通过系统调用可观测性诊断应用程序的输入/输出行为

Tˆania Esteves Ricardo Macedo Rui Oliveira Jo˜ao Paulo*INESC TEC & U. Minho INESC TEC & U. Minho INESCTEC& U. Minho INESCTEC&U.Minho*tania.c.araujo@inesctec.pt ricardo.g.macedo@inesctec.pt rui.oliveira@inesctec.pt jtpaulo@inesctec.pt

arXiv:2304.08569v1 [cs.DC] 17 Apr 2023

摘要—**我们介绍 DIO，这是一款通用工具，可用于观察应用程序与内核存储系统之间低效且错误的输入/输出交互，这些交互会导致性能、可依赖性和正确性问题。DIO 能够分析并近实时可视化数据密集型应用程序的复杂输入/输出模式，此类应用程序会生成数百万个存储请求。为此，DIO 以非侵入方式拦截系统调用，为收集的数据补充相关上下文，并及时分析和可视化跟踪事件。**

我们通过分析两个生产级应用程序来展示其效用。结果表明，DIO 能够诊断多线程输入/输出中的资源争用（这会导致较高的尾延迟），以及导致数据丢失的错误文件访问。

索引词—**存储系统、输入/输出诊断、跟踪、分析**

I. 引言

数据密集型应用程序（例如数据库、键值存储、分析引擎和机器学习框架）的性能、正确性和可依赖性，深受其与内核中的 POSIX 存储后端（如文件系统和块设备）交互方式的影响 [1], [2]。

由于人为错误以及缺乏关于如何高效且正确地访问存储后端的详细知识，开发人员经常实现出具有以下问题的应用程序：*i)* 代价高昂的访问模式，例如小尺寸输入/输出请求或随机访问；*ii)* 并发请求访问共享存储资源所导致的输入/输出争用；以及*iii)* 错误使用输入/输出调用，例如访问了错误的文件偏移量。这些模式会导致低效或不正确的存储 I/O 访问，不仅会削弱各存储后端中实现的优化（例如缓存和调度）的效用，还可能最终降低端到端性能、损害应用程序的可用性，甚至造成数据丢失。

这些应用程序产生的存储操作数量庞大，每秒可达数百至数千次，因此手动分析这些操作既复杂又耗时。因此，能够帮助用户和开发者更精准地分析应用程序与相应存储后端之间输入/输出交互的诊断工具，对于调试错误、发现性能和可依赖性问题，以及识别应用程序的潜在优化至关重要 [3]–[5]。

本文的主要见解是，将系统调用（简称系统调用）跟踪与可定制分析流水线相结合，可以为使用内核中 POSIX 存储系统（例如文件系统、Linux 块设备）的应用程序提供非侵入式且全面的输入/输出诊断。为此，需要克服现有方法的以下局限。

侵入性。收集输入/输出请求的信息通常通过源代码插桩来完成 [6]–[9]。这种方法不容易应用于不同的应用程序，因为它要求用户手动分析并插桩各个规模可能很大的代码库。

实用性。 可以通过内核级跟踪技术以非侵入方式拦截输入/输出请求。然而，广泛使用的解决方案（例如 strace [10],）会给应用程序带来性能损耗，因此对于数据密集型工作负载而言并不实用。具体来说，它会显著增加跟踪请求所需的时间，而且由于性能下降，可能掩盖细微的并发问题，例如输入/输出争用或饥饿 [4], [11]。这一挑战促使 eBPF [12]和 LTTng [13],等技术应运而生；这些技术采用非阻塞跟踪策略，以丢弃可能无法及时处理的输入/输出事件为代价，降低性能开销。

缺乏分析流水线。虽然高效的输入/输出跟踪是应用程序性能分析的重要一步，但仅靠它并不足够，因为需要解析、关联并以可视化方式呈现大量采集的事件（轻易可达数千万），才能提供有价值的信息（例如，展示多线程 I/O 中的竞争）。有些解决方案只涵盖跟踪采集这一步，将其他耗时任务交由用户完成 [10], [14]–[16]。

灵活性。提供完整应用程序诊断流水线的解决方案，通常针对严格限定的分析场景设计，例如检测不可复现构建 [17], 、观察文件偏移访问模式 [5], 或识别安全问题 [18], [19]。因此，对于多用途的性能分析任务，必须组合使用多种此类工具，并多次重复对同一应用程序进行跟踪、分析和可视化。理想情况下，诊断工具应能根据用户目标，灵活缩小或扩大跟踪和分析的范围。这样，使用同一工具便可探索应用程序可能呈现的更广泛的性能、正确性和可依赖性问题，例如 §III 中所识别的问题。

本文提出 DIO，这是一种用于观测和诊断应用程序存储 I/O 的通用工具，能够解决这些

挑战，具体贡献如下。

非侵入式、全面且灵活的跟踪。DIO 提供了一种新型基于 eBPF 的跟踪器，可拦截应用程序发出的系统调用，无需修改其源代码或对二进制文件进行插桩。该跟踪器支持 42 种存储相关系统调用，并为每个采集到的操作记录全面的信息，包括其类型、参数、返回值、时间戳、进程 ID（PID）和线程 ID（TID）。DIO 采用灵活的设计，允许仅采集感兴趣的事件，并在内核空间按系统调用类型、PID、TID 或文件路径对事件进行过滤。这使用户能够根据需求缩小跟踪范围，并最大限度地降低对目标应用程序的性能开销。

增强分析。DIO 会为每个系统调用采集的数据补充内核中可获取的上下文信息（例如，进程名称、文件类型、文件偏移量），以便改进对请求的关联和分析（例如，将不同的系统调用关联到某个文件路径，区分针对普通文件或目录的操作）。这些功能支持对错误或低效的输入/输出模式进行更丰富、更全面的分析。

异步事件处理。 只有系统调用拦截是同步执行的，而采集的事件则会异步发送并存储到远程后端。这避免了在输入/输出请求的关键路径中增加额外延迟，并支持对耗时长、数据密集型存储工作负载进行实用分析。

近实时流水线。 DIO 提供了一条实用且可定制的流水线，用户可以创建自己的查询、关联算法和可视化仪表板，以分析采集的数据。该流水线采用内联方式，这意味着跟踪到的信息一经捕获，就会自动解析并转发至分析和可视化组件，无需用户手动干预。

