https://crad.ict.ac.cn
62(3): 633−647, 2025

面向 SCSI 子系统的用户空间存储架构设计

郝栋栋[1] 高聪明[1] 舒继武[1,2]
[1](厦门大学信息学院 福建厦门 361005)
[2](清华大学计算机科学与技术系 北京 100084)
(haodongdong@stu.xmu.edu.cn)

Design of User Space Storage Architecture for SCSI Subsystem

Hao Dongdong[1], Gao Congming[1], and Shu Jiwu[1,2]
[1](*School of Informatics*, *Xiamen University*, *Xiamen*, *Fujian* 361005)
[2](*Department of Computer Science and Technology*, *Tsinghua University*, *Beijing* 100084)

Abstract For the past few years, the storage industry has undergone tremendous changes. Semiconductor storage devices, like solid state drives (SSDs), have flourished and are able to completely outperform traditional hard disk drives (HDDs), addressing data by moving magnetic head. Nowadays, the mainstream protocols supporting SSDs are NVMe and SAS. NVMe is a high-performance storage protocol designed specifically for SSDs that can maximize the performance of SSDs; while the SAS protocol fully considers the requirements of data centers, providing high reliability and high scalability while considering the balance between system performance and cost. Compared with the increasingly fast storage media, the time overhead of the software stack designed for slow storage devices in an I/O process is becoming increasingly significant. To address this issue, numerous excellent works have been proposed by academia and industry. For example, Intel's SPDK (storage performance development kit) has greatly shortened the response time of NVMe SSD to applications by implementing device drivers in user space and polling I/O completion, extremely improving the performance of the entire system. However, previous research on the optimization of SAS SSD storage software stack is very limited. Therefore the SAS software stack optimization for SSD is implemented in user space. Experimental result shows that this optimization can effectively improve the data access efficiency with applications and storage devices. Besides, aiming to accurately evaluate the time cost of storage devices in I/O stack, a hardware performance testing tool HwPerfIO is proposed, which can eliminate the impact of most software overhead to measure the more accurate storage equipment performance.
Key words SSD; HwPerfIO; SAS protocol; I/O stack; user space driver

摘 要 近年来存储行业经历了巨大的变革,以固态硬盘(solid state drive, SSD)为代表的半导体存储设备迅猛发展,在性能上显著超越了通过磁头移动寻址的机械硬盘(hard disk drive, HDD).目前支持 SSD 的 2 种协议主要包括非易失性内存主机控制器接口规范(nonvolatile memory express, NVMe)协议与串行 SCSI(serial attached small computer system interface, SAS)协议,即 SAS. NVMe 是专为 SSD 设计的高性能存储协议,能够很大限度地发挥 SSD 的性能;而 SAS 协议则充分考虑数据中心的需求,在提供高可靠

收稿日期: 2024-07-21; **修回日期**: 2025-01-12
基金项目: 国家自然科学基金项目 (U22B2023); 国家自然科学基金青年科学基金项目 (62102219); 中央高校基本科研业务费专项资金 (20720230071)
This work was supported by the National Natural Science Foundation of China (U22B2023), the National Natural Science Foundation of China for Young Scientists (62102219), and the Fundamental Research Funds for the Central Universities (20720230071).
通信作者: 舒继武(shujw@tsinghua.edu.cn)

性与高可扩展性的同时,兼顾了系统性能与成本的平衡.相对于日益增速的存储介质,针对慢速存储设备所设计的软件栈在一次I/O过程中所耗费的时间开销愈发显著.针对该问题学界及工业界都相继提出了众多解决方案,例如Intel提出的高性能存储开发包(storage performance development kit, SPDK)通过将设备驱动实现在用户空间,并采用轮询感知I/O完成等方式大幅度缩短了NVMe SSD对应用程序的响应时间,极大地提升了整个系统的整体性能.然而之前的研究工作针对SAS SSD存储软件栈的优化非常有限,为此在用户空间实现了针对SAS SSD的软件栈优化.实验结果表明,该优化能够有效缩短存储设备对应用程序的响应时间,提高应用对存储设备的访存效率.此外,为了准确评估I/O栈中存储设备的时间开销,硬件性能测试工具HwPerfIO被提出,能够消除大部分软件开销的影响以测得更加准确的存储设备性能.

关键词 固态硬盘;HwPerfIO;SAS协议;I/O栈;用户空间驱动

中图法分类号 TP302.1

DOI: 10.7544/issn1000-1239.202440632 **CSTR:** 32373.14.issn1000-1239.202440632

NAND SSD[1]、持久性内存(persistent memory, PM)[2]等新兴存储介质的性能已经远超针对慢速I/O设备所设计的传统I/O栈能处理的范围.新型存储设备的低延迟(数百ns至100 μs[3])特性要求重新设计和优化I/O路径,以充分发挥其性能优势.已有相关工作[4-6]针对NVMe SSD软件栈进行了优化.同时,持续发展的SAS协议[7],如24G SAS(当前的主流版本是12G SAS)以及预计2026年前后完成制定的24G+ SAS,将为SAS子系统带来更高的传输速率以及更多的新特性,为优化SAS存储协议栈提供了新的机遇与挑战.

然而设计并实现这样的存储协议栈存在着非常大的挑战.具体而言,首先,由于大多数性能测试工具位于用户空间,在测试过程中,其不可避免地会引入存储I/O栈中的诸多开销.例如用户空间与内核空间的切换开销、中断上下文的切换开销等,从而导致针对低延迟I/O设备的性能测试不够准确(尤其是延迟),进而无法准确评估传统I/O栈对设备带来的性能损耗.此外,为了避免不同空间的切换损耗以及内核空间的时间开销,旁路内核实现完全用户空间的I/O栈是一个重要的方向,但是用户空间管理SAS存储设备需要考虑各种问题:1)现有的机制可以在用户空间直接接管PCIe设备,但是SAS设备处于SAS控制器(PCIe设备)的下一级总线,而目前用户空间缺乏对这类设备的控制与访存管理;2)内核空间对外设的访问有着严格的权限来保证安全,用户空间必须在数据安全可靠的前提下优化存储设备的读写性能;3)当前外设对CPU的中断必须由操作系统处理,所以为了完全屏蔽内核的影响,必须能够在用户空间感知外设的中断.

为了解决上述问题,本文首先提出了硬件性能测试工具HwPerfIO,能够消除大部分软件开销的影响以测得更加准确的硬件性能.HwPerfIO运行在内核空间,能够根据用户输入的模型直接在内核空间构造请求,并且采用CPU轮询的形式去除了用户空间与内核空间的切换开销及中断上下文切换所引入的影响.HwPerfIO直接调用外设驱动的I/O接口,能够绕过冗长的内核I/O栈,最大程度地发掘硬件性能并最小限度地引入软件开销.其次,本文分析了HwPerfIO的测试结果,并与典型性能测试工具fio[8]的结果进行对比,分析内核软件栈对性能测试的影响.最后,本文基于SPDK设计并实现了用户空间的SCSI存储软件栈,通过在用户空间实现高效的数据通路降低存储设备的响应时间.在该实现中提出了用户空间SCSI设备管理框架,能够高效地管理设备与驱动,并提出了SCSI底层驱动接口规范以兼容众多的SCSI存储设备;利用输入输出内存管理单元(input output memory management unit, IOMMU)[9]实现了用户空间安全的DMA访问,在安全隔离的情况下保证了数据的高效访问;采用轮询的方式直接在用户空间感知I/O的完成.

本文的主要贡献有3个方面:

1)提出了硬件性能测试工具HwPerfIO,能够更准确地测试低延迟I/O设备的性能,并结合测试结果分析了内核I/O栈对性能测试带来的影响.

2)设计并实现了基于SPDK的用户空间SCSI存储I/O栈,减少了应用程序与存储设备I/O路径上复杂的软件开销,充分发掘了低延迟存储设备的性能优势.

3)通过实验分析,基于用户空间的SCSI存储I/O栈相比内核能够提供更快的访问路径,在队列深度为1时,读写延迟分别降低23.8%和30.1%.

1 背景介绍

在本节中，主要介绍新型存储介质的特征以及存储协议 SAS 的发展现状，并分析传统 I/O 栈在低延迟设备上面临的挑战，以及新型存储设备与协议在 I/O 虚拟化中的应用.

1.1 新型存储介质与 SAS 协议的发展现状

在现代计算机存储系统中，由于不同存储介质的成本与读写性能的差异，存储器体系结构遵循金字塔[3]原则. 如图 1 所示，从顶部的 CPU 寄存器和高速缓存到底部的传统机械硬盘 (HDD) 以及远端存储等，存储设备容量逐级递增，访问效率逐级递减，成本也相应降低. 这种层次结构能够在兼顾成本与容量的同时，保证存储系统的可用性. 可以看出，持久性内存及 NAND SSD 等新兴存储介质的出现填补了动态随机存储器（dynamic random access memory, DRAM）与 HDD 之间巨大的性能鸿沟.

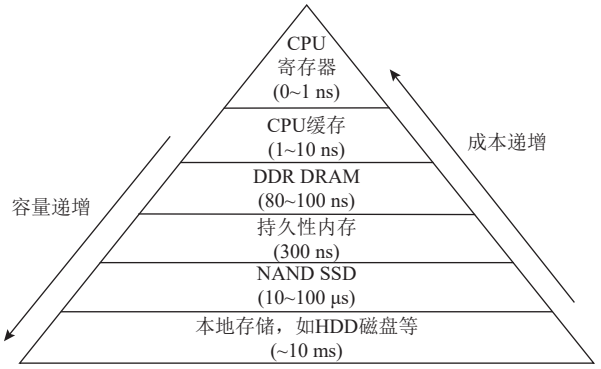


Fig. 1 Storage pyramid [3]

图 1 存储金字塔[3]

得益于 NAND Flash 技术的成熟，高性能 SSD 已经得到了广泛的应用. SSD 起初通过引入闪存转换层（flash translation layer, FTL）[10]以兼容串行 ATA（serial advanced technology attachment, SATA）[11]接口及相应的高级主机控制器接口（advanced host control interface, AHCI）[12]协议迅速进入了消费领域. 但 SATA 协议的带宽上限只有 600 MBps[12]，无法发挥出 SSD 的内在性能. 为了解决 SSD 性能受协议限制的问题，NVMe 协议[13]标准被提出，这是一种高性能低延迟的存储协议，通过 PCIe 总线直接与 CPU 通信，消除了 SATA 协议的性能瓶颈，提供了更高的带宽与更低的延迟.

