未发生变化。

诊断。NCCL 可观测性（§3.2）确认各秩的集合通信延迟一致，排除了通信和拖尾问题。相反，CPU 火焰图凭借混合栈回溯（§3.3）和集中式符号解析（§3.4）获得了准确的调用栈，揭示 cpfs（云原生并行文件系统客户端）和 ossutils（对象存储工具）上的耗时显著高于正常基线。

根本原因。 数据集大小增加，但存储层级未相应升级，导致数据管道的吞吐量下降。I/O 密集型数据加载耗尽了主机 CPU 资源，使所有秩的运行速度都一致变慢。解决方案是升级存储后端并提高数据加载器并行度。单靠 GPU 性能分析只能看到正常的内核耗时。要将性能下降归因于数据管道，需要结合观察正常的 NCCL 计时，以及存储 I/O 路径中主机侧 CPU 占用偏高的情况（§3.1）。我们在其他多个事件中也观察到了类似的 I/O 相关性能下降：在一个案例中，共租户争用导致对象存储（OSS）速率限制，使训练吞吐量下降了 30%；在另一个案例中，获取训练数据时网络速度缓慢，导致吞吐量下降 9×。

6 相关工作

人工智能训练和 GPU 性能分析。表 1 对比了 SysOM-AI 与现有工具。MegaScan [32] 跟踪 Python 层级的 Megatron

的执行情况；MegaScale [14] 收集 GPU 内核运行时数据；Nsight Systems [20] 以 10–30% 的开销提供 GPU 时间线 [19]；Stro- belight [34]分析 C++ 调用栈，但无法查看 GPU 或 Pytho n；DeepContext [29] 将 Python 上下文与 GPU 代码关联起来，但无法查看操作系统内核；Flare [2] 在 6,000 个 GPU 上部署全栈跟踪，但侧重于故障检测，而非持续性能分析。所有这些工具都在进程级别运行，无法查看内核栈。对于 GPU 内部分析，Neutrino [12] 和 KPerfIR [10] 提供指令级 GPU 内核性能分析，可与 SysOM-AI 的主机端方法互为补充。近期的特征研究 [11, 24] 对训练故障进行了归类，但未提供诊断系统。

持续性能剖析与分布式追踪。 Google 全域性能剖析 [27]开创了始终在线的全机群性能剖析；Parca [23] 和 Pyroscope [8] 提供基于 eBPF 的持续性能剖析，但仅使用帧指针回溯，因此准确性受限。Elastic 通用性能分析 [5]会为 eBPF 预处理 D WARF，但不提供按函数自适应选择。Meta 的 Dynolog [17] 集成了 CPU/GPU 遥测，但未解决栈展开准确性问题。Linux 内核的 ORC 展开器 [25]会预先计算内核空间栈展开表；帧指针回溯 [9] 的每帧开销为 $O(1)$，而 DWARF [4] 准确但开销高；SysOM-AI 的贡献在于按函数自适应组合这两种方法。因果性能分析（Coz [3], BCOZ [1]）和等待分析（wPerf [33]）针对单节点瓶颈；SysOM-AI 则解决与之互补的跨层、跨秩诊断问题。在分布式追踪领域，Fay [6] 建立了从内核到集群的跨层追踪；Pivot Tracing [16],lprof [31], Stitch [30], 和 Hubble [15] 重建分布式请求流；EXIST [28] 提供低开销的服务内追踪；Jaeger [13], Zipkin [21], 和 Sage [7] 则在微服务场景中关联各个跨度。这些方法面向以请求为中心的系统；SysOM-AI 面向批量同步 GPU 训练，采用按秩性能剖析，而非按请求追踪。

7 局限性

SysOM-AI 依赖 Linux eBPF（内核 5.4+）、CUDA 运行时探针以及 NCCL 结构布局（目前支持 NCCL 2.14–2.21 和 ACCL）。Python 栈帧提取支持 CPython 3.10+；不生成标准 ELF 映射的 JIT 运行时不受支持。慢任务检测假设每个通信组中只有少数异常秩；当多数秩同时退化时，统计异常值模型的效力会下降，此时需要进行时间基线比较。诊断覆盖范围因根因类型而异：GPU 侧停顿可被可靠检测；CPU 上的干扰可通过火焰图分析捕获；然而，CPU 外阻塞（例如训练进程等待缓慢的数据读取）无法通过 CPU 上采样捕获，而 RDMA 层面的网络问题虽可识别为集合通信缓慢，但若无额外的网络层检测手段，则无法确定根因。SysOM-AI 通过主机端 CUDA 事件采集 GPU 内核耗时，但不采集 GPU 侧调用栈；更细粒度的分析需要配合使用 Nsight Compute 等工具 [20]。中央分析服务部署在单个集群中，未配置异地冗余；如果服务不可用，节点代理最多可在本地缓存数据 1 小时。

