中国科学: 信息科学 2024 年 第 54 卷 第 3 期: 461–490
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
泛在操作系统理论、技术与开源生态构建专题 • 评述

面向泛在操作系统的结构化存储

范晓鹏, 阎松, 翁楚良*

华东师范大学数据科学与工程学院, 上海 200062
* 通信作者. E-mail: clweng@dase.ecnu.edu.cn

收稿日期: 2022–10–24; 修回日期: 2023–10–12; 接受日期: 2024–01–16; 网络出版日期: 2024–03–12

国家自然科学基金 (批准号: 62141214, 62272171) 资助项目

摘要 人机物融合泛在计算的新场景和新模式, 需要新型的操作系统, 即泛在操作系统. 存储管理作为泛在操作系统的一项核心功能, 设计轻量化、高性能和动态可适配的存储系统是推进泛在操作系统发展的必要举措. 然而, 对于 "端 (终端设备) – 边 (边缘端) – 云 (云端)" 泛在场景中普遍存在的结构化数据, 传统存储解决方案存在 I/O 放大严重、系统体量过大和软件栈冗余等问题, 难以满足泛在应用的需求. 为此, 本文从系统全栈的角度进行原创性探索, 提出原生表存储系统. 本文首先回顾了计算机系统的发展历史; 之后分析泛在计算时代的新需求, 总结出泛在操作系统的基本形态, 并介绍最新研究成果; 然后剖析了现有结构化存储方案在 "端 – 边 – 云" 场景下面临的挑战; 进一步提出面向泛在操作系统的原生表存储系统, 并从端侧和边/云侧两个场景深入分析其架构优势. 最后, 总结全文并展望未来发展趋势.

关键词 泛在计算, 泛在操作系统, 原生表存储, 软件栈, 端 – 边 – 云

1 引言

泛在计算 [1] 是继主机计算、个人计算、移动计算之后出现的新型计算模式, 该模式下的计算无处不在, 无迹可寻, 信息技术在这种泛在化的场景中蕴含了人 – 机 – 物融合发展的能力 [2]. 它正在引领着计算机系统在形态、结构与行为上的新变革, 与此同时也带来了一系列的新问题与新挑战. 当前的操作系统难以满足人机物融合泛在计算的环境多变性、广谱多样性、需求多样性和场景复杂性 [3]. 为了高效管理海量的异构资源, 满足泛在应用需求, 一类新型操作系统, 即泛在操作系统 (ubiquitous operating system, UOS), 正在出现并处于探索成型期 [4]. 存储管理作为泛在操作系统的一项核心功能, 如何面向泛在操作系统设计轻量化、高性能和动态可适配的存储系统, 是推进泛在操作系统发展的必要举措.

引用格式: 范晓鹏, 阎松, 翁楚良. 面向泛在操作系统的结构化存储. 中国科学: 信息科学, 2024, 54: 461–490, doi: 10.1360/SSI-2022-0415
Fan X P, Yan S, Weng C L. Structured storage for ubiquitous operating systems (in Chinese). Sci Sin Inform, 2024, 54: 461–490, doi: 10.1360/SSI-2022-0415

在泛在操作系统环境中, 面临着 “端 (终端设备) – 边 (边缘端) – 云 (云端)” 3 个层次的存储需求. 在 3 个层级上均存在大量的结构化数据的存储需求. 端侧设备呈现资源极端化和异构化等特性, 需要存储系统具备模块化和强适配性, 在边/云侧可以部署新型存储设备来加速 I/O 处理, 满足泛在应用的低时延和实时性需求. 通行的解决方案是在字节流的文件系统基础之上实现结构化存储. 然而, 这种传统解决方案存在诸多挑战, 难以满足泛在场景需求.

面向端侧的结构化数据存储.

(1) I/O 放大严重. 终端设备上的结构化数据管理依赖于堆叠式的软件栈, 导致了严重的输入/输出 (input/output, I/O) 放大, 这不仅降低了带宽利用率, 还缩短了设备的寿命. 堆叠式软件栈使得 I/O 路径复杂, 造成大量冗余开销, 无法满足泛在应用的 I/O 时效性需求.

(2) 硬件资源受限. 终端设备资源有限, 且设备间资源差异明显. 许多系统针对特定应用场景, 需要优化或重新配置部分功能. 然而, 现有的存储系统缺乏模块化结构, 一体化的笨重的存储系统难以适配到小型、低功耗的端侧设备, 且升级或添加新模块变得困难.

(3) 应用需求多样. 不同泛在应用在一致性需求方面具有多样性. 通常, 采用弱一致性策略可降低硬件资源消耗并提高性能, 而更强的一致性需要更多资源, 可能损害性能. 由于终端设备资源有限, 存储系统需具备灵活的一致性机制, 以在满足泛在应用需求的同时, 最小化资源消耗并提高性能.

面向边/云侧的结构化数据存储.

(4) 硬件不友好. 高性能的新型存储设备, 如 NVMe SSD (nonvolatile memory express solid state disk) 和 PM (persistent memory) 为数据存储带来了新的生机. 它们拥有独特的硬件特性, 比如 NVMe SSD 支持高达 64k 个 I/O 队列, 理论上队列深度高达 64k, PM 具备可按字节寻址和设备内部的条带化特性. 然而, 既有的存储系统往往忽略了或者仅考虑到新型存储设备的部分特性, 导致直接将现有存储系统部署在新硬件上时, 无法充分发挥新硬件的全部潜能.

(5) 软件栈冗余. 对于结构化数据存储, 传统的堆叠式软件栈成为限制泛在操作系统性能的潜在瓶颈. 对于磁盘毫秒级别的硬件延迟, I/O 路径上的瓶颈主要由硬件引起, 这使得厚重的软件开销可以被忽略. 然而, 边/云侧部署的新型存储设备在硬件延迟上已经有了数量级的提升, 厚重的软件栈成为了影响性能的关键因素. 在这种情况下, 如果仍然忽略软件开销, 将无法充分释放新型存储设备的性能潜力, 导致存储系统难以满足泛在应用的低时延和实时性需求.

传统的解决方案无法满足泛在场景下的存储需求, 亟须根据当前泛在操作系统生态构建的发展趋势, 综合考虑 “端 – 边 – 云” 的泛在计算场景, 开展全面系统的新型存储机制的研究. 本文提出原生表存储系统, 以负责泛在操作系统中的存储管理, 从资源严格受限的端侧和基于新型高性能存储设备的边/云侧两个场景进行深入分析.

本文第 2 节对计算机系统发展历史进行简要回顾; 之后第 3 节分析泛在计算时代对泛在操作系统的新需求, 总结出泛在操作系统的基本形态, 并介绍了最新研究成果 XiUOS [5]; 第 4 节剖析了在泛在计算场景下, 现有端侧和边/云侧结构化存储方案存在的诸多问题; 第 5 节提出面向泛在操作系统的原生表存储系统, 并从端侧和边/云侧两个场景深入分析其架构优势. 最后, 对本文进行总结并展望未来的发展趋势.

2 计算机系统发展历程

回顾计算机系统几十年的发展, 其大致经历了大型机计算时代、小型机计算时代、个人计算时代和移动计算时代, 并引领了各自领域的系统生态. 数据存储作为计算机系统的核心功能之一, 在存储

硬件和存储软件方面取得了突破性进展. 一系列高性能、新架构和新特性的存储设备不断涌现并逐步成为不同时代的主流技术, 相应地, 存储软件也经历了不断的优化和革新, 以更好地适配新硬件和满足上层应用多样化的需求. 接下来, 本文结合表 1 梳理计算机系统及数据存储发展的主要历程.