SAS 协议是另一种支持 SSD 的重要存储协议，由 SCSI 协议发展而来，适用于各种存储介质，包括磁带、HDD 以及 SSD 等. 近 20 年来，SAS 一直是企业级存储的首选，根据国际数据中心 (IDC) 的估计，超过 70% 的企业级存储都在使用 SAS[14]，SAS 作为底层存储协议被广泛应用于 FC-SAN[15]，IP-SAN[16]等存储系统中. 从性能角度来看，NVMe SSD 在带宽、延迟以及每秒进行的读写操作数（input output operations per second, IOPS）等方面都具有一定的领先优势. 但数据中心的构建需要综合考虑成本、性能、可扩展性等诸多因素才能够找到最佳的解决方案.

而 SAS 协议的制定充分考虑了数据中心的需求，在提供高可靠性与高可扩展性的同时，实现了成本与性能的平衡. 在使用扩展器链接的情况下，SAS 协议可以扩展至 64 000 个设备[17]，能够提供巨大容量的存储池，基于 SAS 协议的 SSD 容量可达数十 TB，远高于同时期的 NVMe SSD. 且 NVMe SSD 通过 PCIe 直接连接到主机 CPU，很难在服务器机箱外进行扩展，在该方面 SAS 具有明显的优势.

此外，NVMe 性能优于 SAS 的原因之一是相对于 SCSI 命令集，NVMe 命令集删除了很多关于管理与纠错的功能，而 SAS 拥有一套非常强大的命令集合，功能包括但不限于高级错误检测与纠正、热拔插处理、错误处理及数据保护等. 对于银行、证券等依赖存储实现超高可靠性的场景而言，SAS 的可靠性优势至关重要. 最后，数据中心在技术选型时需要考虑预算与成本，得益于价格优势，SAS 能够有效改进数据中心的总体拥有成本（total cost of ownership, TCO）. 因此，SAS 在未来几年内将继续成为数据中心的基础组件. 此外，SAS 标准仍在不断修订与演进中. 目前 SAS 的主流版本是 2013 年发布的 SAS-3，即 12G SAS. 最新标准是 2017 年开始制定的 SAS-4，即 24G SAS，其实际传输速率为 22.5 Gbps. 24G SAS 之后，SAS 预计将推出 24G+ SAS，预计将于 2026 年后制定完毕，其详细发展趋势路线图详见图 2[7].

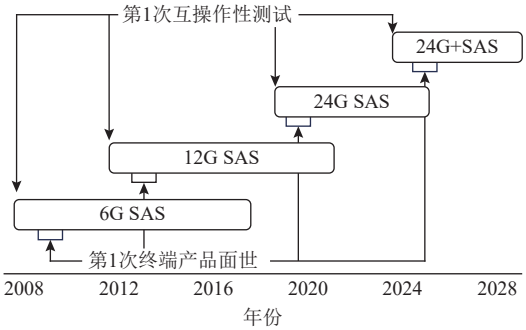


Fig. 2 SAS technical route [7]

图 2 SAS 技术路线[7]

1.2 传统 I/O 栈所面临的挑战

传统内核 I/O 栈是针对慢速 I/O 设备所设计的, DRAM 与 HDD 之间存在巨大的性能差异. 参考图 1[3] 可知, 相较于慢速 I/O 设备 10 ms 级别的延迟, I/O 流程中的软件开销几乎可以忽略不计. 因此针对慢速 I/O 设备所设计的内核 I/O 栈工作良好. 但是随着新兴存储介质的高速发展, 这些设备具有微秒甚至纳秒级的低延迟, 其性能已经远超传统 I/O 栈所能处理的范围.

以 Linux 内核为例, 在传统 I/O 栈中, 用户数据的访存需要经过用户空间的标准库, 通过 I/O 系统调用从用户空间进入内核空间. 一个 I/O 请求在内核空间中将依次经过虚拟文件系统 (virtual file system, VFS)、硬盘文件系统, 以及通用块层和 I/O 调度层等. 然后通过控制器驱动将 I/O 请求提交到对应控制器, 最后由控制器完成真正的数据传输工作, 将主机侧的数据传输到存储设备, 或将存储设备中的数据读取到主机. 由于 DRMA 与 HDD 之间的性能差异巨大, 传统 I/O 栈中的文件系统大都利用 DRAM 构造缓冲区高速缓存来提高读写访问的性能, 并通过异步冲刷缓冲区高速缓存到磁盘. 这种方式能够在保证正确性的前提下对应用程序隐藏 HDD 等慢速存储设备的高 I/O 延迟, 并提高 CPU 利用率. 否则大量的 CPU 周期将用于等待慢速 I/O 设备的响应, 造成 CPU 资源的浪费. 通用块层与 I/O 调度层则通过 I/O 合并及 I/O 调度等机制, 充分考虑传统磁盘顺序读写性能高、随机读写性能低的特性, 最大限度地发掘机械硬盘的性能. 一个 I/O 请求所经历的每个阶段都会引入不同的时间开销, 图 3 显示了这些阶段的时间开销分布. 用户空间发起的 I/O 请求首先经由系统调用进入内核空间, 用户空间与内核空间的切换开销在 0.5~2 μs[18], 某些情况下会更长. 读写请求在整个 I/O 栈中会经历 5~15 μs[19] 的软件延迟, 每一层的处理都会增加额外的开销. 例如在访问 DRAM 时, 内存管理器必须检查是否发生缺页, 若发生缺页, 需要触发缺页中断执行调页. 文件系统则必须检查当前进程是否拥有目标文件的访问权限, 同时维护快照以及日志等机制来确保崩溃一致性. 对于扩展文件系统, 这些机制会引入 40%~200% 的开销, 具体取决于日志的使用情况. 对于 B 树文件系统(B tree file system, Btrfs), 开销可能增加高达 5 倍[20]. 如果发起直接 I/O 请求, Linux 内核必须检查 I/O 请求是否按照目标设备的块粒度对齐, 这些检查全部由 CPU 完成. 当 I/O 请求被提交到控制器后, 后续的延迟由存储设备决定, HDD 的延迟约为 10 ms, SSD 的延迟范围在 10~100 μs. 当存储设备完成 I/O 之后, 向主机发起中断, 此过程涉及中断上下文的切换与中断处理程序的调度, 通常会引入若干微秒的时间开销[21]. 最后中断处理程序沿着调用路径返回到用户程序, 最终应用程序感知到一次 I/O 的完成. 因此, 新型存储介质的低延迟特性要求重新设计或优化 I/O 路径, 减少传统 I/O 栈的性能开销, 以提升系统的整体效率, 满足高性能应用的需求.

1.3 I/O 虚拟化

I/O 虚拟化是虚拟化的重要组成部分, 主要为运行在虚拟机中的应用程序提供 I/O 服务. I/O 操作的最后一步通常是写外设对应的门铃寄存器, 该操作是一条特权指令, 因此需要相应的机制来实现 I/O 虚拟化. 在 SPDK 出现之前主要包括 4 种类型: 第 1 种

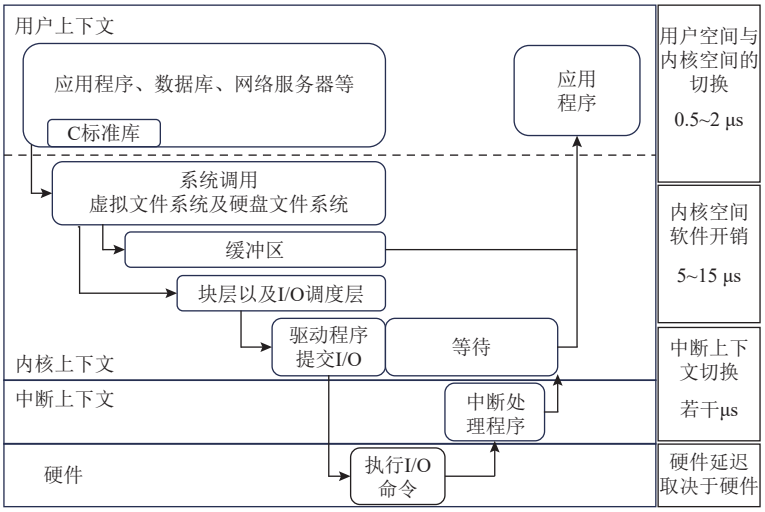


Fig. 3 Linux kernel I/O stack latency distribution

图 3 Linux 内核 I/O 栈延迟分布

是纯软件设备模拟(全虚拟化). 虚拟机感知到的 I/O 设备是由宿主机通过软件模拟呈现的, 客户机发起的 I/O 操作会被宿主机捕获, 宿主机通过模拟软件将 I/O 请求转发到真实的物理设备中. 这是最简单的 I/O 虚拟化技术, 虽然兼容性好、使用方便, 但是由于 I/O 路径过于冗长, 所以性能相对较差. 第 2 种是 I/O 虚拟化技术, 被称为半虚拟化. 这种方式需要在客户机中部署 virtio[22] 前端驱动, 在 QEMU[23] 中实现后端驱动, I/O 请求写外设门铃寄存器的操作均在 virtio 驱动中实现, 能够有效减少客户机与宿主机指令切换的次数. 相对于纯软件的模拟, 有效降低了 I/O 延迟, 性能相对较好. 第 3 种是虚拟化技术, 被称为设备直通. 它利用 CPU 的虚拟化能力, 所以又被称为硬件辅助虚拟化, 尽管能够获得与真机接近的性能, 但是设备只能被唯一的客户机独占. 第 4 种是基于第 3 种类型进一步提出的 SR-IOV[24] 技术, 能够完美解决设备不能在虚拟机之间共享的问题, 但是需要特定外设的支持.