DIO 使用 eBPF [12], 、Elasticsearch [20], 和 Kibana [21], 实现为开源原型，并在生产级系统上进行了验证。结果表明，DIO 能够诊断*i)* 导致 Fluent Bit 数据丢失的错误文件访问，以及*ii)* 多线程输入/输出中的资源争用；后者会导致 RocksDB 用户工作负载出现较高的尾延迟。本文讨论的所有工件，包括 DIO、工作负载、脚本以及相应的分析和

可视化输出已公开提供。[1]

II. DIO 工具

DIO 是一种用于观察和诊断应用程序与内核 POSIX 存储系统之间 I/O 交互的通用工具。其设计建立在以下核心原则之上，这些原则应对了 §I 中讨论的挑战。

透明性。DIO 依赖 Linux 内核跟踪基础设施，即跟踪点和内核探针，在无需修改应用程序的源代码或底层库的情况下拦截其系统调用。

实用且及时的分析。跟踪数据会异步发送到远程分析流水线，避免在应用程序的关键

[1]https://github .com/dsrhaslab/dio

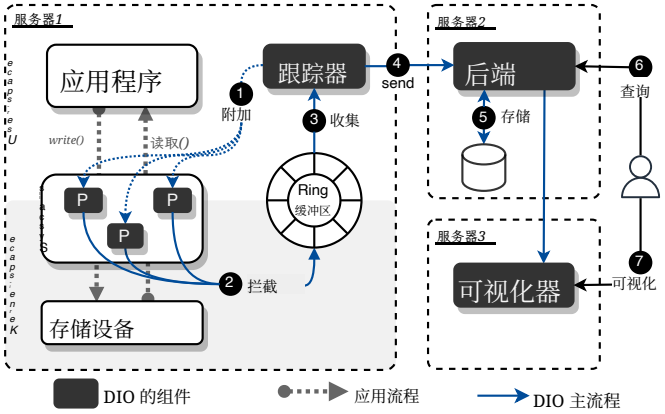

图 1： DIO 的设计和事件流程。

输入/输出路径上增加额外延迟，同时让用户能够近乎实时地直观查看收集到的数据。

事后分析。 DIO 允许存储来自同一或不同应用程序的多次跟踪执行记录，并在之后对其进行分析和比较。

灵活且全面的跟踪。 DIO 会拦截不同类型的存储相关系统调用，涵盖数据（例如，写入）、元数据（例如，文件状态）、扩展属性（例如，获取扩展属性）以及目录管理（例如，创建特殊文件）请求。用户可以选择仅捕获与其分析目标相关的系统调用，并进一步根据目标进程 ID、线程标识符或文件路径进行筛选。

丰富系统调用分析。 DIO 会将每个系统调用直接提供的信息（即类型、参数、返回值）与来自内核的其他上下文信息相结合，例如发起请求的进程名称、所访问文件的类型及其偏移量。

数据查询与关联。借助 DIO，用户可以查询跟踪到的数据、应用过滤器以分析特定信息（例如，特定线程 ID 执行的系统调用），并关联不同类型的数据（例如，将文件描述符与文件路径关联起来）。

定制化可视化。DIO 包含一个可视化组件，提供简化跟踪数据探索并构建定制化可视化图表的机制。

A. 系统概述

如图 1 所示，DIO 由三个主要组件构成，即跟踪器、后端和可视化器。DIO 的分析流水线包含后两个组件。

跟踪器（§II-B）拦截应用程序发出的系统调用，根据用户的配置（例如，按线程 ID）对其进行过滤，并将相关信息打包成事件，异步发送至后端（ ）。后端（§II-C）持久化并索引事件（ ），并允许用户查询和汇总（例如，聚合）已存储的信息（ ）。

可视化器 (§II-D) 通过查询后端 (7) 近乎实时地呈现跟踪事件。用户依靠可视化器简化数据探索和

表 I： *DIO* 支持的系统调用。

Type	系统调用
Data	读取、预定位读取64、向量读取、写入、预定位写入64 向量写入，文件同步，文件数据同步、预读
元数据	创建文件、打开、相对目录打开、关闭 文件定位、截断、文件截断 重命名、相对目录重命名、相对目录重命名2 取消链接、取消链接（相对目录文件描述符）、读取符号链接、读取符号链接（相对目录文件描述符） stat，符号链接状态，文件描述符状态，文件系统状态，相对目录文件描述符的文件状态
扩展属性	获取扩展属性、获取符号链接扩展属性、获取文件描述符扩展属性 设置扩展属性，设置扩展属性（不跟随符号链接），设置文件扩展属性 列出扩展属性，列出扩展属性（不跟随符号链接），列出文件扩展属性 移除扩展属性，移除扩展属性（不跟随符号链接），移除文件扩展属性
目录	创建特殊文件，基于目录文 件描述符创建特殊文件

分析过程，可通过选择特定类型的数据（例如，系统调用类型、参数）来构建不同的自定义表示形式。

B. 跟踪器

该跟踪器 以非侵入的方式拦截应用程序执行的系统调用。为此，它依赖 eBPF 技术 [12],，通过在每当到达指定的关注点（例如，跟踪点、内核探针）时执行小型程序（即 eBPF 程序），来对 Linux 内核进行插桩。

具体而言，DIO 的跟踪器 由一组 eBPF 程序组成，这些程序会在初始化阶段（ ）自动且●明地附加到系统调用跟踪点。每当到达这些跟踪点时（即，调用系统调用时），eBPF 程序便会收集有关请求的所需信息，并将其放入每 CPU 的环形缓冲区 （ ）中。环形缓冲区是一块连续的内存区域，用于在内核（生产者）与用户空间（消费者）进程之间交换数据。在用户空间，跟踪器 会异步地从环形缓冲区（ ）中获取信息，将其解析为事件（以 JSON 对象指定），并发送到●端。为尽量降低网络和性能开销，跟踪器会将多个事件分组成批次，再分批发送到后端并建立索引。