1956 年在 IBM 704 上实现的 GM-NAA I/O 是世界上第 1 个公认的操作系统, 也是第 1 个典型的批处理操作系统, 实现了输入输出自动化管理. 同年, 全球第 1 台硬盘驱动器问世, 它可存储 5 MB 数据, 传输速度达到 10 kB/s, 这表明磁盘存储时代的到来. 该时代计算机的软件是针对特定硬件而设计的, 没有相对独立且具备通用性的存储系统.

1964 年, IBM 发布了 System/360 系列的大型机, 配备 OS/360 系列操作系统. 同时期, IBM 发布了可移动硬盘驱动器. 当时主流的存储系统是面向记录的文件系统 [6], 它们与大型计算机的操作系统息息相关. 与基于字节流的文件系统不同, 在面向记录的文件系统中, 文件是一组记录的集合, 程序读取和写入均以记录为单位. 这种类型的文件系统为后来存储大量面向记录型数据的应用程序和更高级的数据抽象提供了一个很好的设计思路.

Unix 起源于 20 世纪 70 年代的 Multics 项目 [4], 在此期间, 存储设备也进行着革新, 比如 IBM 推出的软盘和温氏硬盘. 围绕软/硬盘的特性开发了大量的存储系统, 如 RT-11 file system 和 CP/M file system. 20 世纪 60 年代末 ∼ 70 年代初, 结构化数据的存储逐渐向上抽象为一个独立于文件系统的数据库管理系统. 1974 年, IBM 推出了 System R, 而后的 1979 年, 第 1 个商业版的关系数据库 Oracle Release 1 诞生, 并提供商用 SQL.

20 世纪 80 年代开始, IBM PC 的推出标志着个人计算时代的到来. Microsoft 的 DOS/Windows 系列、Unix 的变种 Linux 以及 Apple 公司的 macOS 成为主流的 PC 端操作系统 [7]. 存储硬件需要满足大容量、方便携带等需求, CD/DVD, 可移动磁盘开始部署在个人计算机中. 同时, 也涌现出很多针对磁盘优化的存储系统, 如 FatFs, NTFS 和 EXT 系列文件系统. 在这一时代, 用于处理结构化数据存储的关系型数据, DB2, PostgreSQL, MySQL 等纷纷登上了历史舞台.

2000 年以后, 计算机产业进入到移动计算时代, iOS 和安卓的出现很快主导了市场. 这两大操作系统都可以溯源到 Unix, 在核心技术上无实质性变化, 更注重易用性、耐用性和低功耗. 自 20 世纪末 NOR 和 NAND flash 问世以来, 以 flash 为核心的存储设备成为移动计算的主流, 同时涌现出许多针对闪存优化的存储系统, 如 JFFS2 和 F2FS [8]. 进入 21 世纪, 非关系型数据库 NoSQL 开始盛行, 2011 年, 结合 SQL 和 NoSQL 的 NewSQL 概念也开始出现.

步入 2020 年, 一个万物互联的泛在计算时代即将到来 [4]. 泛在计算的环境多变、需求多样、场景复杂, 需要 “沉淀” 一类新型操作系统, 称为 “泛在操作系统” [3]. 泛在操作系统一方面需要具备软件定义和动态适配能力以管理海量的异构存储设备, 另一方面, 随着 NVMe SSD[1)] 和 PM[2)] 等新型高性能存储设备逐渐成为主流, 泛在操作系统需重新思考现有的存储软件栈, 结合硬件特性以释放性能潜力, 并具备柔性灵活的动态可适配能力, 以满足泛在应用的新需求.

3 面向人机物融合泛在计算的操作系统

泛在计算 [1], 又称普适计算, 是继主机计算、个人计算、移动计算之后出现的新型计算模式, 其具备万物数字化、交互网络化、系统智能化、计算泛在化 4 个基本特性 [9]. 人机物融合的泛在计算正在引领着计算机系统在形态、结构与行为上的新变革, 这其中既有新形态下的操作系统又有新形态下的

1) NVM Express Overview. http://www.nvmexpress.org/nvmexpressoverview/.
2) Intel Corporation. http://pmem.io/.

表 1 计算机系统及其相关技术的演进

Table 1 Evolution of computer systems and related technologies

Time	Computer system	Representative OS(s)	Storage hardware	Hardware features	Storage software	Software features
1956	IBM 704, IBM RAMAC 305	GM-NAA I/O	Disk	Size of two refrigerators; weighing one ton; capacity is 5 MB, speed is 10 kB/s	–	Strong coupling with the operating system
1960s	IBM 360 series mainframes	IBM OS/360 series	Removable hard drives	Includes six 14-inch platters; capacity is 2.6 MB	Record-oriented file system	Associated with mainframe operating systems; accessed in record units
1970s	Minicomputers, workstations	Unix	Floppy, Winchester disk	Floppy disk: 8 inches, capacity of 80 kB; Winchester disk: the magnetic head is suspended above the platter	RT-11 FS, CP/M FS, System R, Oracle Release 1	RT-11: each file consists of a contiguous group of blocks; CP/M FS: ancestor of MS-DOS; relational databases handle structured data
1980s∼1990s	Personal computers	macOS, Windows, Linux	CD/DVD, removable floppy and disk, NOR flash, NAND flash	NOR flash: on-chip execution, high read performance, small capacity; NAND flash: high write/erase performance, large capacity	FatFs, NTFS, EXT, EXT2, MINIX V1 FS, DB2, PostgreSQL, MySQL	File systems optimized for disk; industrial grade open source database
2000∼2020	Smartphones	Apple iOS, Google Android	MMC card, SD card, USB	SD card includes encryption hardware and MMC; USB encapsulated flash chip	F2FS, JFFS2, NoSQL, NewSQL	File systems optimized for SSD; NoSQL handles semi-structured data
2020s	Human-cyber-physical, ubiquitous computing	Ubiquitous operating system	NVMe SSD, PM	NVMe SSD: more and deeper I/O queues; PM: byte-addressable, near DRAM performance, persistence	New storage systems (i.e., table storage)	Hardware feature-awareness; lightweight software stack; dynamic adaptability

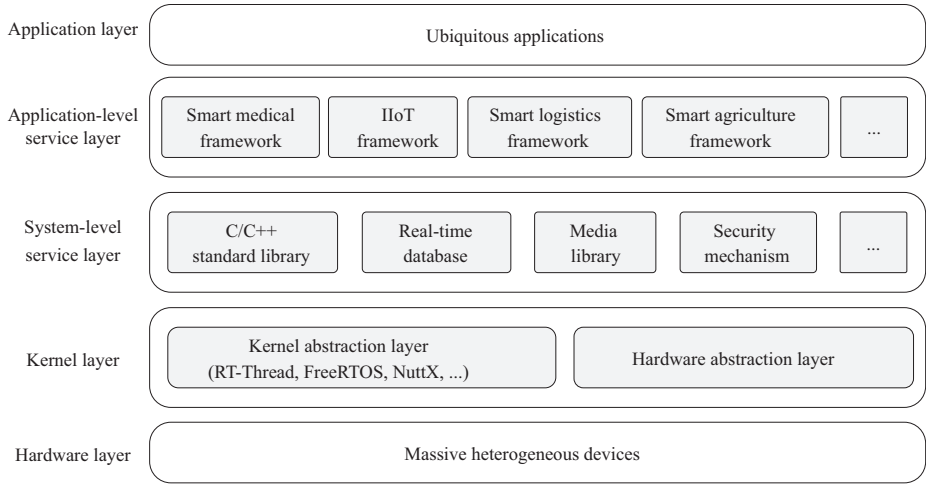

图 1 泛在操作系统的架构

Figure 1 Architecture of UOS

存储系统等, 与此同时也带来了一系列的新问题与新挑战.