在上述 4 种类型的 I/O 虚拟化中, virtio 已经完全并入 Linux 内核主线, 由于其他虚拟化方案相对苛刻的硬件要求以及 virtio 在性能上的一定优势, virtio 已经作为主流的 I/O 虚拟化方案被广泛部署. virtio 后端用于处理客户机的 I/O 请求, 位于 QEMU 进程之外的实现方案被称为 vhost[25], I/O 处理模块被置于 Linux 内核的方案被称为 vhost-kernel. 随着 SPDK 等用户空间方案的提出, 虚拟化 I/O 的处理模式发生了改变, 虚拟机本身位于用户空间, SPDK 的 I/O 处理模块也位于用户空间, 与 vhost-kernel 对应的 vhost-user 明显是更优的解决方案. vhost-user 利用 SPDK 在用户空间提供的 I/O 处理能力, 使虚拟机能够更快地访问后端存储设备.

2 相关工作

目前市场上已有几款支持 24G SAS 的 SSD, 包括铠侠 PM6 与 PM7 系列[26] 以及三星 PM1653[27] 等. 支持 24G SAS 的存储控制器主要包括博通 96 系列 HBA/RAID 卡[28], 以及 Microchip 旗下的 Adaptec Smart RAID 3200[29] 等. 以 PM7 系列在双端口模式下的官方测试结果为例, 128 KB 顺序读性能最高可达 4200 MBps, 128 KB 顺序写性能最高为 4100 MBps. 而 4 KB 随机读性能达到了惊人的 720 KIOPS, 4 KB 随机写性能最高则为 175 KIOPS. 通过使用 fio 对博通 9670 RAID 控制器连接铠侠 KPM71RUG3T84 24G SAS SSD 的初步实验测试可知, 其单次 I/O 写延迟约为 18.43 μs, 读延迟约为 42.3 μs. 此外, 由于 RAID 控制器普遍采用 DRAM 缓存, Arrakis[20] 提出其所测试 RAID 控制器的平均写延迟约为 25 μs, 在最极端的情况下, 软件栈延迟可能高达 17 μs 甚至更高. 在一次完整的 I/O 过程中, 软件栈开销的占比将超过 50%. 因此, 针对慢速 I/O 设备所设计的传统内核软件栈会限制 SAS SSD 的性能, 无法满足日益增长的高速 I/O 设备的性能需求.

为了解决该问题, 学界与工业界相继提出了众多 I/O 栈优化方案. 以 TH-iSSD[30], ParaBit[31] 等为代表的工作尝试在存储设备中执行计算逻辑, 将 I/O 路径中的部分工作卸载到更高效的硬件设备中, 极大地减轻了 I/O 栈的负担; 文献 [32−33] 则通过优化闪存文件系统以提高 I/O 路径的整体效率, NVStore[34] 通过自适应元数据管理[35] 降低了云存储环境中 NVMe SSD 的未对齐写, 降低了无效写入, 提高了系统带宽; 另一部分研究工作[5-6,36-40] 认为冗长的内核 I/O 路径所引入的时间开销是限制低延迟 I/O 设备性能的主要因素. 这些研究工作旁路部分或全部内核, 在用户空间向应用程序提供直接访问存储设备的能力或在整个 I/O 路径上减少内核的参与. 下文将着重介绍旁路内核的相关工作.

2.1 用户空间文件系统

用户空间文件系统(filesystem in user space, FUSE)[41] 允许用户在无需编译内核的情况下, 创建自定义的文件系统, 最小化 I/O 路径中来自内核本身的干扰. FUSE 主要包括 2 部分, 内核的 FUSE 模块以及用户空间的 libfuse 库. 在 Linux 内核中, 所有文件请求都会通过系统调用进入虚拟文件系统 VFS, VFS 作为统一的接口承接了上层的全部文件请求. 随后 VFS 根据打开文件句柄调用具体文件系统实现的对应文件操作. 假如该文件位于 Btrfs, 则 VFS 将继续调用 Btrfs 实现的文件操作. 如果该文件是用户空间文件系统 FUSE 下的文件对象, 那么 FUSE 所提供的内核模块就能够将该请求转发到位于用户空间的文件系统, 处于用户空间的 libfuse 库负责接收转发自内核空间的 I/O 请求并实现文件系统的公共部分, 而具体文件系统的实现全部位于用户空间. 尽管 FUSE 探索出了一条全新的 I/O 路径, 但是在用户空间实现文件系统必然会引入额外的内核空间/用户空间切换开销, 并不适用于元数据密集型的应用场景, 而是广泛应用于加密场景或作为分布式文件系统的客户端使用等. 为了尽量降低 FUSE 引入的额外开销, 研究者们提出了若干针对 FUSE 的优化方案.

如 ExtFUSE[36] 利用 eBPF 优化 FUSE 性能. AVFS[37] 使用 LD_PRELOAD 重定向 libc 的 POSIX API 接口, 能够在不进行用户内核空间切换的情况下直接调用用户空间文件系统的接口. Direct-FUSE[38] 的所有部分均位于用户空间中, 应用程序可以直接使用预定义的统一文件系统接口来与不同的用户空间文件系统进行交互, 结果显示 Direct-FUSE 的平均性能要比某些原生 FUSE 文件系统高出 11.9%. XFUSE[39] 主要针对高性能存储设备上的文件系统, 允许多个守护进程并行处理不同的 I/O 请求, 充分利用多 CPU 的性能优势, 根据 I/O 请求自适应守护进程的等待周期, 其端到端的延迟可以控制在 4 μs 以内.

2.2 用户空间驱动

用户空间驱动则在应用程序与存储设备之间直接构造了一条高速数据通路, 完全旁路了内核. 其中最典型的工作即 Intel 公司提出的 SPDK[4], SPDK 是一套高性能存储开发工具包, 用于加速 NVMe SSD 的应用软件加速库. SPDK 将设备驱动全部实现在用户空间, 避免了用户空间与内核空间切换的时间开销以及冗长的内核 I/O 路径. 此外, SPDK 使用轮询代替中断, 极大地降低了 I/O 延迟与性能抖动. 最后, SPDK 在 I/O 路径上避免采用任何锁机制进行同步, 降低延迟的同时提高了吞吐. NVMeDirect[5] 是一个用户空间 I/O 框架, 旨在最大化 NVMe SSD 的性能, 同时适应多样化的应用需求. 与 SPDK 不同, NVMeDirect 可以与传统内核 I/O 栈共存, 基于内核的应用程序与基于 NVMeDirect 的应用程序可以同时使用 NVMe SSD 的不同分区. 实验结果表明, 较于内核, NVMeDirect 框架会带来约 30% 的性能提升. uNVMe[6] 是三星电子公司开发的用户空间 NVMe 驱动程序, 其工作模式包括客户端-服务器模型以及客户端-库模型. 前者使用独立的进程管理设备, 支持多客户端的应用同时访问设备; 后者将应用与驱动合并以提高性能但限制每个应用独占一个 NVMe 设备. 实验结果显示, 使用 uNVMe 设备驱动程序, 随机读性能相较于内核能够提高 2~3 倍, 随机写性能的提升则相对有限.

2.3 虚拟化

为了减少 I/O 路径上的开销并限制内核的参与, I/O 虚拟化也是另一个重要的技术手段. Simon 等人[20] 设计并实现了一个全新的操作系统 Arrakis, 其特点在于从数据路径中剔除了 I/O 操作中近乎所有需要内核参与的步骤, 并为应用程序提供与传统 I/O 栈相同的安全模型. Arrakis 将内核划分为数据面与控制面, 应用程序可以在用户空间直接访问底层硬件抽象出的虚拟 I/O 设备实例, 数据平面的操作与数据访问相关, 而控制面只负责虚拟 I/O 设备实例的创建与管理等. 由于数据面与控制面的分离, 进而将内核从绝大多数的数据流中分离. 此外, Caulfield 等人[40] 针对 NVMe SSD 虚拟化所提出的 Moneta-D 是一种全新的存储软硬件架构. 该架构能够消除 I/O 路径中软件开销的两大来源, 即进入内核与文件系统时的权限检查. 主要技术创新在于为每个进程提供私有的虚拟化接口并将文件系统保护检查移至硬件中. 未经修改的应用程序在使用 Moneta 后, 在延迟、带宽和数据库吞吐量方面均有显著的提升. 其中, 带宽提高 2~8 倍, 延迟降低约 60%.

3 硬件性能测试工具 HwPerfIO

为了更加准确地测试硬件 I/O 性能, 本文设计了完全位于内核空间的 HwPerfIO, 在 I/O 执行统计过程中完全无需考虑用户空间与内核空间的切换开销. 并且 HwPerfIO 直接与控制器驱动交互, 最大限度地减少了整个 I/O 路径上的其他软件开销, 所以 HwPerfIO 能够测得最贴近硬件真实情况的性能.