表 I 展示了 DIO 支持的系统调用。由于对系统调用进行插桩可能会在输入/输出请求的关键路径中引入额外处理，DIO 允许用户按以下条件过滤请求：*i)* 类型、*ii)* 进程或线程 ID，以及 *iii)* 目标文件或目录路径。通过在内核中实现这些过滤器，DIO 减少了发送到用户空间的信息量。

收集的信息。对于每个被拦截的系统调用，DIO 都会收集以下相关信息：*i)* 请求（类型、参数和返回值）；*ii)* 进程（进程 ID、线程 ID 和进程名称）；以及*iii)* 时间（进入和退出时间戳）。仅凭这些信息，就能深入了解应用程序的输入/输出行为（例如，随时间发出的系统调用、输入/输出请求的大小）；若将其与其他类型的数据关联起来，则能进一步丰富信息并简化用户的分析（详见第 III 节）。因此，跟踪器利用 eBPF 对内核结构的访问能力，为跟踪到的信息补充：

- 系统调用所针对的文件类型，从而能够区分对普通文件、目录、套接字、块设备/字符设备、管道、符号链接及其他文件的访问。
- 数据相关系统调用所访问的 文件偏移量 。即使对于不将文件偏移量作为参数提供的系统调用（例如，读取、写入），这些信息也有助于观察文件访问模式（例如，随机访问）。
- 一个文件标签，用于标记处理文件描述符的系统调用（例如，读取、关闭）。该标签包含设备号、inode 编号和首次文件访问时间戳，能够唯一标识正在访问的文件。

C. 后端

后端 支持持久化、搜索和分析跟踪事件中的数据。它使用 Elasticsearch [20] 分布式引擎来存储和处理大量数据。其灵活的面向文档的模式允许将事件编入索引并存储为文档，即使这些事件的结构可能各不相同（例如，对应系统调用参数的字段不同）。此外，它还提供用于搜索、查询和更新文档的接口，使用户能够开发并集成定制的数据关联算法。

文件路径关联算法。 我们实现了一种自定义算法，以便将系统调用与其访问的特定文件路径关联起来。利用 Elasticsearch 的数据查询和更新功能，与系统调用相关联的文件标签（即由跟踪器 组件生成的唯一标识符）会被转换为存储后端中实际访问的文件路径（例如，/tmp/fileA）。

D. 可视化器

可视化器 提供了一种自动化方式，用于探索（例如，查询和筛选事件）并直观呈现（例如，通过表格、直方图和时间序列图）分析结果。该组件使用 Kibana [21]，（面向 Elasticsearch 的数据可视化仪表板软件），它常用于日志和时间序列分析以及应用程序监控。Kibana 还支持构建自定义可视化图表，因此符合 DIO 的设计原则。

E. 实现

跟踪器由 ≈8K 代码行实现，分为两部分：*i)* 在内核空间运行的 eBPF 程序，以及*ii)*其余跟踪器逻辑所在的用户空间代码。

eBPF 程序使用 C 语言实现，负责收集和过滤相关的存储 I/O 事件。用户空间代码使用 Go（v17.4）实现，负责启用所需的 I/O 跟踪点（附加 eBPF 程序）、指定应用于各跟踪点的用户定义的过滤器、收集并解析在内核空间收集的信息，并将其发送到后端 组件。这是通过 BPF 编译器集合（BCC）框架，借助*gobpf* 库（v0.2.0）实现的。该库提供了创建、附加和交互 eBPF 程序的抽象。与 Elasticsearch 通信时，我们使用

go-elasticsearch （v7.13.1）模块，利用批量索引 API 同时发送多个事件。

后端 和可视化器 组件分别使用 Elasticsearch（v8.5.2）和 Kibana（v8.5.2）。文件路径关联算法可由跟踪器自动执行，也可由用户按需执行。

F. 配置与使用

DIO 的安装和配置分为两个阶段：*i*）设置并初始化分析流水线，以及*ii*）配置并运行跟踪器。**分析流水线**。虽然 DIO 的所有组件都可以部署在同一台机器上，但为避免对目标应用程序的性能造成负面影响（例如增加资源消耗），可以将分析流水线安装在单独的服务器上（图 1）。此外，由于跟踪器 组件会为每次跟踪执行标记唯一的会话名称，因此可以将 DIO 部署为一项服务，在专用服务器上设置分析流水线，并允许不同用户在不同机器上多次运行 DIO 的跟踪器 。

分析流水线的部署和配置包括安装其软件（即 Elasticsearch 和 Kibana）并导入预定义的仪表板。跟踪数据一旦到达流水线，用户便可访问 Kibana 网页，查看 DIO 的仪表板、应用分析筛选器，并编辑或创建新的可视化图表和仪表板。

跟踪器。 部署分析流水线后，用户可以使用 DIO 的跟踪器 收集信息。跟踪器 与目标应用程序一同运行，并在其主进程和子进程结束或用户明确指示时停止。

默认情况下，DIO 的跟踪器 会为所有受支持的系统调用启用跟踪点。不过，用户可以指定要观测的系统调用列表，跟踪器 便只会为这些操作启用跟踪点。此外，用户还可以指定要观测的文件/目录列表，指示跟踪器 仅记录针对这些对象的事件。所有这些配置以及分析流水线的参数（例如，Elasticsearch URL）均可通过配置文件设置。

III. 评估

我们的评估展示了如何使用 DIO 来简化对已知问题的观测和确认，以及对其修复进行验证。为此，我们分析了两个生产级应用程序：Fluent Bit 和 RocksDB。结果表明，DIO 是一种实用工具，无需对大型代码库进行插桩，即可验证正确性（§III-B）和性能（§III-C）问题的根本原因。

除图3外，本节中的所有其他图均由 DIO 生成（仅为便于阅读做了少量修改）。DIO 的完整可视化表示集可在 https://github.com/dsrhaslab/dio 获取。

A. 实验设置

我们的测试环境由三台服务器组成，运行 *Ubuntu 20.04 LTS* ，内核版本为*5.4.0*。运行应用程序和 DIO跟踪器 的服务器配备了 4 核 Intel Core i3-7100，16 GiB 内存、一块 250 GiB NVMe 固态硬盘（用于存放数据集）以及一块 512 GiB SATA 固态硬盘（用于记录日志）。DIO后端 和可视化 组件分别运行在另外两台服务器上，两台服务器均配备了 6 核 Intel i5-9500、16 GiB 内存和一块 250 GiB NVMe 固态硬盘。