3.1 泛在操作系统

当前操作系统难以满足人机物融合的泛在计算的环境多变性、广谱多样性、需求多样性和场景复杂性 [4]. 具体而言, 现有操作系统面临着 3 个缺陷. 一是在 “云 – 边 – 端” 乃至 “人” 的海量异构泛在化资源的管理方面存在不足; 二是现有操作系统技术体系碎片化严重, 不能够支撑场景多变的泛在计算环境; 三是无法应对开放环境下的安全可信挑战. 因此, 有必要结合泛在计算场景特性, “沉淀” 一类新型操作系统 [3].

梅宏教授在 “Toward ubiquitous operating systems: a software-defined perspective” [3] 中提出泛在操作系统的概念, 并将其视为人机物融合泛在计算时代操作系统发展的重要方向和形态. 自该概念提出以来, 泛在操作系统成为基础软件领域的研究热点, 现阶段仍处于探索成型期. 泛在操作系统需具备的基本能力是弹性部署, 即可以根据应用特征和设备的资源能力进行灵活裁剪以满足不同形态设备对操作系统的要求. 笔者认为分层架构是实现弹性部署的有效手段, 是泛在操作系统的基本形态. 图 1 展示了泛在操作系统的分层架构. 其中, 硬件层是泛在场景下海量的异构设备, 系统支持 ARM, RISC-V, x86 等主流体系结构; 内核层采用多内核设计, 内核抽象层负责屏蔽多内核差异并支持根据应用场景和设备资源情况进行定制化; 系统级服务层是系统核心能力的集合, 包括 C/C++ 标准库、轻量级数据库等; 应用级服务层是不同领域或者应用场景定制的特定框架, 比如面向智慧医疗、智能物流等场景的框架, 该层直接为上层的泛在应用提供服务. 系统级服务层和应用级服务层内部均是高度模块化的, 具备动态可裁剪性.

这种分层的形态可以追溯到 MINIX[3)]. MINIX 是经典的采用微内核架构的操作系统, 其中, MINIX3 被组织为内核层、设备驱动层、服务器层和应用层, 每一层执行定义明确的功能. Android 也采用了分层架构的思想, 自下而上依次是 Linux 内核、核心类库、应用程序框架层和应用层. 谷歌公司提出的 Fushsia OS[4)], 包含了 Zircon, Garnet, Peridot 和 Topaz 4 个层次. 国内的 AliOS[5)] 和

3) Andrew Stuart Tanenbaum. https://minix3.org/.
4) Google. https://fuchsia-china.com/.
5) Alibaba. https://www.alios.cn/.

HarmonyOS[6] 等新形态操作系统也都遵循分层的理念.

分层的架构凭借两方面优势将会成为泛在操作系统的基本形态. 首先, 内核之上的若干层可以按需弹性部署, 能够适配不同类别的硬件资源, 满足多样的功能需求; 其次, 上层应用使用下层提供的统一服务, 可以屏蔽下层的差异, 更好地实现高内聚、低耦合.

3.2 XiUOS: 一款面向工业物联网的泛在操作系统

XiUOS [5, 9] 是符合 3.1 小节所提分层架构的一个典型代表, 它是一款面向工业物联网 (industrial Internet of Things, IIoT) 应用场景的泛在操作系统. XiUOS 包含硬件层、内核层、框架层和应用层. 硬件层主要负责对多种处理器、传感器、通信器和控制器的管理, 并且这些物理器件之间采用解耦的模块化设计, 可以方便地以软件定义的方式屏蔽各种硬件的差异. 内核层主要完成单节点的系统管理功能, 支持根据应用场景或设备资源能力进行定制化内核. 系统级服务层框架层提供了若干重要的共性基础设施, 包括 C/C++ 基础库、轻量级实时数据库、安全系统等. 工业物联网应用框架层是 XiUOS 的核心, 包含 "感、联、知、控" 4 个模块 [9].

• 全面感知. XiUOS 的传感框架提供了一个通用的面向物理量的编程模型. 开发者可以方便地对传感器的数据进行解析, 采集及管理, 而无需关心底层设备差异.

• 泛在互联. XiUOS 支持节点自组织网络, 简化了复杂环境下的部署和管理.

• 实时认知. XiUOS 提供人工智能 (artificial intelligence, AI) 算法加速库、微型 AI 引擎等, 支持在节点端即可完成智能分析和认知, 为用户提供轻量化、易部署的工业智能方案.

• 精准控制. XiUOS 提供对各型生产设备、工业控制系统的远程实时监控和精准控制.

在应用层, XiUOS 与多个企业进行了合作, 已经在杭州市萧山区多个工业企业部署应用, 浙江大胜达包装股份有限公司基于 XiUOS 和自研智能感知终端构建了智能环境感知系统 [5].

4 面向泛在场景的结构化数据存储

人机物融合的泛在计算环境对软件适配性提出了更高的要求, 需要支持泛在计算模式的操作系统, 即泛在操作系统, 以有效管理海量的异构资源, 满足泛在化应用需求和场景的多样性. 存储管理是泛在操作系统的一项核心功能, 图 2 展示了存储管理在泛在操作系统架构下的定位, 如何基于泛在操作系统设计轻量化、高性能和动态可适配的存储系统, 是推进泛在操作系统发展的必要举措.

在泛在操作系统环境中, 面临着 "端 (终端设备) – 边 (边缘端) – 云 (云端)" 3 个层级的存储需求. 其中, 每个终端设备是一个配有处理器、控制器、RAM (random access memory)、ROM (read-only memory)、non-volatile storage (如 flash-based device) 以及若干个传感器的 "微型计算机"; 边端和云端的服务器具有较强算力, 硬件资源相对充裕, 可以配置新型存储硬件 (如, NVMe SSD, PM 等) 以满足高性能存储需求. 在 3 个层级上均存在大量的结构化数据 (指按照预定义的模型进行结构化, 或以预设方式组织的数据. 例如数据库中的记录、工业物联网中的监控数据等) 的存储需求. 通行的解决方案是在字节流的文件系统基础之上实现结构化存储. 然而, 在泛在操作系统下, 传统的结构化数据存储解决方案存在 I/O 放大严重、系统体量过大、软件栈冗余等问题, 无法满足泛在应用的存储需求, 下面从面向资源严格受限的端侧和基于新型高性能存储设备的边/云侧两个场景进行深入分析.

6) HUAWEI. https://developer.harmonyos.com/.