主流的硬件性能测试工具包括 fio[8], vdbench[42] 等, 这些工具均位于用户空间, 参考图 4 可知, 在用户空间构造 I/O 请求后通过系统调用接口将 I/O 发送到对应的设备, 这种测试方法无法避免地引入了内核软件栈的时间开销. 而为了评估系统软件栈的开销, 许多研究通常通过模拟将所有读写请求都落入内存来进行测量. 例如, Arrakis[20] 通过在一个密集的 10 000 次迭代循环中依次执行小写和 fsync 操作以确保每一次的写操作都能够落入存储设备而非缓冲区高速缓存中. 尽管这种方法在一定程度上能够统计出 I/O 过程的平均 CPU 开销, 但也具有一定的局限性. 通过 DRAM 模拟磁盘的方式需要向文件系统提供块 I/O 的提交方法, DRAM 模拟盘所提供的实现就是简单的内存拷贝操作. 而通用块层提供的块 I/O 提交方法则涵盖了 I/O 排队、I/O 调度以及 I/O 合并等机制. 因此该方法也就无法准确刻画通用块层中相应的软件开销, 所以主要用于分析文件系统带来的 CPU 开销. 为了实现硬件设备性能的准确测量, 本文设计并实现了一种硬件性能测试方法, 如图 4 所示. 首先, 为了解决用户空间内核空间切换以及内核本身引入的时间开销问题, 本文提出了基于内核空间的测试工具, 将 HwPerfIO 实现在内核空间(3.1 节). 其次, 为了减少中断上下文的切换开销, HwPerfIO 采用

轮询机制达到了有效降低 I/O 路径上延迟开销的目的，能够准确地测试出硬件本身的初始性能（3.2 节）.

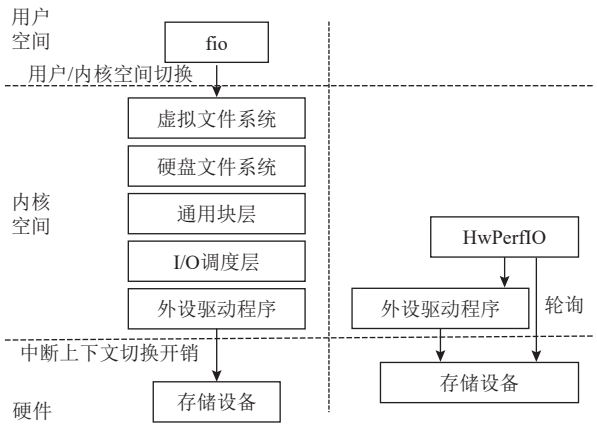


Fig. 4 HwPerfIO vs fio

图 4 HwPerfIO 与 fio 对比

3.1 基于内核空间的 HwPerfIO

通常来说，设备驱动位于内核 I/O 栈的最底层，直接与硬件设备通信，是连接上层 I/O 请求与底层硬件的关键组件，负责维护并管理请求描述符并发起直接内存访问（direct memory access, DMA）[43]. 请求描述符位于连续的一致性缓存 DMA 内存[43] 中，且请求描述符的个数等于外设的队列深度（外设能够同时处理的命令数）. 请求描述符包含完成一次 I/O 的必要信息：1）DMA 控制信息以及对应传输协议的命令信息. 如果外设是 SCSI 控制器，那么请求描述符包含命令描述符块（command descriptor block, CDB）[44]，即 SCSI 命令；如果外设是 NVMe SSD，那么请求描述符包含 NVMe 命令. 2）数据负载信息. 请求描述符需要描述主机内存的负载信息，即分散聚合（scatter-gather, SG）[43]DMA 链表. SG 链表由若干 SG 项构成，SG 项描述了不连续物理内存的地址与长度.

为了防止内核 I/O 栈对性能测试的干扰，HwPerfIO 以内核模块的形式直接实现在内核空间，且不经过任何 I/O 栈直接与外设驱动通信，其核心工作就是构造请求描述符. HwPerfIO 允许用户指定不同的 I/O 模式，典型 I/O 模式包括用于测试吞吐率的 4 KB 小 I/O 随机读写以及用于测试带宽的 256 KB 大 I/O 顺序读写. HwPerfIO 在内核空间直接根据用户输入的 I/O 模型构造外设需要的全部请求描述符，该构造过程仅发生在 HwPerfIO 的初始化过程中，在后续的性能测试中能够节省请求描述符的准备时间. 当所有的准备工作就绪之后，CPU 只需要通过内存映射 I/O（memory map I/O, MMIO）[24] 将请求描述符地址写入外设提供的门铃寄存器中就可以发起 DMA. 写门铃寄存器的操作本质是一条 CPU mov 指令，当该指令返回后，CPU 就可以继续发起下一次 DMA. HwPerfIO 在循环中不断地发起 DMA，直到硬件提交队列中没有空间容纳更多的 I/O 请求，每当有完成命令释放提交队列资源时，HwPerfIO 继续发起 DMA 以保证硬件提交队列始终处于忙碌状态. 硬件的提交队列深度代表了外设能同时处理的命令数，反映了硬件执行 I/O 的能力，当外设的提交队列始终充满待处理 I/O 请求时，硬件性能达到最佳.

3.2 基于轮询的通知机制

当 CPU 发起 DMA 后，后续的数据传输工作将全部由 DMA 控制器完成. 当数据传输完成后，外设主动通过 DMA 将响应描述符传输到主机内存，随后中断 CPU，CPU 通过中断服务例程读取响应描述符中的完成信息，向 I/O 栈上报 I/O 成功或失败的完成状态. HwPerfIO 采用轮询的方式检查 I/O 完成，能够在第一时间感知从而避免中断上下文的切换等开销. 而且 HwPerfIO 采用执行 I/O 任务之外的 CPU 核心来轮询 I/O 的完成并完成性能统计，极大程度地发挥了硬件的性能并最小限度地引入软件开销.

HwPerfIO 能够直观地展现出在指定 I/O 模式下的带宽、吞吐率及平均延迟. 在 4 KB 随机写的同等测试条件下，HwPerfIO 与 fio 测试所得单次 I/O 的平均延迟分别为 13.76 μs 与 18.72 μs. 其中 13.76 μs 接近硬件的访问时间，其差值即为内核 I/O 栈所引入的软件开销，约为 5 μs. 所以 HwPerfIO 的确能够测得更加真实准确的硬件性能. 详细的分析与论证详见 5.1 节.

4 基于 SPDK 的用户空间 SCSI 软件栈

结合第 3 节 HwPerfIO 的测试结果，在特定测试条件下，软件栈的开销占整个 I/O 路径延迟的 26.7%，所以构建基于用户空间的 SCSI I/O 栈具有重要意义. 为了充分释放 SAS SSD 的性能潜力，本文设计并实现了用户空间的 SCSI 系统软件栈，弥补了当前用户空间存储软件架构缺乏对 SCSI 子系统支持的不足.

SCSI 子系统是内核的重要组成部分，通过多层次的抽象设计，提供了统一的接口处理不同类型的 SCSI 设备. 如图 5 所示，Linux SCSI 子系统通常被分为上中下 3 层：1）上层是具体的 SCSI 设备驱动（如机械硬盘），Linux 内核中的实现为 sd.c；2）中间层是 SCSI 总线驱动，提供了一组通用接口与服务，管理着 SCSI 命令的整个生命周期，该部分通常由操作系统开发者维护；3）下层是具体的 SCSI 控制器驱动，

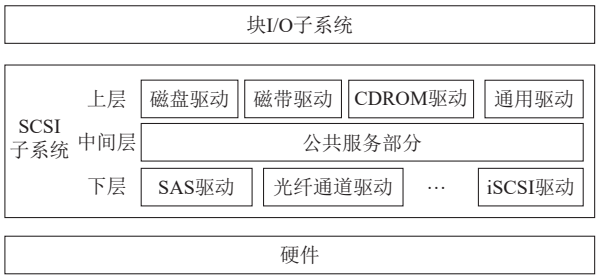


Fig. 5 Linux kernel SCSI subsystem layered architecture

图 5 Linux 内核 SCSI 子系统分层架构

驱动程序直接与硬件通信,负责将中间层传递的 SCSI 命令转换为特定设备所需要的命令,通常由各自外设厂商自行开发维护.

SCSI 三层驱动模型进行了完善的功能划分,定义了高效通用的接口. 如果在用户空间实现 SCSI 软件栈模型,首先需要在用户空间建立有效的 SCSI 设备管理框架;其次需要保证 DMA 数据传输的安全性;最后,用户空间 SCSI 存储子系统应向应用提供便捷的编程接口. 当前的用户空间存储研究中,SPDK 提供了相对完善的基础设施,在 SPDK 的基础上实现用户空间 SCSI 软件栈能够简化实现并丰富用户空间 SCSI 的使用场景. 原生 SPDK 主要包含 3 部分:1)最靠近硬件的是硬件驱动,如 NVMe 驱动等,这是 SPDK 的基础组件;2)包含块设备抽象的存储服务层,主要目的是将底层的不同存储设备抽象为统一形式,在此基础上提供了诸如压缩去冗、逻辑卷管理、服务质量等存储服务;3)存储协议层主要包含 vhost-scsi Target 等一系列存储协议,主要目的是丰富应用场景,为网络对端或虚拟机中的启动器提供存储服务. 因此,本文基于 SPDK 设计并实现了用户空间的 SCSI 驱动框架.

本文提出的用户空间 SCSI 子系统设备管理框架实现在存储服务层,详见 4.1 节所述. 该框架主要对应 Linux 内核 SCSI 子系统的上层及中层,目的是兼容不同厂商的 SCSI 控制器,提供统一的规范与接口,并管理所有的 SCSI 存储设备,以及提供 SCSI 块设备的抽象. 同时该框架对应于 SPDK 中的 NVMe 块设备模块,其中用户空间 SCSI 块设备是本文提出的一种全新设备类型,代表 SCSI 存储设备的抽象,类似于 NVMe 块设备,均可作为 SPDK 原生块设备具体的后端存储设备. 在硬件驱动部分,本文实现了一个用户空间驱动的实例,即 mpt3sas[45],主要对应 SCSI 子系统的下层,同时对应于 NVMe SSD 设备驱动,均位于硬件驱动层,用于在用户空间直接驱动外设,详见 4.2 节. 图 6 描述了用户空间 SCSI 子系统的两大功能模块与原生 SPDK 的关联关系.