B. 识别导致数据丢失的错误操作

DIO 可帮助开发人员和用户诊断其应用程序的正确性。我们通过展示会导致数据丢失的错误 I/O 访问模式来说明这一点。

对于此用例，我们考虑 Fluent Bit [22] (v1.4.0)，这是一款高性能日志与指标处理器和转发器。已有问题报告[2,3] 指出，使用 tail 输入插件时会丢失数据；该插件用于获取添加到日志文件中的新内容。因此，我们实现了一个客户端程序，用于模拟日志文件的生成过程，以供 Fluent Bit 处理，并模仿问题 #1875[2] 中报告的 I/O 行为。我们使用 DIO 同时跟踪并分析客户端程序和 Fluent Bit，方法是过滤属于这些应用程序进程集合的系统调用。

图 2a 展示了由 DIO 生成的表格可视化。客户端程序（应用程序）首先创建 app.log 文件，从偏移量 0 开始写入 26 字节，然后关闭文件（ ）。接着，Fluent Bit（*fluent-bit*）检测到文件内容发生变化，打开该文件，并从偏移量 0 读取 26 字节，这意味着 *fluent-bit* 处理了此前由应用程序 写入的全部内容（ ）。之后，应用程序 通过取消链接系统调用移除该文件，而 *fluent-bit* 则关闭相应的文件描述符（ ）。在操作系统层面，这意味着与该文件关联的 inode 编号（12）现已不再使用，之后会分配给新文件。然而，一种可能的情况是，这个 inode 编号被映射到一个名称相同的新创建文件上。当应用程序 创建一个与之前同名（app.log）的新文件并向其中写入 16 字节时，就会发生这种情况（ ）。问题报告中指出、且可通过 DIO 观察到的错误行为是：*fluent-bit* 打开新的日志文件以读取其内容时，没有像预期那样从偏移量 0 开始读取，而是从偏移量 26 开始读取（ ）。由于起始偏移量错误，读取系统调用返回 0 字节，应用程序 写入的 16 字节因此丢失。

为了解这一行为背后的原因，我们检查了 Fluent Bit 中负责读取日志文件新增内容的代码。在读取文件之前，Fluent Bit 会将文件位置更新为已处理的字节数。该值会为每个受跟踪的文件保存在数据库中，并通过文件名和 inode 编号来识别该文件。错误的是，文件从文件系统中删除时，数据库条目并未被删除。因此，回到我们的示例，由于新创建的文件被赋予了相同的文件名（app.log）和 inode 编号（12），*fluent-bit* 便错误地认为，后一个日志文件开头的 26 个字节已经处理过。

为了验证对这一错误访问模式的修正，我们使用 DIO 分析了更新版本的 Fluent Bit

[2]https://github.com/fluent/fluent-bit/issues/1875
[3]https://github.com/fluent/fluent-bit/issues/4895

time	proc_name	syscall	ret val	file_tag (dev_no|inode_no|timestamp)	offset	
1,679,308,382,363,981,568	app	openat	3	7340032|12|2156997363734041	-	
1,679,308,382,364,387,584	app	write	26	7340032|12|2156997363734041	0	1
1,679,308,382,364,442,624	app	close	0	7340032|12|2156997363734041	-	
1,679,308,386,884,300,800	fluent-bit	openat	23	7340032|12|2156997363734041	-	
1,679,308,386,889,688,320	fluent-bit	read	26	7340032|12|2156997363734041	0	2
1,679,308,386,892,196,096	fluent-bit	read	0	7340032|12|2156997363734041	26	
1,679,308,392,364,854,016	app	unlink	0	-	-	3
1,679,308,392,365,804,032	fluent-bit	close	0	7340032|12|2156997363734041	-	
1,679,308,402,365,455,104	app	openat	3	7340032|12|2157017365367381	-	
1,679,308,402,365,598,976	app	write	16	7340032|12|2157017365367381	0	4
1,679,308,402,365,668,864	app	close	0	7340032|12|2157017365367381	-	
1,679,308,406,884,280,320	fluent-bit	openat	23	7340032|12|2157017365367381	-	
1,679,308,406,884,805,120	fluent-bit	lseek	26	7340032|12|2157017365367381	26	
1,679,308,406,885,053,440	fluent-bit	read	0	7340032|12|2157017365367381	26	5
1,679,308,422,386,589,952	fluent-bit	close	0	7340032|12|2157017365367381	-	

(a) *Fluent Bit*（*v1.4.0*）的错误访问模式。

time	proc_name	syscall	ret val	file_tag (dev_no|inode_no|timestamp)	offset	
1,679,248,356,503,484,160	app	openat	3	7340032|12|2096971503238627	-	
1,679,248,356,503,664,128	app	write	26	7340032|12|2096971503238627	0	1
1,679,248,356,503,719,680	app	close	0	7340032|12|2096971503238627	-	
1,679,248,361,001,024,256	flb-pipeline	openat	46	7340032|12|2096971503238627	-	
1,679,248,361,007,723,776	flb-pipeline	read	26	7340032|12|2096971503238627	0	2
1,679,248,361,008,218,112	flb-pipeline	read	0	7340032|12|2096971503238627	26	
1,679,248,366,503,962,624	app	unlink	0	-	-	3
1,679,248,366,506,702,336	flb-pipeline	close	0	7340032|12|2096971503238627	-	
1,679,248,376,505,657,344	app	openat	3	7340032|12|2096991505568257	-	
1,679,248,376,505,789,184	app	write	16	7340032|12|2096991505568257	0	4
1,679,248,376,505,878,272	app	close	0	7340032|12|2096991505568257	-	
1,679,248,381,000,811,264	flb-pipeline	openat	46	7340032|12|2096991505568257	-	
1,679,248,381,001,634,304	flb-pipeline	read	16	7340032|12|2096991505568257	0	
1,679,248,381,001,834,496	flb-pipeline	read	0	7340032|12|2096991505568257	16	5
1,679,248,381,002,218,240	flb-pipeline	read	0	7340032|12|2096991505568257	16	
1,679,248,397,000,544,000	flb-pipeline	close	0	7340032|12|2096991505568257	-	