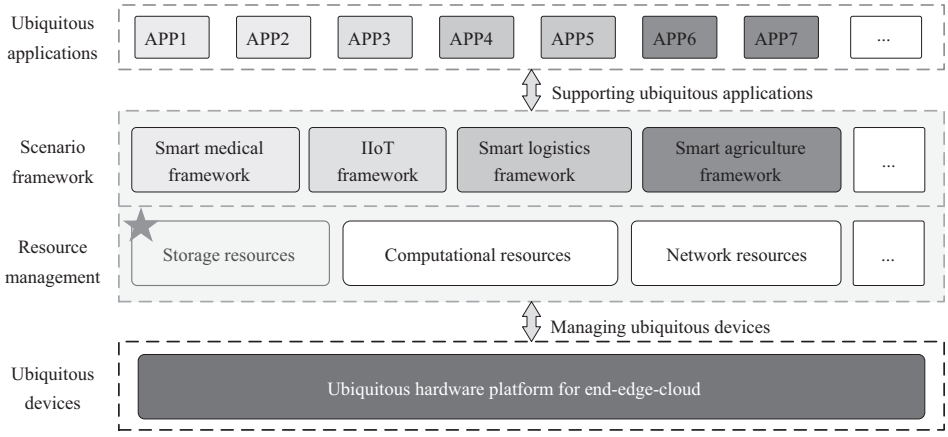

图 2 泛在操作系统架构下的存储管理定位

Figure 2 Storage management positioning in the architecture of UOS

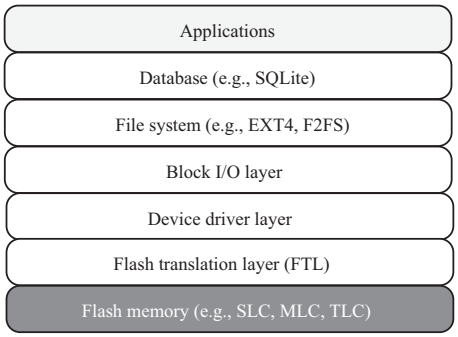

图 3 嵌入式设备的结构化数据存储栈

Figure 3 Structured storage stack of embedded devices

4.1 面向端侧的结构化数据存储

在人机物融合的泛在计算场景下, 面向端侧的结构化数据管理面临着诸多挑战. 一方面, 终端设备呈现出海量、异构、异质等特性, 这些设备的资源通常受到严格的限制, 不同设备之间的资源类型和容量各不相同, 使得存储管理的复杂度呈指数级增加 [3, 4, 9]. 另一方面, 泛在应用呈现更加多样化的特点, 具有明显的领域性和专用性, 为满足应用的个性化需求变得更加困难. 接下来, 本文首先分析端侧存储硬件设备的特点, 然后分析面向端侧的存储软件所面临的挑战.

终端设备一般采用基于闪存的设备, 如安全数字卡 (secure digital card, SD card) 和通用闪存存储 (universal flash storage, UFS), 进行低功耗、非易失性的数据存储 [10]. 出于考虑硬件成本和功耗, 它们通常配备有少量闪存芯片和极有限的内嵌 RAM [10, 11]. 此外, 闪存介质存在严峻的耐久性问题 [11]. 闪存单元具有有限次的擦写 (programming/erase, P/E) 操作, 即每一个闪存单元具有有限的寿命. 闪存单元在接近擦写次数极限时, 无法可靠存储数据状态. 因此, 为了延长闪存的使用寿命, 有必要最大程度地减少 I/O 放大.

I/O 放大严重. 如图 3 所示, 闪存之上按需堆叠的一系列软件构成了端侧设备的结构化数据存储栈. 为了提高终端设备上基于闪存的存储效率, 研究者们在存储栈的不同层次进行了研究, 比如在数据库层 [10~12] 或文件系统层 [10, 11, 13]. 本文测试了 SQLite 在 3 种流行的文件系统 (EXT4, FatFs 和

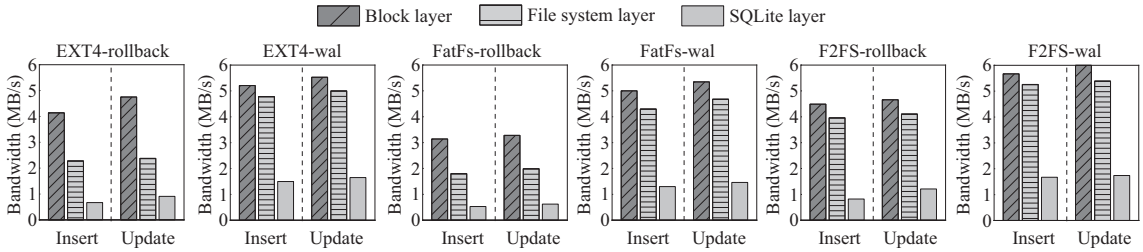

图 4　I/O 路径上的带宽跟踪

Figure 4　Bandwidth tracing on the I/O path

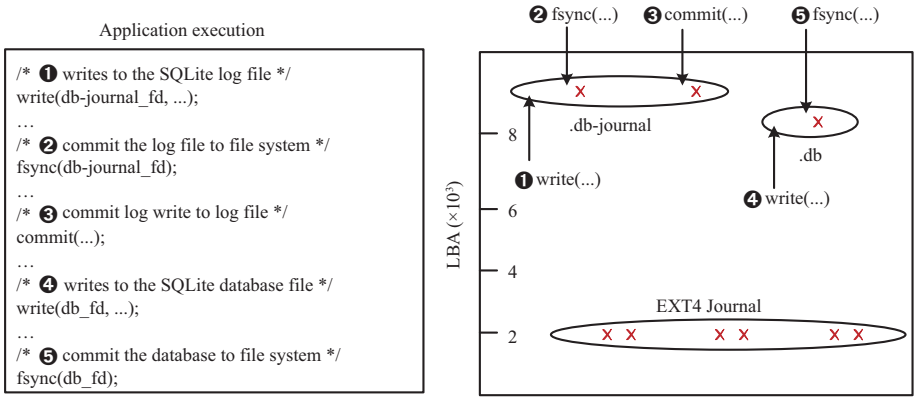

图 5　(网络版彩图) 插入操作期间的块 I/O

Figure 5　(Color online) Block I/O accesses in the insert operation

F2FS) 下的带宽性能 [14]. 如图 4 所示, 在执行插入操作期间, 收集了 SQLite 层、文件系统层和块层的带宽. 与块层相比, 文件系统层损失了 10.9%~48.6% 的带宽, 而数据库层相对于文件系统层进一步损失了 61.7%~78.8% 的带宽. 事实上, 文件是块的抽象集合, 而表是文件之上的抽象, 存在更多的抽象层时, 带宽的损失就会更大. 进一步, 为了更清楚地理解数据库与文件系统之间的交互, 笔者采用了一种更精细的方法来跟踪 SQLite 插入操作期间在块层产生的具体 I/O. 如图 5 所示, 可以观察到数据库的单个插入操作对闪存产生了 9 次块 I/O. 总的来说, 多层的抽象导致了严重的 I/O 放大, 这不仅降低了带宽利用率, 还降低了闪存的寿命. 堆叠式软件栈使得 I/O 路径很长, 造成大量冗余开销, 无法满足泛在化应用对 I/O 时效性的需求.

硬件资源受限. 终端设备的资源是严格受限的, 不同设备之间的资源差异很大, 例如 RAM 的容量从几 MB 到 GB 不等. 此外, 在特定应用场景中, 许多嵌入式系统需要进行优化或重新实现某些功能, 以满足实际需求. 然而, 当前的存储系统不具备模块化架构. 存储系统中的某些功能在某些应用中可能是不必要的, 这使得笨重的存储系统不适用于小型和低功耗的嵌入式设备, 而且升级或添加新模块变得困难, 需要付出大量的努力来重构甚至重新实现它们.

应用需求多样. 不同的泛在应用在一致性需求方面存在多样性, 因此并不存在一种通用的一致性策略适用于所有情况. 一般来说, 采用相对较弱的一致性策略会降低硬件资源消耗并提高性能, 相反, 更强的一致性保证则需要更多的资源投入并可能牺牲性能. 存储系统需要具备灵活的一致性机制, 以在满足不同泛在应用的一致性需求的前提下, 尽量降低资源消耗并提高性能.