4.1 SCSI 设备管理框架的用户空间实现

用户空间 SCSI 设备管理框架主要实现了 4 部分功能:SCSI 块设备模块的实现;RPC 管理接口的实现;驱动设备管理功能的实现;SCSI 底层驱动接口与规范的实现.

1)SCSI 块设备模块的实现. 该模块主要用于接收上层的通用 I/O 命令,这是 SPDK 规定的自定义块设备必须实现的部分. SPDK 原生块设备为所有后端设备提供了统一的 API,上层应用可以通过统一的 I/O 命令下发到目标块设备. 主要接口包括命令入队、命令重置以及超时重传等,并支持通过远程系统调用(remote procedure call, RPC)来配置块设备. 其中最重要的是 I/O 处理,即命令入队函数,它负责通用 I/O 命令与自定义块设备命令的转换. 在本例中,I/O 处理函数需要将来自上层的通用命令转换为 SCSI 协议的标准命令,如果该块设备后端的设备是 NVMe SSD,那么 NVMe 块设备模块负责通用 I/O 命令到 NVMe 命令的转换.

2)RPC 管理接口的实现. 该模块主要用于配置块设备,RPC 是 SPDK 的一种重要通信机制,允许用户以及应用程序对 SPDK 进行远程配置与控制. 主要功能包括但不限于设备的创建与销毁、设备状态的查询等. SPDK 的 RPC 基于 JSON-RPC 2.0 实现,所有的请求与响应均以 JSON 编码,有效降低了复杂存储系统的管理难度.

3)驱动设备管理功能的实现. 用户空间无法沿用内核空间的设备驱动管理方式. 所以本文设计实现了一种全新的驱动设备管理办法,在设备管理框架中实现了 2 个全局链表,分别用于链接设备与驱动. 所有需要初始化的设备及驱动在 SPDK 初始化阶段利用编译器指令__attribute__(constructor) 先于主函数完成构造. 所有 SCSI 后端存储设备依赖于底层驱动的设备上报机制,每当底层驱动上报一个物理设备时,SCSI 中间层会将其注册到全局设备链表中,每一个设备的成功注册都会遍历全局驱动链表,如果设备与驱动能够完成匹配,则进入驱动初始化阶段,直到初始化过程结束. 此时用户或应用程序就可以通过 RPC 接口来管理设备. 以 SCSI 块设备的创建为例,用户编写如图 7 所示的 JSON 示例便可基于底层驱动上报的物理设备 sda 创建一个名为 custom 的 400 GB 的 SCSI 块设备. SCSI 块设备与具体的物理 SCSI 设备之间存在用户定义的映射关系,该机制允许用户便捷地定制自己对硬件设备的需求,允许多用户共享同

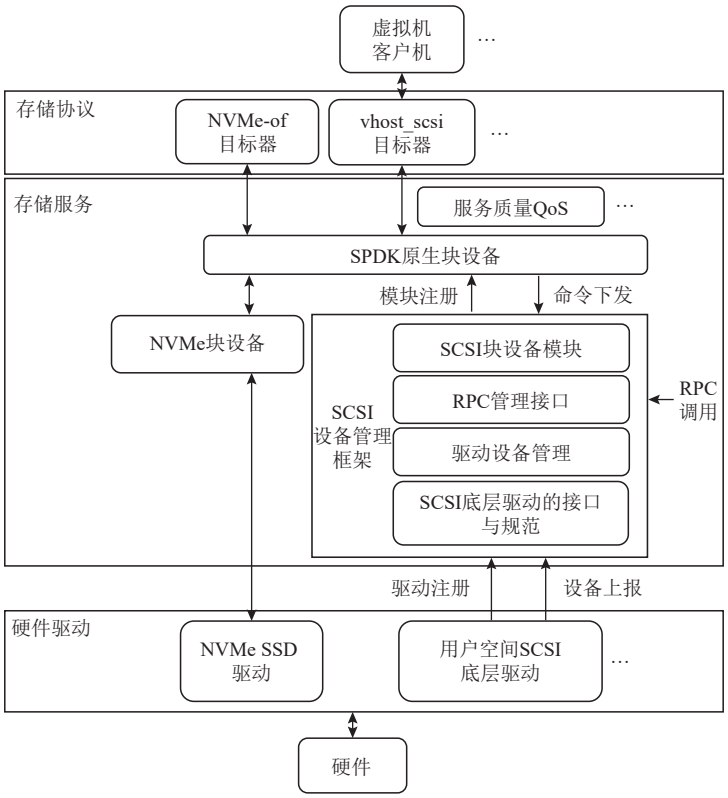


Fig. 6 User space SCSI storage I/O stack based on SPDK

图 6 基于 SPDK 的用户空间 SCSI 存储 I/O 栈

```
{
 "subsystems": [
  {
   "subsystem": "bdev",
   "config": [
    {
     "method": "bdev_usr_scsi_create", /*创建了SCSI块设备方法*/
     "params": {
      "name": "custom", /*SCSI块设备实例名称*/
      "num_blocks": 838860800, /*SCSI块设备容量*/
      "block_size": 512, /*SCSI块设备逻辑块大小*/
      "disk_name": "sda"/*SCSI块设备后端物理设备*/
     }
    }
   ]
  }
 ]
}
```

Fig. 7 JSON command to create SCSI block device

图 7 创建 SCSI 块设备的 JSON 命令

一个存储设备.

4)SCSI 底层驱动接口与规范的实现. 本文在用户空间实现了 SCSI 子系统存储框架, 由于硬件设备驱动大都由硬件制造厂商开发维护, 为了能够兼容不同厂商的存储产品, 本文在 SCSI 设备管理框架与用户空间 SCSI 驱动之间实现了一套接口规范, 遵从该规范实现的用户空间 SCSI 底层驱动均可无缝接入该系统. 该接口规范主要明确了用户空间 SCSI 底层驱动必须实现的方法, 主要包括但不限于:

① SCSI 命令入队. 该方法用于接收 SCSI 设备管理框架所规定的命令格式并转换为存储厂商自定义的命令格式以适配不同厂商的 SCSI 存储控制器.

② SCSI 总线扫描. 底层驱动必须实现总线扫描方法, 该方法用于主动发现 SCSI 控制器连接的 SCSI 存储设备, 被发现的存储设备会被上报给 SCSI 设备管理框架.

③ SCSI 控制器重置. 该方法用于尝试恢复故障的控制器.

④ SCSI 设备重置. 该方法用于尝试恢复故障的设备.

4.2 SCSI 硬件驱动的用户空间实现

SPDK 硬件驱动部分负责直接与硬件交互, 该部分实现了博通 mpt3sas[45] 驱动的用户空间化, 核心逻辑主要包括 3 部分: 1)存储控制器基址寄存器(base address register, BAR)空间的映射; 2)DMA 空间的申请; 3)轮询机制的实现.

1)BAR 空间的映射.PCIe 配置空间与 BAR 空间是 PCIe 设备向操作系统暴露的编程接口, 这 2 段空间均需要映射到系统物理地址空间(system physical address, SPA), 即主机的 CPU 地址空间中, 其中配置空间的映射在基本输入输出系统(basic input/output system, BIOS)启动阶段完成, BAR 空间的映射在驱

动初始化过程中完成. 完成映射后，CPU 可以使用与内存访问相同的方式直接访问 PCIe 设备的寄存器接口，相比于端口 I/O 极大地提高了编程的便携性与统一性. 如果驱动位于内核空间，内核通过 remap 机制建立内核页表，构造虚拟地址与 PCIe 寄存器地址的映射关系，如果将驱动实现在用户空间，则必须利用用户空间驱动框架 VFIO(virtual function I/O)[46] 将 PCIe 的配置空间与 BAR 空间映射到用户空间，即建立 PCIe 寄存器物理地址与虚拟地址的用户空间页表. 在 Intel 平台 VFIO 充分利用 Vt-d 所提供的 DMA 重映射以及中断重映射技术，在保证直通设备 DMA 安全性的同时可以达到接近物理设备的 I/O 性能，用户空间进程可以直接使用 VFIO 驱动访问硬件，从而可以在用户空间完成设备驱动框架的构建.

2)DMA 空间的申请.DMA 是数据传输的主要方式. 在内核空间中，驱动程序通过内核实现的安全接口申请固定的、缓存一致性的 DMA 内存，该内存用于存放固定的信息(如请求描述符、响应描述符等)，该内存空间在驱动的整个生命周期中均有效，所以这种 DMA 映射方式被称为一致性 DMA 映射. 与之相对应的是流式 DMA 映射，DMA 需要显式地冲刷或失效缓存来保证数据负载 DMA 缓存的一致性，其具体实现方式取决于不同类型的 CPU 架构. 在用户空间中，SPDK 同样实现了类似的 DMA 分配函数来保证 DMA 的缓存一致性. SPDK 通过 Linux 内核的大页机制保证了物理地址与虚拟地址的固定绑定关系，用户空间所有的 DMA 操作都必须依赖该函数，这样申请出的 DMA 内存空间才是安全可靠的.

3)轮询机制的实现. 外设通知内核 I/O 完成的主要方式是中断，中断控制器接收到外设发送的中断信号后中断 CPU，进而在中断上下文执行对应的中断处理函数. 中断上下文切换以及中断处理函数调度所带来的 CPU 开销在新型高速存储设备上无法忽略，所以 SPDK 采用事件通知的机制轮询 I/O 操作的完成，轮询器可以有效降低系统调用以及上下文切换带来的 CPU 开销，提升 I/O 处理效率.