(b) *Fluent Bit*（*v2.0.5*）的正确访问模式。

图 2： *Fluent Bit* 导致数据丢失的错误访问模式。

（v2.0.5），该版本已应用修复，以避免这一数据丢失问题。图 2b 展示了修复版本的类似表格可视化。虽然两个版本表现出相似的行为（对 - 的文件访问相同），但区别在于 Fluent Bit（*flb-pipeline*）读取新文件（ ）时访问的文件偏移量。这一次，Fluent Bit 从文件开头（偏移量 0）开始读取，因此能够读取*app*写入的新 16 个字节。

此示例表明，DIO 可帮助用户诊断应用程序的不正确输入/输出行为，并找出数据丢失等可依赖性问题的根本原因。此外，虽然此示例仅展示了少量数据丢失，但处理较大的日志文件时，数据丢失量可能会显著增加。此外，此用例还展示了 DIO 如何帮助验证应用程序实现中所做的修正。

C. 查找性能异常的根本原因

现在，我们将展示 DIO 如何通过找出嵌入式键值存储（KVS）RocksDB 的客户端请求出现高尾延迟的根本原因，帮助诊断性能问题 [23]。

这一现象最早在 SILK [24] 中被观察到，因此我们沿用相同的测试方法将其复现。我们使用*db bench* 基准测试 [25]，配置 8 个客户端线程，以闭环方式执行读写混合请求（YCSB A [26]）。RocksDB 配置了 8 个后台线程，其中 1 个用于刷新操作，7 个用于压实操作。图3展示了一次持续 5 小时的运行样本，并描绘了客户端经历的 99th 百分位延迟。在整个样本期间，客户端观察到多次延迟尖峰，范围为 1.5 毫秒至 3.5 毫秒。

要通过在 RocksDB代码库中插桩来找出这种性能损失的根本原因，需要检查超过 44 万代码行，并向多个核心组件添加调试代码。另一种方法是使用 DIO，轻松地

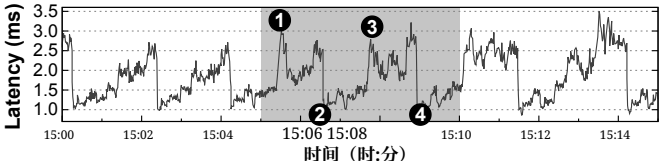

图3： 99th 百分位的 *RocksDB* 客户端操作延迟。

跟踪、分析并可视化 RocksDB 的执行情况，如图4所示。由于工作负载以数据为中心，我们将 DIO 的 跟踪器 配置为仅捕获打开、读取、写入和关闭系统调用。客户端线程表示为 db_bench，而 rocksdb:high0 对应刷新线程，其余线程（rocksdb:lowX）对应压实线程。

通过观察 RocksDB 不同线程随时间提交的系统调用，可以识别性能竞争。具体而言，如红色高亮框所示，当多个压实线程提交输入/输出请求时，db_bench 线程的系统调用数量会减少，导致客户端感知到尾延迟立即飙升，如图3和图4所示（在区间 和 中，至少有 5 个压实线程提交请求）。当执行输入/输出操作的压实线程较少时，db_bench 线程的性能会有所提升，尾延迟和吞吐量均得到改善（在区间 和 中，只有 1 到 2 个压实线程在执行输入/输出操作）。

如果结合对日志结构合并树（LSM）键值存储的工作方式的了解来补充前面的观察，问题就很清楚了：RocksDB 使用前台线程处理客户端请求（db_bench 线程），这些请求会排入队列，并按先进先出顺序处理。与此同时，后台线程负责内部操作，即刷新操作（rocksdb:high0）和压实操作（rocksdb:lowX）。刷新操作可确保内存中的键值对按顺序写入持久化LSM树的第一级（L_0）；只有在L_0有足够空间时才能进行这些操作。压实操作会进入先进先出队列，等待

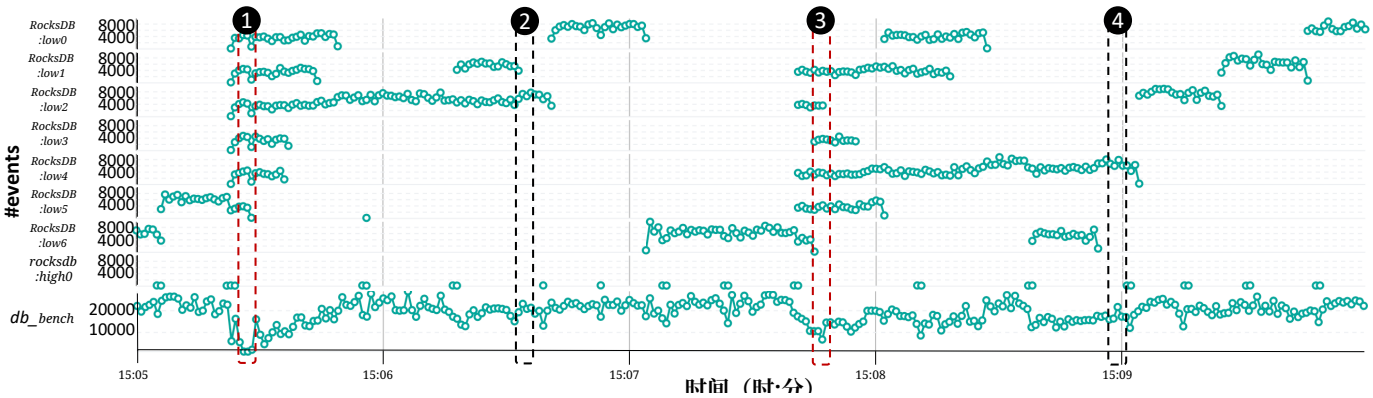

图4： *RocksDB* 随时间发出的系统调用，按线程名称汇总。*db_bench* 包含 *8* 个客户端线程，*rocksdb:low*[0-6] 指各个压实线程，而 *rocksdb:high0* 指刷新线程。