针对泛在操作系统的特点以及在端侧存储管理上面临的挑战, Fan 等 [14] 提出了 LUNAR, 一个

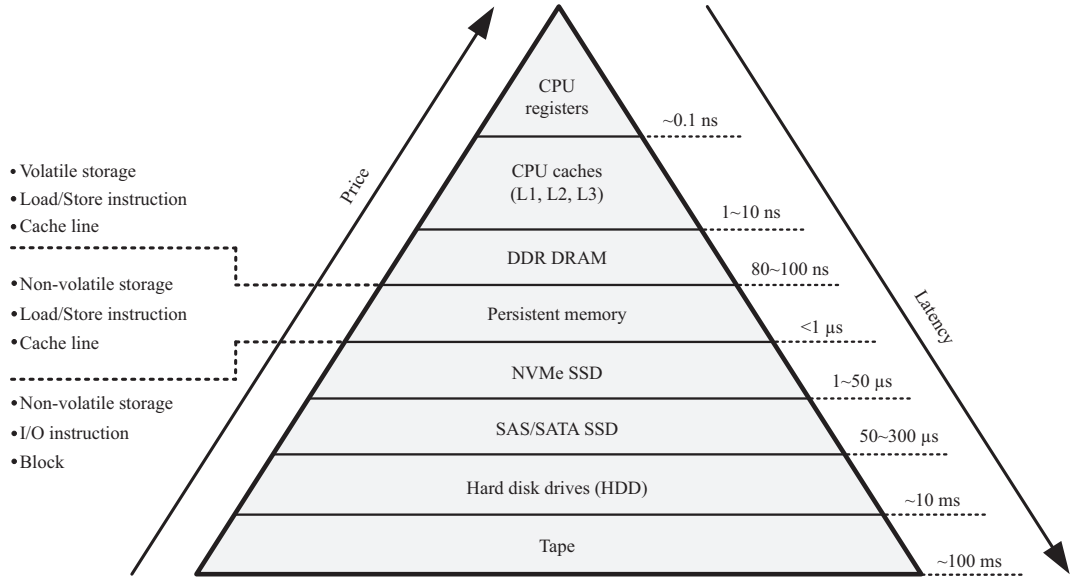

图 6 存储层次结构

Figure 6 Storage hierarchy

专为端侧设备设计的原生表存储引擎, 它具有如下 4 个设计点 (设计细节见 5.1 小节): (1) 轻量化. LUNAR 通过整合数据库和文件系统实现了跨层次设计. 它摒弃了文件抽象, 直接在设备上构建表结构, 同时提供了 SQL 兼容的 API. LUNAR 降低了 I/O 放大, 提高了带宽利用率, 并提供了更短的 I/O 路径. (2) 存储高效. 由于 LUNAR 构建在更低的层次 (设备驱动) 之上, LUNAR 拥有对存储设备的直接控制能力, 允许设计高效的数据布局和分配器, 以适应数据库访问模式. 具体地, LUNAR 采用了一种新颖的类型感知存储布局, 考虑了不同数据类型的访问模式, 以优化性能. 此外, 它使用了可变大小的块分配器来减少存储碎片, 最小化 RAM 和 I/O 带宽的浪费. (3) 模块化. LUNAR 采用了模块化设计. 具体地, LUNAR 包括 3 类模块: 基本模块、系统优化模块和表优化模块. 基本模块是必选的, 是表存储引擎的核心, 提供了表引擎所需的基本存储功能. 基本模块的资源占用非常小, 可以很好地适应资源有限的终端设备. 系统优化模块为系统中所有表提供服务. 此外, 每个表也可以独立选择表优化模块. LUNAR 的模块化设计, 可以适应资源有限的端侧设备, 也有助于模块的升级和扩展. (4) 灵活性. 端侧设备通常资源严格受限, 不同泛在应用对一致性的需求也不尽相同. LUNAR 在资源消耗和一致性之间进行权衡. 一般来说, 较低一致性产生较低的资源消耗和更好的性能. LUNAR 提供 4 种灵活的一致性模式, 在满足泛在应用的一致性需求的前提下, 尽量降低资源消耗并提高性能.

4.2 面向边/云侧的结构化数据存储

边/云侧的服务器具有较强算力, 硬件资源相对充裕, 可以配置新型存储硬件以满足高性能存储需求. 随着存储技术的发展, 新型非易失性设备正逐步替代传统的慢速存储设备, 成为新兴计算平台上的主流技术. 如图 6 所示, NVMe SSD 和 PM [15] 的出现填补了存储器金字塔. 本文将存储设备按照易失性分类, 可以分为易失性存储和非易失性存储, PM 成为了易失和非易失的分界线. 按照时延分类, 可以分为毫秒级、微秒级、亚微秒级、纳秒级存储. 表 2 对比了不同存储设备的性能和物理特征. 新型硬件的不断发展为面向泛在操作系统的数据存储带来了新的机遇, 同时也提出了新的挑战. 一方面, 借助新硬件有望打破目前的存储性能瓶颈; 另一方面, 存储软件需要做轻量化的处理, 同时结合新硬

表 2 存储设备的性能对比

Table 2 Performance comparison of storage devices

Characteristic	DRAM	NVDIMM-N	Optane DC PM	PCM	NVMe SSD	SAS SSD	Disk
Read latency (ns)	55	55	70	48	2×10^4	10^5	3×10^6
Write latency (ns)	55	55	150	160	2×10^4	3×10^5	7×10^6
Power consumption	Medium	Medium	Medium	Medium	Medium	High	Very high
Non-volatile	×	✓	✓	✓	✓	✓	✓

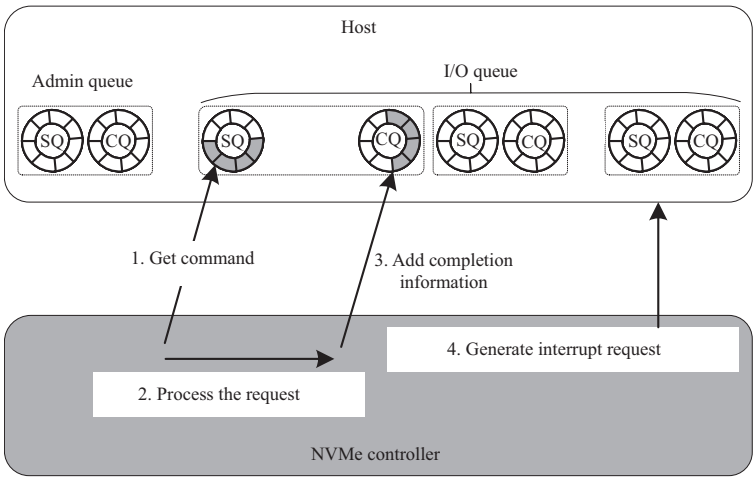

图 7 NVMe 中的多 I/O 队列

Figure 7 Multiple I/O queues in NVMe

件的特性以发挥其全部潜力. 接下来, 本文首先分析边/云侧新型存储硬件的特点, 然后分析现有存储软件所面临的挑战.

4.2.1 新型存储硬件

(1) NVMe SSD. 非易失性内存主机控制器接口规范 (NVMe) 是可扩展的主机控制器接口协议, 能够提供对非易失性存储设备的有效访问[1)]. NVMe 协议从传输协议角度对主机与硬件设备间数据传输效率进行了优化, 继而提高了带宽并降低了硬件延迟.