5 实验结果分析

本文的测试均基于 Dell PowerEdge R750 服务器进行，相关配置信息详见表 1.

5.1 HwPerfIO 基础性能测试与分析

首先使用 HwPerfIO 测试铠侠 SAS SSD 的单盘基础性能，包括带宽、延迟及吞吐率，并将结果与 fio 在相同测试条件下进行比较. 测试参数见表 2，实验结果如图 8 所示.

Table 1 Experiment Environment

表 1 实验环境

软硬件	配置
CPU	Intel 至强金牌 CPU 6348 56 核心
内存/GB	256
操作系统	Ubuntu22.04 5.15.0-72-generic
HBA	9500-8e HBA [47]
SPDK 版本	v23.01
驱动	mpt3sas
SAS SSD	铠侠 24 G SAS SSD KPM71RUG3T84 单端口模式
测试工具	fio-3.35 filebench-1.5[48] HwPerfIO

Table 2 Basic Performance Test Parameters

表 2 基础性能测试参数

参数类型	带宽测试对应参数配置	延迟测试对应参数配置	吞吐率测试对应参数配置
队列深度	1	1	1
工作任务数	16	1	1
I/O 负载类型	顺序读写	随机读写	随机读写
单次 I/O 大小/KB	128	4	4
运行时间/s	30	30	30
fio ioengine	libaio	libaio	libaio

1)可以观察到 HwPerfIO 与 fio 测试所得的带宽没有明显差异. 这是因为带宽测试的 I/O 模型并非延迟敏感型负载，单次 I/O 延迟主要取决于块大小，且与数据块大小正相关，如 128 KB 的读写延迟在 200 μs 左右，内核软件栈所引入的微秒量级时间开销对整体性能测试并不会产生明显影响. 由于单次 I/O 传输数据块大，硬件大部分时间都在进行有效的数据传输工作，相对小的队列深度就可以保证硬件持续不断地工作，所以队列深度对带宽测试的影响相对较低. 并且考虑到大多数情况下单个 CPU 无法满足硬件的能力上限，因此带宽的测试结果主要取决于工作线程数，即测试所用的 CPU 核心数. 当到达硬件带宽上限后，继续增加 CPU 核心不会显著提升性能. 相反，当工作线程的数量超过系统 CPU 核心数量后，由于任务之间的相互竞争，性能可能会有所下降，这种现象在计算能力较弱的 CPU 上表现更加明显，所以工作线程的数量不能超过 CPU 总数. 选用表 2 所描述的测试参数即可测得铠侠 SAS SSD 在单端口模式下的带宽上限.因此，在带宽测试方面，HwPerfIO 并不会比 fio 得到更加准确的结果，同时也说明 fio 和

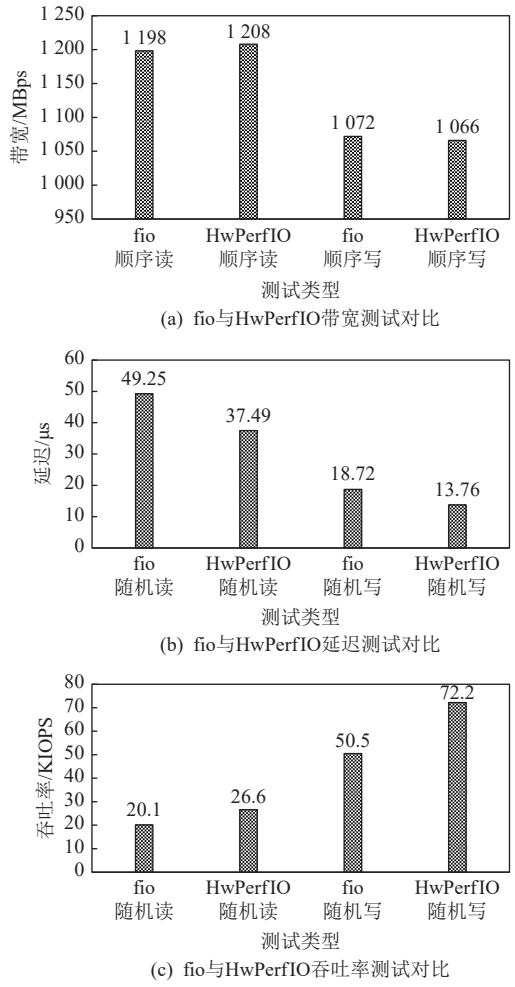


Fig. 8 Comparison of fio and HwPerfIO I/O test

图 8 fio 与 HwPerfIO I/O 测试对比

HwPerfIO 都能够提供可靠的性能评估.

2)延迟是存储设备需要考虑的重要性能指标之一,单次 I/O 延迟能够最直观地表现出 I/O 栈的额外时间开销,对于 4 KB 的随机读操作,HwPerfIO 与 fio 测得的延迟分别是 13.76 μs 与 18.72 μs;对于 4 KB 的随机写操作,延迟分别为 37.49 μs 与 49.25 μs. 由于一定深度的硬件队列普遍存在于主流存储设备中,对于刚入队的 I/O 请求而言,如果此时队列处于繁忙状态,那么这个 I/O 请求将持续排队直到被硬件处理. 尽管会得到相对较高的吞吐率,但是平均延迟也会增大,无法反映出硬件处理单个 I/O 请求的实际时间,所以本文在测试平均延迟时始终保持队列深度为 1.

3)在 fio 与 HwPerfIO 的测试结果中,随机读的吞吐率分别为 20.1 KIOPS 与 26.6 KIOPS,随机写的吞吐率分别为 50.5 KIOPS 与 72.2 KIOPS,在 4 KB 的随机读写测试中,HwPerfIO 测得的吞吐率分别比 fio 高出约 24.4% 与 30.5%,即内核 I/O 栈为性能测试带来了一定的不准确性. 吞吐率主要考察硬件处理密集型随机读写小 I/O 的能力,属于延迟敏感型负载. 不同于带宽负载,小 I/O 负载时,硬件大部分时间都在处理小 I/O 之间的切换,而并非数据传输. 为了测试出硬件的极限性能,随机读写的小 I/O 必须充分利用硬件所提供的异步提交队列,当队列中待处理请求充足时,额外的 CPU 资源并不会带来正向的吞吐率收益. 但是当硬件性能到达极限后,单个 I/O 平均延迟增大,且流水线会稀释每一个 I/O 请求在 I/O 路径上的延迟,无法准确反映出软件栈对吞吐率的影响. 所以测试参数同表 2 中所述,选择队列深度为 1,工作任务数为 1,单次 I/O 大小为 4 KB,这样能够反映出软件栈对吞吐率的最大影响. 单次 I/O 延迟对整体吞吐率的影响非常显著,内核软件栈所引入的微秒量级时间开销会对测试结果产生明显的影响,HwPerfIO 能够测试出硬件更加准确的吞吐率,这也从侧面印证了 Linux 内核 I/O 栈中软件开销对性能测试的影响.

5.2 用户空间 SCSI I/O 栈的性能测试与分析

为了评估 SAS SSD 在不同软件栈下的性能表现,我们使用 SPDK 提供的内置工具 bdevperf[49] 和主流的内核 I/O 栈性能测试工具 fio 分别对用户空间 SCSI 与 Linux 内核管理的 SAS SSD 进行了性能测试与分析. fio 是由 Jens Axboe(Linux 块层的维护者)主导开发的开源 I/O 测试工具,支持多种不同类型的 I/O 引擎,能够提供丰富的 I/O 性能指标,其作为 Linux 下典型存储性能测试工具得到了广泛的认可与应用,所以选择 fio 作为传统 Linux 内核 I/O 栈的性能测试工具. 此处不使用 HwPerfIO 代替 fio 测试 Linux 内核管理的 SAS SSD 的原因是:本节的主要目的是评估用户空间 SCSI I/O 栈与内核空间 I/O 栈的性能差异,如果使用 HwPerfIO 代替 fio 进行测试,将无法测试出内核 I/O 栈对存储设备的性能影响,所以无法采用 HwPerfIO 代替 fio 进行测试.

而基于 SPDK 构建的用户空间 SCSI 软件栈与传统 Linux 内核软件栈并不兼容,所以无法直接使用 fio 进行性能测试. 而 bdevperf 是 SPDK 中提供的性能评估工具. bdevperf 提供了与 fio 类似的命令行格式与配置方式,支持用户自定义测试参数,包括但不限于 CPU 核心数、块大小、I/O 队列深度等,并能够支持一系列的基准操作,包括顺序读写、随机读写以及各种混合 I/O 模式. HwPerfIO 与用户空间 SAS SSD 性能测试工具 bdevperf 的主要区别在于其目的不同. HwPerfIO 的主要目的是消除软件栈中可能引入的开销以测试硬件本身的性能. 而基于用户空间驱动所构建的 SAS SSD 性能测试工具(如 bdevperf)实际是在测试用户空间软件栈与 SAS SSD 的整体性能.

图 9 给出了在工作任务数量固定为 1 且测试时间固定的前提下, 分别测试 Linux 内核 I/O 栈与用户空间 SCSI I/O 栈下 4 KB 随机读写性能随队列深度变化的情况, 其他相关测试参数详见表 3.