表 II： *RocksDB 3* 次独立运行的平均执行时间和标准差。

	原生版本	*Sysdig*	*DIO*	*strace*
平均执行时间	03h48m	03小时56分钟	05小时12分钟	06小时30分钟
标准差	±1.2分钟	±1.7分钟	±2.4m	±4.6m
开销		1.04x	1.37	1.71x

由专用线程池执行。除低级压实操作（$L_0 \to L_1$）外，这些操作可以并行进行。然而，压实操作的一个常见问题是输入/输出工作流之间会相互干扰，从而导致客户端请求出现延迟尖峰。具体来说，当客户端线程无法继续执行时，就会出现延迟尖峰，因为 $L_0 \to L_1$ 压实操作和刷新操作进展缓慢或被暂停；例如，多个线程争用共享的磁盘带宽（造成竞争）时，就会发生这种情况。这正是 SILK 所指出的现象，它可能会对键值存储及使用它们的服务的响应时间乃至可用性造成负面影响 [27], [28]，而且无需任何代码插桩，使用 DIO 即可观察到这一现象。

D. 性能影响与 *I/O* 事件处理

为了解在拦截输入/输出系统调用时，DIO 所带来的性能影响，我们选取了包含一项基准测试的 RocksDB 用例，并测量了三次独立运行的平均执行时间。

性能分析。表 II 比较了原生版本部署（即，不跟踪其执行情况）与 DIO、*strace* [10]——一种广泛使用的系统调用跟踪器，以及 *Sysdig* [14] ——一种最先进的基于 eBPF 的系统调用跟踪器。

执行原生版本 配置大约需要 3 小时 48 分钟。与原生版本相比，*Sysdig* 带来的开销最小（1.04 倍），执行时间仅多出 8 分钟；而*strace* 带来的开销最大（1.71 倍），执行时间多出 2 小时 42 分钟。与*Sysdig*相比，DIO 会使性能开销增加 1.37 倍，需要额外 1 小时 16 分钟；但与 *strace*相比，可节省最多 1 小时 18 分钟。

strace 与两种基于 eBPF 的跟踪器（即*sysdig* 和 DIO）之间的差异，可以用其底层跟踪技术来解释。*strace*用于拦截系统调用的陷阱机制以及上下文切换，会给目标应用程序带来可观的开销 [11]。

在基于 eBPF 的跟踪器中，*Sysdig* 的性能开销低于 DIO，但向用户提供的信息也较少。具体而言，DIO 最多无法报告 5% 的采集事件的文件路径（由于事件被丢弃），而*Sysdig* 则有 45% 的事件无法报告文件路径。正如预期，向用户提供更多信息会加重应用程序执行时间方面的性能损耗。

I/O 事件处理。如 §II 所述，DIO 使用固定大小的环形缓冲区在用户空间采集信息；在这些实验中，该缓冲区的配置为每个 CPU 核心 256 MiB。当缓冲区已满时（即，如果内核进程向环形缓冲区写入 I/O 事件的速度快于用户空间进程的读取速度），在内核级拦截到的新 I/O 事件将被丢弃。

对于 RocksDB 实验，鉴于其输入/输出行为十分密集，发出的系统调用中有 3.5%（549M 中的 ≈19M）在环形缓冲区中被丢弃，因此未存储到 DIO 的后端中。

E. 总结

前述用例展示了 DIO 诊断不同输入/输出模式的能力，使用户能够观察应用程序的输入/输出行为、确认已知问题，并验证修复是否有效。此外，我们集成的跟踪和分析流水线使用户无需进行代码插桩，也无需手动组合多个工具，即可观察这些输入/输出模式。

初步实验结果表明，DIO 能够收集、解析并将所有必需的跟踪信息转发至分析流水线，同时降低性能开销。此外，尽管 RocksDB 中有一些输入/输出事件被丢弃，我们仍证明 DIO 能够定位资源争用并帮助诊断其根本原因。此外，与*strace* 和*sysdig*不同，DIO 跟踪的信息会

在被拦截并传输到后端组件后立即可供可视化。

IV. 相关工作

输入/输出跟踪。 存储 I/O 诊断通常通过源代码插桩 [6]–[9] 在用户空间捕获应用程序的请求；也可以通过中间件库 [29], [30]来实现，但这类库仅适用于特定应用程序集（例如，*LD PRELOAD* 仅适用于动态库）；还可以在更底层的内核层进行 [5], [19], [30], ，例如虚拟文件系统。在这些层中，输入/输出合并等优化会导致无法观察应用程序提交的确切请求。

为了以非侵入方式拦截输入/输出操作，并更贴近应用程序发出的请求，其他解决方案依赖系统调用接口。如表 III 所示，这些方案采用了不同的跟踪技术，包括 ptrace（ [10], [17]）、eBPF（[4], [14], [16]）、LTTng（[3], [15], [31]）和 auditd（[18]），这些技术可以收集与系统调用入口和出口相关的信息，包括其参数、返回值、时间戳、进程 ID等。与 DIO 类似，一些工具会用额外信息丰富跟踪数据，例如进程名称（[4], [14], [16], [18]），这有助于观察 §III-B 和 §III-C 中的输入/输出模式。不过，DIO 是唯一会收集文件偏移量的工具，而这对于诊断 §III-B 中介绍的用例至关重要。

只有 CaT [4],Tracee [16], 和 DIO 会将每个系统调用入口 和出口 处包含的信息汇总为单个事件，从而简化后续分析。此操作在内核空间中完成，以减少传输到用户空间的数据。此外，只有这些工具，以及 *strace* [10]和 *Sysdig* [14], 支持在跟踪阶段进行过滤。
集成分析流水线。 </style}有几种解决方案仅涵盖跟踪步骤，分析流水线的集成则留给用户完成 [10], [14]–[16]。其他工具提供用于自动分析跟踪数据的模块，但采用离线方式：数据需要先存储起来，之后再解析并作为输入提供给分析流水线 [3], [4], [17], [31]。只有 DIO 和 Longline [18]会采用内联（近实时）方式，自动解析跟踪事件并将其转发到分析流水线。