NVMe 协议和传统的适用于磁盘的高级主机控制器接口 (advanced host controller interface, AHCI) 协议有着诸多不同, 其核心在于队列的管理. NVMe 协议支持多个深度的 I/O 队列, 以提高数据处理的并行性. 队列数最多可以达到 64k, 深度最大为 64k[1)]. NVMe 协议中包含 Admin 队列和 I/O 队列共两种类型的队列. 每个 NVMe 控制器都对应于一对 Admin 队列和多对 I/O 队列. 其中, Admin 队列用于处理管理员命令, 包括 I/O 队列的初始化和命令控制 [16]. I/O 队列用于接收和处理来自主机系统的 I/O 命令. 每对队列包括一个提交队列 (submission queue, SQ) 和一个完成队列 (completion queue, CQ). SQ 和 CQ 都是位于主机系统中一段存储区域的环形缓冲区. 如图 7 所示, 在收到 I/O 命令之后, NVMe 协议要求首先在 SQ 上插入这个 I/O 命令. 当 NVMe 控制器完成此 I/O 命令时会在相关联的 CQ 中加入完成消息. 多个 I/O 队列能够并行地执行 I/O 命令使得 NVMe SSD 的带宽显著增加.

伴随着 NVMe SSD 的普及, 出现了一些针对 NVMe SSD 特性的研究工作. 文献 [16] 首次对 NVMe

驱动器进行了深入的性能分析, 将驱动程序插装与系统监控工具相结合, 展示了整个系统中 I/O 请求访问时间的细分, 此外, 还对 NVMe 驱动器的低延迟、高吞吐量特性的所有因素进行了详细、定量的分析. NVMeDirect [17] 提出了一个用户级 I/O 框架, 通过允许用户应用程序直接访问 NVMe SSD, 而无需任何硬件修改, 从而提高性能. 还有一些针对 NVMe 指令进行优化的工作, 比如减少远程访问的代价 [18]、减少内存占用 [19]、I/O 资源共享 [20]、一致性的优化方案, 以及在 Linux NVMe 驱动程序中引入了加权轮询 (weighted round robin, WRR) 支持 [21].

凭借其高性能优势, NVMe SSD 将逐渐成为泛在计算时代下的主要存储设备. 但在面向泛在操作系统的场景中, 结合泛在化应用的需求, 设计 NVMe SSD 硬件友好的存储系统仍存在诸多挑战.

(2) PM. PM 以高集成度、低功耗、可字节寻址、持久性和接近 DRAM (dynamic RAM) 的性能等特征, 为存储系统发展带来巨大机遇, 是未来存储器系统中一种很有前途的技术. 目前主流的 PM 包括 5 类, 分别是相变存储器 (phase change memory, PCM)、自旋矩存储器 (spin transfer torque RAM, STT-RAM)、阻变存储器 (resistive RAM, RRAM)、赛道存储器 (racetrack memory, RM) 和被 Intel 和 Micron 联合开发的 3D XPoint 技术 [15]. PM 的出现弥补了慢速持久存储 (即磁盘和固态硬盘) 和 DRAM 之间的鸿沟, 并将从根本上改变内存和存储系统的架构, 以满足日益增长的性能需求. 在 “端 – 边 – 云” 的泛在场景下, PM 主要被配置于边端和云端的服务器, 相比于传统的磁盘或者固态硬盘, PM 拥有一些独特的硬件特性, 在设计面向泛在操作系统的存储系统时需加以考虑才能充分释放其性能潜力.

随着 Intel Optane DC 持久内存的发布, 第 1 款非易失性内存开始商用, 图 8 展示它的内部体系结构, 理解它的性能特征对于构建高效的持久应用程序非常重要. 近几年来, 涌现了许多关于 PM 性能细致而深入的研究测试工作 [15, 22]. 总的来说, 因其表现出复杂的行为, 性能受到访问模式、访问粒度、设备内缓存和预取机制、非一致性存储器访问架构 (non uniform memory access architecture, NUMA) 等诸多因素的影响, 不应该简单地将 PM 视为较慢的、持久的 DRAM. 本文选取 4 个最重要的特征加以阐述. (1) 持久内存通常由于其物理特性而显示出非对称的 I/O 性能. 主要表现在两个方面. 首先, 对于读写 I/O, 读时延和 DRAM 相差 2∼3 倍, 写时延和 DRAM 差不太多, 这主要得益于 PM 的控制器配备了一个名为 XPBuffer 的写合并缓冲区, 用于掩盖写入延迟. 单根 PM 的最大读写带宽分别为 6.6 和 2.3 GB/s, 而 DRAM 的读写带宽差距要小得多 (约 1.3 倍) [15]. 其次, 在顺序和随机 I/O 上, PM 的顺序 I/O 带宽最大是随机 I/O 的 3.5 倍, 这主要是因为 PM 内部存在一个名为 XPPrefetcher 的预取器以加速顺序 I/O 的性能 [15]. (2) CPU 访问粒度是 64 字节的缓存行, 而 PM 以 256 字节的粒度, 被称为 XPLine, 来访问数据. 这两者的访问粒度不匹配, 如果数据访问粒度小于 256 字节, 则会导致读写放大 [15]. 为了解决由于访问粒度差异而引发的读写放大问题, PM 控制器配备了 XPBuffer, 以支持读 – 改 – 写操作. 此外, 连接 PM 和集成内存控制器 (integrated memory controller, iMC) 的 DDR-T 协议支持异步存储, 以隐藏长写入延迟, 并针对小型写操作进行合并来减少 I/O 放大. (3) 随着 NUMA 架构的流行, 研究远端 NUMA 访问的性能影响是很重要的. 文献 [15] 表明, NUMA-Local 写 PM 的峰值性能约为 12.5 GB/s, 但写远端 NUMA 的 PM 带宽峰值约 7 GB/s. 跨越 NUMA 访问导致带宽下降, 主要有两方面原因. 首先, 若写远端 NUMA 的 PM, 那么所有数据都需要通过高速通道互联 (ultra path interconnect, UPI), 阻塞写操作的延迟增加, 降低了带宽. 其次, clwb 甚至一些 ntstore 的行为均类似于缓存行的读 – 改 – 写操作. 这种读 – 改 – 写本质上是一种混合的读写工作负载, 因此降低了带宽利用率. (4) PM 支持跨双列直插内存模块 (dual in-line memory module, DIMM) 的数据交错 [15]. 数据交错旨在通过把顺序数据访问散布在多个 DIMM 中来增加读写整体吞吐量, 与独立磁盘冗余阵列 (redundant array of independent disks, RAID) 中的 RAID0 类似. 这样可以更好地进行并行访问, 获得

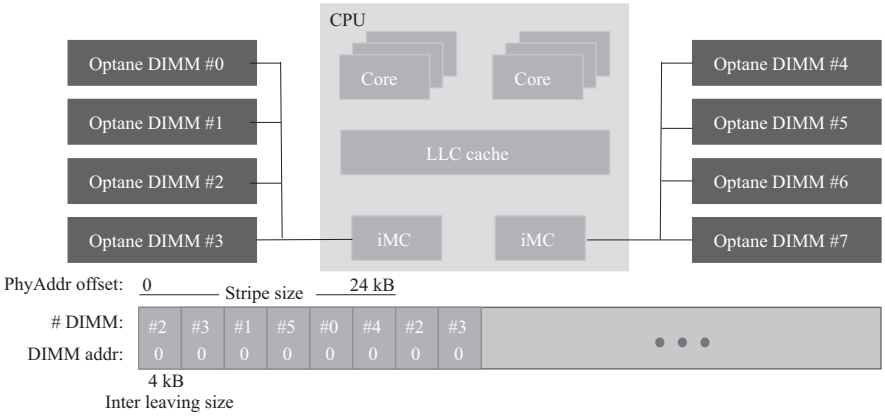

图 8 Intel Optane PM 的微体系结构

Figure 8 Microarchitecture of Intel Optane PM

更高的吞吐量. 如图 8 所示, 在 PM 中, 数据通常以 4 kB 的块粒度进行交错, 将每个连续的 24 kB 的块分布在所有 6 个可用 DIMM 中以最大化并行性能.