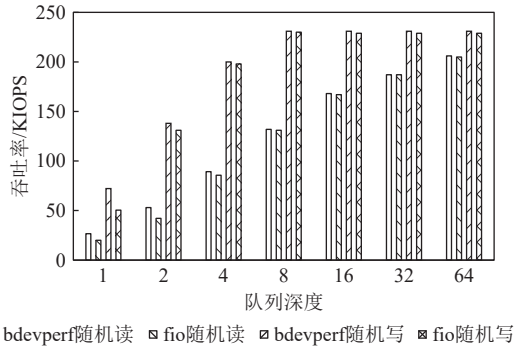


Fig. 9 IOPS under different number of queue depth

图 9 不同队列深度下的吞吐率

Table 3 fio and bdevperf Test Configuration

表 3 fio 与 bdevperf 测试配置

参数类型	带宽测试对应参数配置	吞吐率测试对应参数配置
队列深度	1	1~64
工作任务数	1~16	1
块大小/KB	128	4
执行时间/s	300	300
I/O 模型	随机/顺序读写	随机/顺序读写
fio ioengine	libaio	libaio

随着队列深度的增大, 吞吐率也随之增大, 直至到达硬件上限. bdevperf 的测试结果反映本文所提出的用户空间 SCSI I/O 栈, fio 的测试结果对应于传统 Linux 内核 I/O 栈. 可以看出, 在队列深度较小时, 基于用户空间的 I/O 栈相比于内核空间 I/O 栈在延迟敏感型的小 I/O 处理上表现更佳, 读写 IOPS 分别提升约为 24.4% 和 30.1%. 这主要归因于减少了内核空间与用户空间的切换开销, 优化了 I/O 路径, 使得系统响应速度更快. 当队列深度增大后, 命令提交队列的流水线开始起主要作用, 软件栈所占的开销被流水线所隐藏, 硬件的极限吞吐率差异相对较小. 图 10 展示了在队列深度保持为 1 的情况下, Linux 内核 I/O 栈与用户空间 SCSI I/O 栈下 128 KB 顺序读写随工作任务数变化的性能变化情况. 在顺序读写测试中, 用户空间 SCSI I/O 栈与 Linux 内核 I/O 栈的带宽表现几乎一致, 即对于非延迟敏感型操作, 单次 I/O 的延迟影响较小, 用户空间 SCSI I/O 栈主要优化目标为延迟敏感型的小 I/O 负载.

5.3 基于用户空间 SCSI 的 I/O 虚拟化

尽管 vhost-user 提升了 I/O 性能, 但 SCSI 始终处

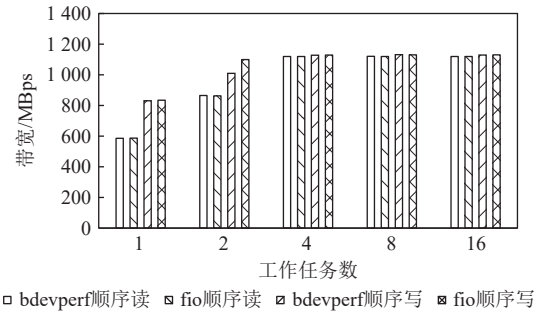


Fig. 10 Bandwidth under different number of tasks

图 10 不同工作任务数下的带宽

于内核空间, 无法利用 vhost-user 来提升其性能. 本文提出的用户空间 SCSI I/O 栈则填补了这一空缺, 在虚拟机与 SCSI 后端存储设备之间提供了一条高速 I/O 通路. 图 11 描述了客户机如何利用 vhost-user 访问用户空间 SCSI 所管理的存储设备. 虚拟机中生成的 I/O 请求通过客户机 virtio 驱动进入到 virtqueues 队列, SPDK 持续轮询该队列的 I/O 请求, 如果有新的 I/O 请求产生, 则转发到 SCSI 块设备模块, 并通过实现在用户空间的 SCSI 驱动将 I/O 请求发往具体的存储设备, 该过程全部在用户空间完成, 有效降低了 Linux 内核所带来的各种延迟开销.

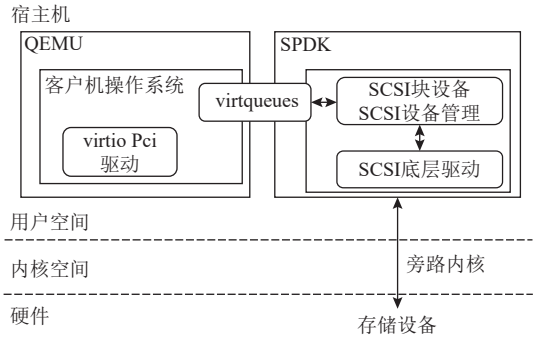


Fig. 11 I/O virtualization based on user space SCSI I/O stack

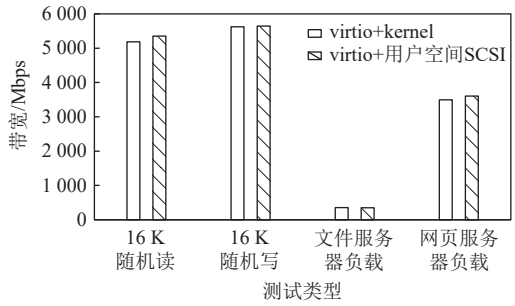
图 11 基于用户空间 SCSI I/O 栈的 I/O 虚拟化

为了测试基于用户空间 SCSI 存储框架的 I/O 虚拟化性能, 本文对比了 2 种方案: virtio + Linux 内核 SCSI 与 virtio + 用户空间 SCSI 的文件系统性能. 具体来说, 存储设备分别通过 Linux 内核与 vhost-user 的方式被客户机识别后, 在该虚拟 I/O 设备上建立第 4 代扩展文件系统(fourth extended filesystem, EXT4), 并进行了 filebench 自定义随机读写、网页服务器负载 webserver 以及文件服务器负载 fileserver 的测试.

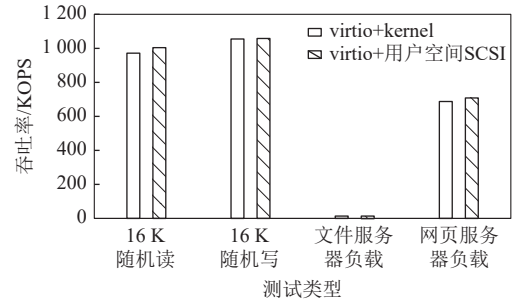
本文使用 filebench 工具构造了随机读写负载, 测试过程仅执行 3 个简单操作: 1)打开文件; 2)读取整个 16 KB 文件/写入整个 16 KB 文件; 3)关闭文件.

图 12 展示了 2 种方案在自定义随机读写、网页服

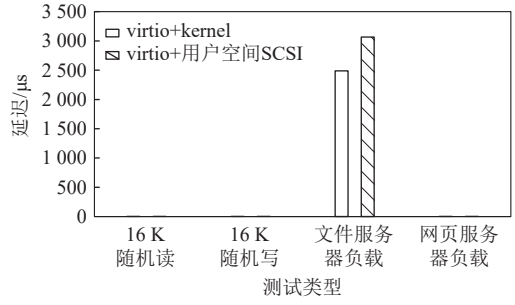
务器负载以及文件服务器负载下的性能表现. 对比结果显示, virtio + 用户空间 SCSI 方案相对于 Linux 内核 SCSI 方案在性能上有一定的优势, 具体结果如下. Virtio + 用户空间 SCSI 在随机读写操作与文件服务器负载中均呈现出了与 Linux 内核 SCSI 方案相当的性能. 并且在自定义随机读操作以及网页服务器负载中表现出了一定的优势, 带宽与单位时间内的操作数提升幅度基本一致, 分别为 3.24% 与 3.12%. 同时, 自定义随机读操作以及网页服务器负载的单个操作平均延迟降幅均约为 3.12%. 这表明用户空间 SCSI 在性能不降低的前提下, 能够应用于大部分的使用场景, 为 SCSI 存储设备提供更加丰富的 I/O 虚拟化方案.



(a) 用户空间SCSI I/O虚拟化带宽



(b) 用户空间SCSI I/O虚拟化吞吐率



(c) 用户空间SCSI I/O虚拟化延迟

Fig. 12 User space SCSI I/O virtualization performance

图 12 用户空间 SCSI I/O 虚拟化性能

6 结 论

本文首先提出了存储设备性能测试工具 HwPerfIO, 它能够有效避免 I/O 栈对测试结果的影响, 准确地反映存储设备的硬件性能. HwPerfIO 对 SAS SSD 的测试结果表明, 在 SCSI 协议栈中, 由软件所引入的时间开销已不可忽略. 进而本文提出了一种基于 SPDK 的用户空间 SCSI 存储 I/O 栈, 通过将 SCSI 子系统用户空间化, 实现用户空间对 SCSI 设备的管理与控制, 在应用程序与 SAS SSD 高速设备之间提供了一条高速 I/O 通路. 实验结果表明, 对于延迟敏感的小 I/O 随机负载, 用户空间 SCSI 存储 I/O 栈为存储系统带来了一定的性能提升, 能够进一步发挥出 SAS SSD 硬件的性能潜力.

尽管 CPU 轮询 I/O 能够降低存储系统对应用程序的响应时间, 但是轮询始终会占用 CPU 资源, Intel 的 Sapphire Rapid[50]CPU 架构支持直接在用户空间接收外设中断, 借助 Sapphire Rapid 构建完全用户空间的存储软件栈将是下一步工作计划.

作者贡献声明: 郝栋栋设计了论文的整体逻辑架构, 并完成了论文实验方案的设计与验证; 高聪明完善论文内容, 负责论文结构的讨论与修改; 舒继武提出论文撰写指导意见.