系统调用分析。 一些现有工具支持针对具体用例的专用分析模块（例如，因果关系 [4], [17],安全分析 [18]），这些模块只考虑从跟踪中收集的特定信息（例如，系统调用类型）。因此，这些工具既无法灵活地实现自定义分析算法，也无法让用户访问和探索所收集的输入/输出跟踪中包含的其他信息。另一方面，类似 DIO 且支持可定制分析的解决方案，未能捕获诊断本文所讨论用例所需的相关信息 [3], [31]。

DIO 让用户能够访问捕获信息的完整集合（例如，系统调用类型、参数、偏移量），从而针对与其分析目标更相关的数据字段构建新的算法。

表 III： *DIO* 与其他解决方案在以下方面的比较：捕获的跟踪信息、过滤能力、跟踪与分析集成（*O*-离线，*I*-内联）、分析定制以及预定义可视化支持。尽管有些工具能够跟踪（*T*）本文用例所需的信息，但只有DIO 为用户提供诊断这些用例所需的分析（*A*）能力。

Tracing	系统调用信息	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	*f*偏移量	-	-	-	-	-	-	-	-	-	✓
	*f*类型	-	✓	-	-	-	-	-	-	-	✓
	进程名称	-	✓	-	-	✓	✓	-	-	✓	✓
	过滤器	✓	✓	-	-	✓	✓	-	-	-	✓
Analysis pipeline	集成	-	-	-	O	-	O	O	O	I	I
	可自定义	-	-	-	-	-	-	✓	✓	-	✓
	预定义可视化	-	-	-	-	-	✓	✓	✓	✓	✓
Use cases	§III-C	-	T	-	-	T	T	-	-	T	TA
	§III-B	-	-	-	-	-	-	-	-	-	TA

系统调用可视化。DIO 提供预定义的表示形式，可自动汇总并实现对论文中讨论的输入/输出模式进行可视化。此外，我们的工具还支持用户创建其他诊断解决方案中常见的可视化图表（例如，表格、饼图、直方图、热力图、时间序列） [3], [18], [31]。

V. 未来方向

通过以非侵入式的方式拦截应用程序的输入/输出系统调用，并自动解析收集到的数据并将其转发至分析流水线，DIO 免去了用户理解应用程序源代码、为相关部分添加插桩以及手动解析所得数据所需的时间。

虽然当前原型已能提供有关目标应用程序的相关摘要信息，但我们认为可以进一步简化用户的分析过程，例如引入新的自动化关联算法。因此，作为未来的研究方向，我们计划探索 Elasticsearch 查询 API，并构建一套关联算法，以便快速识别前述应用程序中观察到的低效行为。

此外，我们计划拓展 DIO 的应用范围，以展示其探索其他应用程序的能力，即使用户并不熟悉这些应用程序；与此同时，还可能发现新的输入/输出模式，以及此类应用程序在性能、可依赖性、正确性和安全性方面尚未识别的问题。

最后，我们希望更详细地分析 DIO 对目标应用程序造成的性能开销，并研究新的优化方法，以尽量降低这一开销，同时减少在跟踪阶段被丢弃的输入/输出事件数量。

VI. 结论

本文介绍 DIO，这是一种通用工具，用于观测和诊断应用程序与内核中的

POSIX 存储系统之间的输入/输出交互。DIO 通过一条流水线，自动完成数百万次系统调用的跟踪、过滤、关联和可视化，并利用额外的上下文丰富这些系统调用所提供的信息，帮助用户观测输入/输出问题，同时缩小查找其根本原因的范围，例如在需要进行源代码检查时。

我们对两种广泛使用的系统进行的实验表明，DIO 提供了关键信息，可用于观察导致数据丢失的错误的 I/O 访问模式，以及识别导致尾延迟升高的多线程 I/O 中的资源争用。