随着对 PM 硬件特性的深入探索, 出现了许多与设备特性相结合的研究工作. CCEH [23] 是基于 PM 的持久性哈希 (Hash), 其哈希桶的大小设置为 256 字节以最小化 I/O 放大. Flatstore [24] 是一个基于 PM 的采用日志结构的高效键值存储引擎, 采用压缩日志技术将多条小日志条目打包到 256 字节的批次中进行处理. 文献 [22] 详细分析了 PM 内部的读写缓冲的设计, 以及它与 CPU 缓存、DDR-T 协议的交互. Nap [25] 将并发 PM 索引转换为支持 NUMA 的对应索引. Nap 使用 NUMA-aware 层来吸收热点数据的访问, 在不引发额外的本地 PM 访问的情况下消除了远程 PM 访问. Xu 等 [26] 为 NOVA 文件系统 [27] 提供 NUMA 感知接口. 这些研究工作仅考虑到部分硬件特性, 需进一步全面考虑 PM 特性以发掘硬件性能潜力.

综上所述, 这些高性能的新硬件为数据存储与管理带来了新的生机, 有望突破现有的性能瓶颈. 然而, 既有的传统软件系统往往忽略了或者仅考虑到部分新硬件的新特性, 这导致直接将现有存储系统部署在新硬件上时, 无法充分发挥新硬件的全部潜能. 独特的硬件特性为系统软件的设计带来了新的挑战, 比如如何高效利用 NVMe SSD 的多深 I/O 队列以及处理好硬件 I/O 队列和 CPU 核心之间的关系、怎样尽可能降低 PM 中因访问粒度不一致而导致的 I/O 放大、如何利用 PM 的数据交错特征以及怎样有效发掘设备内预取器的效能等. 因此, 有必要结合新硬件的特性对现有存储软件进行革新, 以满足面向泛在操作系统新场景下的存储需求.

4.2.2 存储软件栈

在边端和云端, NVMe SSD 和 PM [15] 等新型存储设备正在逐步替代传统的慢速设备. 当在现有系统上直接应用这些高性能的存储硬件时, 传统软件技术难以充分释放新硬件的全部潜力. 这是由于随着硬件技术的不断进步, 存储设备在硬件延迟上已经有了数量级的提升, 从而使传统的厚重系统软件开销比重急剧增加, 无法释放硬件的全部红利, 软件成为了制约系统性能的一大瓶颈.

传统的存储系统依赖于多层的堆叠式软件栈, 使得其很难充分发挥新硬件的潜能 [28]. 如图 9 所示, 通常情况下, 对结构化数据存储而言, 传统的数据库运行于文件系统之上, 且其软件栈从上到下由数据库层、虚拟文件系统层、页缓存层、文件系统层、操作系统层 (以块设备层为主) 和设备驱动层组成, 读写请求要经过这几个不同层次的流转才能最终到达存储设备. 因此, 不同层级冗余的功能, 比如,

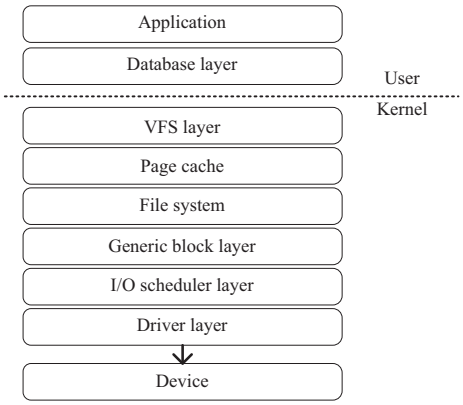

图 9 传统结构化数据存储栈

Figure 9 Traditional storage stack for structured data

缓存、缓冲区、数据拷贝、队列等, 将不可避免地造成一些额外开销. 在磁盘时代, 相较于毫秒级别的硬件延迟, I/O 路径上的瓶颈是由硬件引起的, 这使得这些厚重的软件开销可以被忽略. 当 NVMe 设备将硬件延迟减少到几十微秒以至于和软件开销处于同一延迟级别时, 软件栈的开销可以高达总延迟的 70% [29], 成为了影响性能的关键因素. 在这种情况下, 如果仍然忽略软件开销, 将无法充分释放新硬件的性能潜力, 从而导致存储系统难以实现最佳的运行时性能. 从图 10 中可以看出, 在同步和异步 I/O 中, NVMe SSD 上的软件开销都因过度冗余而占有很大的比重. 此外, 由于数据库建立在文件系统之上, 很难结合硬件特性来设计优化存储布局, 以提升数据库的性能.

随着 NVMe 设备的兴起, 涌现出了许多针对 NVMe SSD 的研究工作. Leanstore [30] 是一个基于 NVMe 设备的数据库, 利用 NVMe 设备的多 I/O 队列设计了并行的数据处理机制. KVell [31] 是基于 NVMe SSD 的键值存储系统, 数据存储在 SSD 上时不进行排序, 并且采用了无共享架构, 以避免同步开销. PA-Tree [32] 采用了一种新的轮询模式异步执行范式. PA-Tree 能够充分利用 NVMe 设备的内部并行性, 而无需承受与多线程相关的巨大开销. URFS [33] 是一种基于 NVMe 的用户态文件系统, 结合了 NVMe 协议和闪存介质的特性, 通过去除日志、多队列分离等技术手段提升系统的性能. 文献 [34] 观察并分析了在 NVMe SSD 上采用日志记录的文件系统中性能波动现象, 提出了 MIM. MIM 基于 NVMe 设备的多队列实现高效的写入合并. 这些研究工作具有一定的借鉴和指导作用, 但它们仅对传统存储栈进行了单一层级的优化, 比如在数据库层或文件系统层, 缺乏从软件全栈的角度进行系统性的精简. 它们仍然依赖于堆叠式的厚重软件栈, 无法充分释放新型存储设备的性能潜力.