参 考 文 献

[1] Goda A. 3-D NAND technology achievements and future scaling perspectives[J]. IEEE Transactions on Electron Devices, 2020, 67(4): 1373−1381

[2] Baldassin A, Barreto J, Castro D. Persistent memory: A survey of programming support and implementations[J]. ACM Computing Surveys, 2021, 54(7): 1−37

[3] Shu Jiwu. Data Storage Architectures and Technologies [M]. Berlin: Springer, 2024

[4] Yang Z, Harris J R, Walker B, et al. SPDK: A development kit to build high performance storage applications [C] //Proc of the 2017 IEEE Int Conf on Cloud Computing Technology and Science (CloudCom). Piscataway, NJ: IEEE, 2017: 154–161

[5] Kim H J, Lee Y S, Kim J S. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs [C] //Proc of the 8th USENIX Conf on Hot Topics in Storage and File Systems. Berkeley, CA: USENIX Association, 2016: 41–45

[6] OpenMPDK. uNVMe: KV and LBA SSD userspace NVMe driver [CP/OL]. [2024-09-12]. https://github.com/OpenMPDK/uNVMe

[7] SNIA. Serial attached SCSI technology roadmap [EB/OL]. [2024-09-12]. https://www.snia.org/groups/scsi-trade-association-sta-forum/sas-roadmaps/serial-attached-scsi-technology-roadmap

[8] axboe. fio: Flexible I/O Tester [CP/OL]. [2024-09-12]. https://github.com/axboe/fio

[9] AMD. AMD I/O Virtualization Technology (IOMMU) Specification, Rev 3.09-PUB [S]. Santa Clara: Advanced Micro Devices Inc, 2023

[10] Shin J Y, Xia Zenglin, Xu Ningyi, et al. FTL design exploration in reconfigurable high-performance SSD for server applications [C] //Proc of the 23rd Int Conf on Supercomputing. New York: ACM, 2009: 338−349

[11] Serial ATA International Organization. Serial ATA Revision 3.1 [S]. Beaverton: Serial ATA International Organization, 2011

[12] Intel. Serial ATA AHCI 1.3. 1 Specification [S]. Santa Clara: Intel Corp, 2015

[13] NVM Express. Specifications: NVMe specifications overview [EB/OL]. [2024-09-12]. https://nvmexpress.org/specifications/

[14] SNIA. Building an all-flash array with SAS, NVMe or SATA [R]. Santa Clara: SNIA, 2018

[15] Shu Jiwu, Li Bigang, Zheng Weimin. Design and implementation of a SAN system based on the fiber channel protocol[J]. IEEE Transactions on Computers, 2005, 54(4): 439−448

[16] Sun Microsystems, Inc. IP SAN fundamentals: An introduction to IP SANs and iSCSI [EB/OL]. [2024-09-12]. https://www.oracle.com/technetwork/systems/articles/ip-san-fundamentals-149896.pdf

[17] INCITS. SCSI Architecture Model-5 (SAM-5) [S]. Washington: American National Standards Institute, Inc, 2016

[18] The Linux Foundation. I/O latency optimization with polling [R]. San Francisco, CA: The Linux Foundation, 2017

[19] Huang Jian, Badam A, Qureshi M K, et al. Unified address translation for memory-mapped SSDs with FlashMap [C] //Proc of the 42nd Annual Int Symp on Computer Architecture. New York, ACM, 2015: 580−591

[20] Simon P, Li Jialin, Zhang I, et al. Arrakis: The operating system is the control plane [C/OL] //Proc of the 11th USENIX Conf on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2014 [2024-09-12]. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf

[21] Gerhorst L. Analysis of interrupt handling overhead in the Linux kernel [D]. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Department of Computer Science, 2018

[22] OASIS Open. Virtual I/O Device (VIRTIO) Version 1.2 [S]. Burlington: OASIS Open, 2022

[23] qemu. qemu: Official QEMU mirror [CP/OL]. [2024-09-12]. https://github.com/qemu/qemu

[24] PCI-SIG. PCI Express® Base Specification Revision 5.0, Version 1.0 [S]. Beaverton: PCI-SIG, 2019

[25] Michael S. Vhost [CP/OL]. [2024-09-12]. https://github.com/torvalds/linux/tree/master/drivers/vhost

[26] KIOXIA America, Inc. KIOXIA PM7-R series (2.5-inch) [EB/OL]. [2024-09-12].https://americas.kioxia.com/en-us/business/ssd/enterprise-ssd/pm7-r.html

[27] Samsung Electronics Co, Ltd. A landmark release for Enterprise SSDs [EB/OL]. [2024-09-12]. https://semiconductor.samsung.com/cn/ssd/enterprise-ssd/pm1653/

[28] Broadcom Inc. MegaRAID 9670−24i [EB/OL]. [2024-09-12]. https://www.broadcom.com/products/storage/raid-controllers/megaraid-9670-24i

[29] Microchip Technology Inc. Adaptec® RAID adapters [EB/OL]. [2024-09-12]. https://www.microchip.com/en-us/products/storage/adaptec-smartraid-raid-adapters

[30] Shu Jiwu, Fang Kedong, Chen Youmin, et al. TH-iSSD: Design and implementation of a generic and reconfigurable near-data processing framework[J]. ACM Transactions on Embedded Computing Systems, 2022, 37(4): 96−119

[31] Gao Congming, Xin Xin, Lu Youyou, et al. ParaBit: Processing parallel bitwise operations in NAND flash memory based SSDs [C] //Proc of the 54th IEEE/ACM Int Symp on Microarchitecture (MICRO54). New York: ACM, 2021: 59−70

[32] Lu Youyou, Shu Jiwu, Wang Wei. ReconFS: A reconstructable file system on flash storage [C] //Proc of the 12th USENIX Conf on File and Storage Technologies (FAST). Berkeley, CA: USENIX Association, 2014: 75−88

[33] Zhang Jiacheng, Shu Jiwu, Lu Youyou. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices [C] //Proc of the 2016 USENIX Annual Technical Conf (USENIX ATC). Berkeley, CA: USENIX Association, 2016: 87−100

[34] Shu Jiwu, Li Fei, Li Siyang, et al. Towards unaligned writes optimization in cloud storage with high-performance SSDs[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(12): 2923−2937

[35] Li Weijia, Xue Wei, Shu Jiwu, et al. Dynamic hashing: Adaptive metadata management for petabyte-scale file systems [C] //Proc of the 23rd IEEE Conf on Mass Storage Systems and Technologies (NASA/MSST2006). Piscataway, NJ: IEEE, 2006: 159−164

[36] Bijlani A, Ramachandran U. Extension framework for file systems in user space [C] //Proc of the 2019 USENIX Annual Technical Conf (USENIX ATC). Berkeley, CA: USENIX Association, 2019: 121−134

[37] avfs. AVFS —A virtual filesystem [CP/OL]. [2024-09-12]. http://avf.sourceforge.net/

[38] Zhu Yue, Wang Teng, Mohror K, et al. Direct-FUSE: Removing the middleman for high-performance fuse file system support [C] //Proc of the 8th Int Workshop on Runtime and Operating Systems for Super-computers. New York, ACM, 2018 [2024-09-12]. https://www.osti.gov/servlets/purl/1458703

[39] Huai Qianbo, Hsu W, Lu Jiwei, et al. XFUSE: An infrastructure for running filesystem services in user space [C] //Proc of the 2021 USENIX Annual Technical Conf (USENIX ATC). Berkeley, CA: USENIX Association, 2021: 863–875

[40] Caulfield A M, Mollov T I, Eisner L A, et al. Providing safe, user space access to fast, solid state disks[J]. ACM SIGPLAN Notices, 2012, 47(4): 387−400

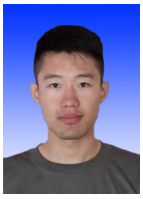
[41] libfuse. libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface [CP/OL]. [2024-09-12]. https://github.com/libfuse/libfuse

[42] Oracle. vdbench users guide [EB/OL]. [2024-09-12]. https://www.oracle.com/technetwork/server-storage/vdbench-1901683.pdf

[43] Corbet J, Rubini A, Kroah-Hartman G. Linux Device Drivers 3rd Edition: Memory Mapping and DMA [M]. Sebastopol: O'Reilly Media, 2006

[44] INCITS. SAS Protocol Layer-5 (SPL-5) [S]. Washington: American

National Standards Institute, Inc, 2023
[45] Sathya P. mpt3sas [CP/OL]. [2024-09-12]. https://github.com/torvalds/linux/tree/master/drivers/scsi/mpt3sas
[46] Alex W. VFIO [CP/OL]. [2024-09-12]. https://github.com/torvalds/linux/tree/master/drivers/vfio
[47] Broadcom Inc. HBA 9500−8e tri-mode storage adapter [EB/OL]. [2024-09-12]. https://www.broadcom.com/products/storage/host-bus-adapters/sas-nvme-9500-8e
[48] filebench. filebench: File system and storage benchmark that uses a custom language to generate a large variety of workloads [CP/OL]. [2024-09-12]. https://github.com/filebench/filebench
[49] Intel. bdevperf [EB/OL]. [2024-09-12]. https://spdk.io/doc/bdevperf.html
[50] The Linux Foundation. User interrupts—A faster way to signal [R]. San Francisco, CA: The Linux Foundation, 2021



Hao Dongdong, born in 1992. PhD candidate. His main research interest includes computer storage systems.

郝栋栋, 1992 年生. 博士研究生. 主要研究方向为计算机存储系统.



Gao Congming, born in 1992. PhD, associate professor, PhD supervisor. Member of CCF. His main research interests include storage system, flash memory, and emerging non-volatile memory technology.

高聪明, 1992 年生. 博士, 副教授, 博士生导师. CCF 会员. 主要研究方向为存储系统、闪存、新型非易失性存储技术.



Shu Jiwu, born in 1968. Professor, PhD supervisor. Fellow of CCF. His main research interests include intelligent storage system, non-volatile memory systems and technologies, storage security and reliability, and parallel and distributed computing.

舒继武, 1968 年生. 教授, 博士生导师. CCF 会士. 主要研究方向为智能存储系统、非易失内存系统与技术、存储安全与可靠性、并行与分布式计算.