参考文献

[1] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau 和 R. H. Arpaci-Dusseau，“冗余并不意味着容错：对分布式存储系统应对单一错误和数据损坏的分析”，载于第十五届 USENIX 文件与存储技术会议（FAST’17）。USENIX 协会，2017 年，第 149–166 页。 [在线]。获取地址：https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan [2]D. S. Roselli、J. R. Lorch 和 T. E. Anderson，“文件系统工作负载比较”，载于 USENIX 年度技术会议论文集（ATC’00）。USENIX 协会，2000 年。 [在线]。获取地址：https://www.usenix.org/legacy/event/usenix2000/general/fullpapers/roselli/roselli.pdf [3] H. Daoud 和 M. R. Dagenais，“基于内核和用户空间跟踪数据的分布式存储集群性能分析”，《软件：实践与经验》，第 51 卷，第 1 期，第 5–24 页，2021 年。 [在线]。获取地址：https://doi.org/10 1002 2889 . /spe. T. Esteves、F. Neves、R. [4] Oliveira 和 J. Paulo，“CAT：面向分布式系统的内容感知跟踪与分析”，载于第 22届国际中间件会议（Middleware’21）。ACM，2021 年，第 223–235 页。 [ ] 10 1145 3464298 3493396 在线。获取地址：https://doi.org/ . / . A. Saif、L. Nussbaum 和 Y.-Q. Song，“IOscope：用于工作负载 I/O 模式表征的灵活 I/O 跟踪器”，载于《高性能计算：ISC 高性能计算大会 2018 国际研讨会》。施普林格，2018 年，第 103–116 页。 [在线]。获取地址：https://doi.org/ 10 1007 978 3 030 02465 9 7 . / - - - - “Jaeger: [6] 开源端到端分布式跟踪”，2022 年。 [在线]。获取地址：https://www.jaegertracing.io [7] “Zipkin”，2022 年。 [在线]。获取地址：https://zipkin.io [8] S. J. Kim、S. W. Son、W.-k. Liao、M. Kandemir、R. Thakur 和 A. Choudhary，“IOPin：高性能计算系统中并行 I/O 的运行时性能分析”，载于《2012 年 SC 计算伴侣：高性能计算、网络、存储与分析》。IEEE，2012 年，第 18–23 页。 [在线]。获取地址：https://doi.org/10 1109 2012 14 . /SC.Companion. . [9] K. Vijayakumar、F. Mueller、X. Ma 和 P. C. Roth，“可扩展的 I/O 跟踪与分析”，载于第4届年度拍字节级数据存储研讨会。ACM，2009 年，第 26–31 页。 [在线]。获取地址：https: //doi.org/ 10 1145 1713072 1713080 . / . “strace: [10] Linux 系统调用跟踪器”。 [在线]。获取地址：https://strace.io [11] M. Gebai 和 M. R. Dagenais，“Linux 内核与用户空间跟踪器调查与分析：设计、实现和开销”，《ACM 计算调查》，第 51 卷，第 2 期，第 1–33 页，2018 年。 [在线]。获取地址：https://doi.org/ 10 1145 3158644 . / S. [12] McCanne 和 V. Jacobson，“BSD 分组过滤器：一种新的用户级分组捕获架构”，载于《1993 年冬季 USENIX 会议论文集》，第 46 卷。USENIX 协会，1993 年，第 259–269 页。 [在线]。获取地址：https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet [13] M. Desnoyers 和 M. R. Dagenais，“LTTng 跟踪器：一种对 GNU/Linux 影响较小的性能与行为监控工具”，载于渥太华 Linux 研讨会（OLS），第 2006 卷。Citeseer，2006 年，第 209–224 页。 [在线]。获取地址：https://www.kernel.org/doc/ols/2006/ols2006v1-pages-209-224.pdf [14] “Sysdig”，2022 年。 [在线]。获取地址：https://github.com/draios/sysdig/ [15] I. U. Akgun、G. Kuenning 和 E. Zadok，“Re-animator：通用高保真存储系统跟踪与重放”，载于第13届 ACM 国际系统与存储会议（SYSTOR’20）。ACM，2020 年，第 61–74 页。 [在线]。获取地址：https://doi.org/10.1145/3383669. 3398276 [16] “Tracee：使用 eBPF 实现 Linux 运行时安全与取证”，2022 年。[在线]。获取地址：https://github.com/aquasecurity/tracee [17] Z. Ren、C. Liu、X. Xiao、H. Jiang 和 T. Xie，“通过对系统调用跟踪进行因果分析，定位不可复现构建的根因”，载于第34届 IEEE/ACM 国际自动化软件工程会议（ASE’19）。IEEE，2019 年，第 527–538 页。 [在线]。获取地址：https://doi.org/10 1109 2019 00056 . /ASE. . S. Yoo、J. Jo、B. Ki [18] m 和 J. Seo，“LongLine：面向大规模审计日志的可视分析系统”，《Visual Informatics》，第 2, 1, 82 97, 卷，第 期，第 – 2018 [ ] 10 1016 2018 04 页。在线。获取地址：https://doi.org/ . /j.visinf. . . 009 [19] D. N. Jha、G. Lenton、J. Asker、D. Blundell 和 D. Wallom，“公有云环境中的整体运行时性能与安全感知监控”，载于第22届 IEEE 集群、云和互联网计算国际研讨会（CCGrid’22）。IEEE，2022 年，第 1052–1059 页。 [ ] 10 1109 54584 2022 00128 在线。获取地 [20] 址：https://doi.org/ . /CCGrid . . E. B.V.，“Elasticsearch：免费开放 Elastic Stack 的核心”，2022 年。 [在线]。获取地址：https://www.elastic.co/elasticsearch/ [21] ——，“Kibana：通往 Elastic Stack 的窗口”，2022 年。 [在线]。获取地址：https://www.elastic.co/kibana/ [22] “Fluent Bit：端到端可观测性流水线”，2022 年。 [在线]。获取地址：https://fluentbit.io [23] Facebook，“RocksDB：面向快速存储环境的持久化键值存储”，2022 年。 [在线]。获取地址：https://rocksdb.org [24] O. Balmau、F. Dinu、W. Zwaenepoel、K. Gupta、R. Chandhiramoorthi 和 D. Didona，“SILK：防止日志结构合并键值存储中的延迟尖峰”，载于 USENIX 年度技术会议（ATC’19）。USENIX 协会，2019 年，第 753–766 页。 [在线]。获取地址：https://www.usenix.org/conference/atc19/presentation/balmau [25] “db bench”，2022 年。 [在线]。获取地址：https://github.com/facebook/rocksdb/wiki/Benchmarking-tools [26] B. F. Cooper、A. Silberstein、E. Tam、R. Ramakrishnan 和 R. Sears，“使用 YCSB 对云服务系统进行基准测试”，载于第1届 ACM 云计算研讨会论文集（SoCC’10）。ACM，2010 年，第 143–154 页。 [在线]。获取地址：https: //doi.org/ 10 1145 1807128 1807152 . / . J. Dean [27] 和 L. A. Barroso，“规模化下的长尾”，《ACM 通讯》，第 56 卷，第 2 期，第 74–80 页，2013 年 2 月。 [在线]。获取地址：https://doi.org/10 1145 2408776 2408794 . / . J. Li、N. K. Sharma、D. R. K. [28] Ports 和 S. D. Gribble，“长尾故事：尾延迟的硬件、操作系统和应用程序层来源”，载于 ACM 云计算研讨会。ACM，2014 年，第 1–14 页。 [ ] 10 1145 2670979 2670988 在线。获取地址：https://doi.org/ . / . S. Snyder、P. Carns、K. Harms、R. Ross、G. K. Lockwood 和 N. J. Wright，“ 使用 Darshan 进行模块化高性能计算 I/O 特征分析 ”，载于第 5 届极大规模编程工具研讨会（ESPT@SC’16）。IEEE，2016 年，第 9–17 页。 [在线]。获取地址：https://doi.org/10.1109/ ESPT.2016.006 [30] M. I. Naas、F. Trahay、A. Colin、P. Olivier、S. Rubini、F. Singhoff 和 J. Boukhobza，“EZIOTracer：统一数据密集型应用程序的内核与用户空间 I/O 跟踪”，载于高效高性能存储系统的挑战与机遇研讨会论文集（CHEOPS ’21）。ACM，2021 年，第 1–11 页。 [在线]。获取地址：https://doi.org/ 10 1145 3439839 3458731 . / . I. [31] Kohyarnejadfard、D. Aloise、M. R. Dagenais 和 M. Shakeri，“一种利用跟踪数据分析检测系统性能异常的框架”，《Entropy》，第 23 卷，第 8 期，第 1011 页，2021 年。 [在线]。获取地址：https://doi.org/10.3390/e23081011