8 结论

操作系统级停顿（调度器争用、软中断、内存压力）是人工智能训练性能问题的重要来源，但现有工具缺乏能够与 GPU 和通信活动相关联的持续操作系统内核可见性。我们介绍了 Sys OM-AI，这是一套生产环境跨层可观测性系统，通过自适应混合栈展开、集中式符号解析、框架无关的 NCCL 可观测性以及分层差分诊断来弥补这一不足。SysOM-AI 已在阿里巴巴覆盖 80,000+ 个 GPU，部署一年多，开销低于 0.4%，栈帧准确率达到 95%，并帮助诊断了 94 个已确认的生产问题，将诊断时间中位数从数天缩短至约 10 分钟。我们认为，这种跨层常开式性能分析方法不仅适用于人工智能训练，也可推广到其他大规模分布式系统，在这些系统中，操作系统级干扰难以隔离。

参考文献

[1] Minwoo Ahn、Jeongmin Han、Youngjin Kwon 和 Jinkyu Jeong。2024。结合阻塞样本分析 CPU 上下线瓶颈。载于 第 *18* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI*）论文集。USENIX 协会。[2] Weihao Cui、Ji Zhang、Han Zhao、Chao Liu、Jian Sha、Bingsheng He、Minyi Guo 和 Quan Chen。2025。Flare：千卡以上规模图形处理器集群中大语言模型训练分歧的异常诊断。arXiv: 2502 05413 [ ] 。cs.OS [3] 2015 Charlie Curtsinger 和 Emery D. Berger。Coz：通过因果性能分析找出影响性能的代码。载于 第 *25* 届操作系统原理研讨会（*SOSP*）论文集。ACM, 184 197 10 1145 2815400 2815409 –。doi:。/。[4] 2017 DWARF 调试信息格式委员会。DWARF 调试信息格式，第 5 版。https://dwarfstd.org/dwarf5std.html。[5] Elastic。2024。Elastic 通用性能分析。https://www.elastic.co/observability/ universal-profiling。基于 eBPF 的持续性能分析，支持 DWARF 栈展开。[6] Úlfar Erlingsson、Marcus Peinado、Si mon Peter 和 Mihai Budiu。2011。Fay：从内核到集群的可扩展分布式追踪。载于 第 *23* 届 *ACM* 操作系统原理研讨会（*SOSP*）论文集。ACM，311–326。doi:10.1145/2043556.2043585[7] Yu Gan、Mingyu Liang、Sundar Dev、David Lo 和 Christina Delimitrou。2021。Sage：微服务中实用且可扩展的机器学习驱动性能调试。载于 第 *26* 届 *ACM* 编程语言与操作系统架构支持国际会议（*ASPLOS*）论文集。ACM，线上，135–151。[8] Grafana Labs。2024。Pyroscope：开源持续性能分析平台。https://pyroscope.io/。[9] Brendan Gregg。2020。*BPF* 性能工具：*Linux* 系统与应用可观测性。Addison-Wesley Professional，波士顿，马萨诸塞州。[10] Yue Guan、Yuanwei Fang、Keren Zhou、Corbin Robeck、Manman Ren、Zhongkai Yu、Yufei Ding 和 Adnan Aziz。2025。KPerfIR：面向现代 AI 工作负载的 GPU 内核性能工具开放式、以编译器为中心的生态系统。载于 第 *19* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI*）论文集。USENIX 协会。[11] Qinghao Hu、Zhisheng Ye、Zerui Wang、Guoteng Wang、Meng Zhang、Qiaoling Chen、Peng Sun、Dahua Lin、Xiaolin Wang、Yingwei Luo、Yonggang Wen 和 Tianwei Zhang。2024。数据中心大语言模型开发的特征分析。载于 第 *21* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI*）论文集。USENIX 协会，圣克拉拉，加利福尼亚州，709–729。[12] Songlin Huang 和 Chenshu Wu。2025。Neutrino：通过可编程探测实现细粒度 GPU 内核性能分析。载于 第 *19* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI*）论文集。USENIX 协会。[13] Jaeger 作者。2024。Jaeger：开源端到端分布式追踪。https://www.jaegertracing.io/。[14] Ziheng Jiang、Haibin Lin、Yinmin Zhong、Qi Huang、Yangrui Chen、Zhi Zeng、Songzhu Du、Kaihua Chen、Rui Peng、Yuxuan Li 等。2024。MegaScale：将大语言模型训练扩展到超过 10,000 个 GPU。载于 第 *21* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI*）论文集。USENIX 协会，圣克拉拉，加利福尼亚州，745–760。[15] Yu Luo、Kirk Rodrigues、Cuiqin Li、Feng Zhang、Lijin Jiang、Bing Xia、David Lion 和 Ding Yuan。2022。Hubble：通过生产环境中的即时方法追踪调试安卓性能。载于 第 *16* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI*）论文集。USENIX 协会。

[16] Jonathan Mace、Ryan Roelke 和 Rodrigo Fonseca。2015。Pivot Tracing：分布式系统的动态因果监控。载于 第 *25* 届操作系统原理研讨会（*SOSP*）论文集。ACM，378–393。doi:10.1145/2815400。2815415[17] Meta。2022。Dynolog：开源系统可观测性。https://github.com/ facebookincubator/dynolog。[18] 微软。2024。DeepSpeed：深度学习优化库。https://github.com/microsoft/DeepSpeed。[19] 英伟达。2024。Nsight Systems：最佳实践——尽量降低开销。https: //docs.nvidia.com/nsight-systems/UserGuide/index.html。根据工作负载不同，完整 GPU 跟踪模式会产生 10–30% 的开销。[20] 英伟达。2024。NVIDIA Nsight Systems 用户指南。https://docs.nvidia.com/ nsight-systems/。[21] OpenZipkin。2024。Zipkin：分布式追踪系统。https://zipkin.io/。[22] Andrei Pangin。2024。async-profiler：面向 Java 的低开销采样分析器。https://github.com/async-profiler/async-profiler。[23] Parca 作者。2024。Parca：用于分析 CPU 和内存使用量的持续性能分析。https://www.parca.dev/。[24] Tirthak Patel 和 Devesh Tiwari。2024。重新审视大规模机器学习研究集群的可靠性。载于 *IEEE/IFIP* 国际可靠系统与网络会议（*DSN*）论文集。IEEE，澳大利亚布里斯班，1–12。[25] Josh Poimboeuf。2023。ORC 展开器。Linux 内核文档，https: //docs.kernel.org/x86/orc-unwinder.html。[26] PyTorch 贡献者。2024。PyTorch Profiler。https://pytorch.org/tutorials/ recipes/recipes/profiler_recipe.html。[27] Gang Ren、Eric Tune、Tipp Moseley、Yixin Shi、Silvius Rus 和 Robert Hundt。2010。Google 全域性能剖析：面向数据中心的持续性能剖析基础设施。*IEEEMicro* 30，4（2010），65–79。

[28] Xinkai Wang、Xiaofeng Hou、Chao Li、Yuancheng Li、Du Liu、Guoyao Xu、Guodong Yang、Liping Zhang、Yuemin Wu、Xiaopeng Yuan、Quan Chen 和 Minyi Guo。2025。EXIST：在数据中心实现极高效的服务内追踪可观测性。载于第 *30* 届美国计算机协会编程语言与操作系统架构支持国际会议论文集（*ASPLOS*）。美国计算机协会。doi:10.1145/3676641.3716283[29] Qidong Zhao、Hao Wu、Yuming Hao、Zilingfeng Ye、Jiajia Li、Xu Liu 和 Keren Zhou。2025。DeepContext：一种用于深度学习工作负载性能剖析与分析的上下文感知、跨平台、跨框架工具。载于第 *30* 届美国计算机协会编程语言与操作系统架构支持国际会议论文集（*ASPLOS*）。美国计算机协会。[30] Xu Zhao、Kirk Rodrigues、Yu Luo、Ding Yuan 和 Michael Stumm。2016。基于流重建原理，对整个软件栈进行非侵入式性能剖析。载于第 *12* 届 *USENIX* 操作系统设计与实现研讨会论文集（*OSDI*）。USENIX 协会，603–618。[31] Xu Zhao、Yongle Zhang、David Lion、Muhammad Faizan Ullah、Yu Luo、Ding Yuan 和 Michael Stumm。2014。lprof：一种用于分布式系统的非侵入式请求流分析器。载于第 *11* 届 *USENIX* 操作系统设计与实现研讨会论文集（*OSDI*）。USENIX 协会。[32] 知乎 AI。2024。MegaScan：面向大规模训练的 Python 级性能剖析。https://github.com/zhihu/megascan。针对 Megatron 的 Python 追踪。[33] Fang Zhou、Yifan Gan、Sixiang Ma 和 Yang Wang。2018。wPerf：用于识别瓶颈等待事件的通用离 CPU 分析。载于第 *13* 届 *USENIX* 操作系统设计与实现研讨会论文集（*OSDI*）。USENIX 协会，527–543。[34] Yang Zhou 等。2024。Strobelight：Meta 的轻量且可扩展的持续性能剖析。载于*USENIX* 年度技术会议论文集（*ATC*）。USENIX 协会。