PM 和 NVMe SSD 的硬件特性有着诸多不同. PM 提供接近 DRAM 的性能和字节可寻址的数据持久性 [15]. 如图 11(a) 和 (b) 所示, 基于 PM 的数据库可以通过两种方式来访问 PM: POSIX 系统调用和依赖于文件系统的内存映射 [35,36]. 内存映射方法固然减少了用户/内核上下文切换和虚拟文件系统开销, 但其仍依赖于底层的 PM-aware 文件系统来提供所需的映射文件, 这种抽象并不能完全释放 PM 的性能. 首先, 在读写映射文件时, 它们不仅更新表的元数据, 还更新相关映射文件的元数据. 其次, 一致性处理可能是冗余的. 例如, NOVA [27], KucoFS [37] 和 SplitFS [38] 提供了 mmap 操作的原子性, 而基于 PM 的数据库可能额外使用 PMDK[2)] 等库来保证一致性 [39]. 最后, 表级的每个操作都要经历两个映射, 从逻辑表偏移量到逻辑文件偏移量, 然后到 PM 的物理位置, 最近的研究表明, 映射的成本是十分巨大的 [39,40]. 总的来说, 现有数据库的存储引擎有两种使用 PM 的方式, 但在这两种方式中, 表均建立在文件的抽象之上. 多层的抽象一方面导致软件的冗余, 另一方面也使得存储引擎无法

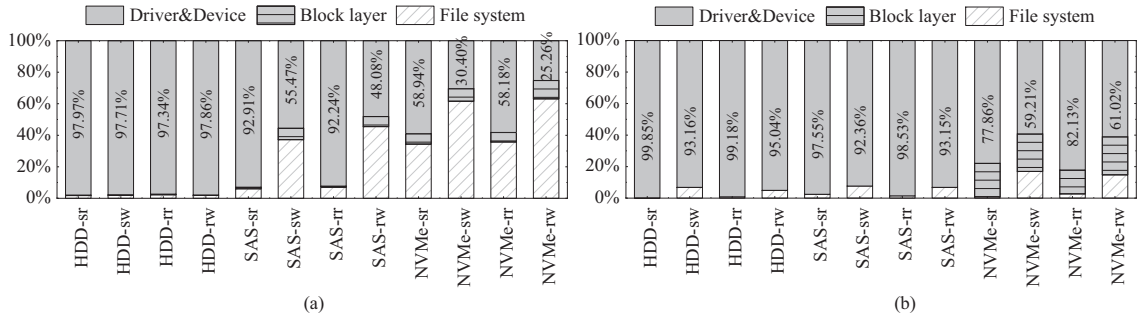

图 10　软件栈开销的分解

Figure 10　Breakdown of software stack overhead. (a) Synchronous I/O; (b) asynchronous I/O

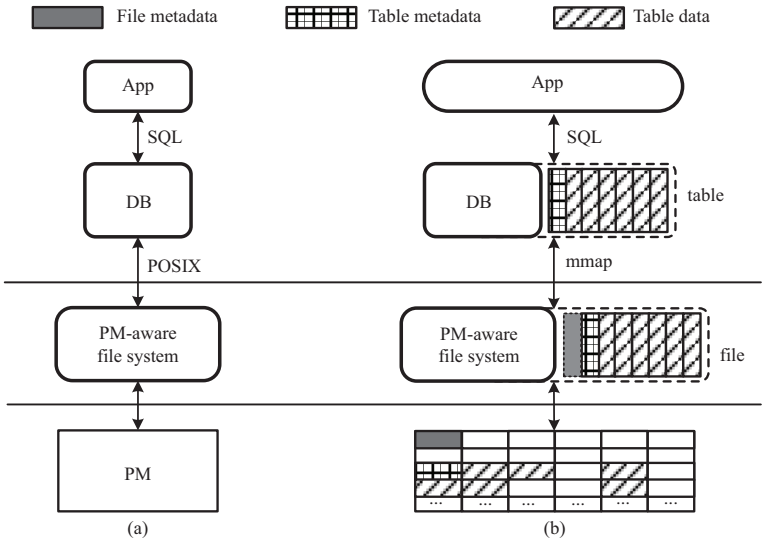

图 11　PM 的不同访问方式

Figure 11　Different access manners of PM. (a) FS + POSIX; (b) FS + mmap

结合硬件特性来设计存储布局, 从而限制了 PM 性能的释放.

近年来, 人们对基于 PM 的文件系统和数据库进行了大量的研究. 研究者们提出了许多 PM-aware 的文件系统, 如 PMFS [41], NOVA [27], SplitFS [38] 和 WineFS [42], 并利用 PM 的可字节寻址能力消除面向块的存储成本. 这些系统还支持创建内存映射文件, 并通过 CPU 的 load/store 等指令 [15] 访问 PM 设备. 此外, 也有一些针对数据库的研究工作. 文献 [43~48] 均是基于 DAX 感知[7]的文件系统, 实现了高性能的内存语义的访问. 它们使用 mmap 系统调用将文件映射到特定的虚拟内存地址, 并构建用户空间编程库, 通过 CPU 的 load/store 等指令访问 PM 设备. 然而, 这些系统仍然依赖于文件的抽象, 仅在数据库层或者文件系统层进行单层优化. 这种优化方式使得两个层次的优化相互割裂, 彼此互不感知, 无法充分发挥新型存储设备的性能潜力. 因此, 需要从全栈的角度进行整合优化, 通过轻量化存储栈, 更好地释放新型存储设备的性能潜力, 进而提高系统的整体性能和效率.

综上分析, 在泛在计算场景中, 现有的边/云侧的结构化数据存储方案存在诸多不足. 首先, 随着新型存储设备的硬件性能的提升, 厚重堆叠式软件栈成为了制约系统性能的主要因素. 其次, 既有的

7) Direct Access. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

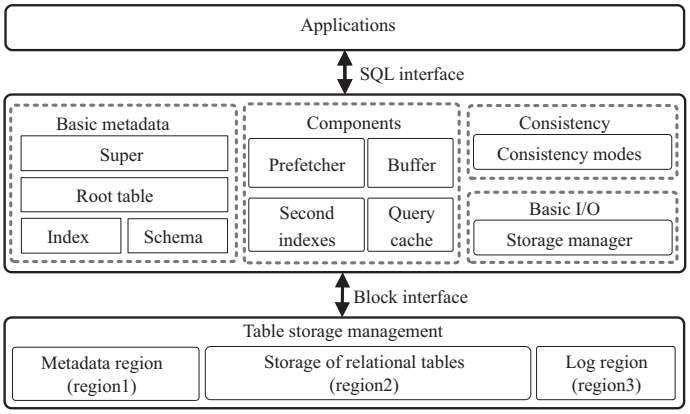

图 12 LUNAR 的架构

Figure 12 Architecture of LUNAR

面向结构化数据的存储系统往往构建在文件系统之上, 有意或无意地忽略新硬件的部分特性, 导致无法充分发挥新硬件的全部潜能. 最后, 针对结构化数据存储, 当前业界研究工作主要分别集中在数据库层和文件系统层. 然而, 这两方面的优化相互割裂, 彼此互不感知, 局限在单一层次优化存储栈并不能得到最大的性能释放. 总之, 现有软件栈已不能满足泛在操作系统的存储需求, 迫切需要从系统全栈的角度重新构建轻量级的存储栈, 优化软件架构, 提升系统整体性能.

5 面向泛在操作系统的表存储

对于泛在操作系统下的结构化数据存储, 传统的解决方案存在 I/O 放大严重、一体化的系统体量过大、未能充分结合存储硬件特性以及软件栈冗余等问题, 不能满足泛在应用的存储需求. 根据当前泛在操作系统生态构建的发展趋势, 综合考虑 “端 – 边 – 云” 泛在计算场景, 亟需凝练面向泛在操作系统的结构化存储关键技术. 笔者提出原生表存储系统来负责泛在操作系统中的存储管理, 接下来从资源严格受限的端侧和基于新型高性能存储设备的边/云侧两个场景进行深入分析.

5.1 面向端侧的原生表存储引擎

5.1.1 系统设计

如 4.1 小节所述, 在泛在计算场景下, 面向端侧的结构化数据管理面临 3 个主要挑战, 包括 I/O 放大严重、硬件资源受限以及应用需求多样化的 3 个挑战. Fan 等提出了 LUNAR [14], 一款专为终端设备的原生表存储引擎, 旨在抽象数据库表格的数据服务, 其关键在于去除了文件抽象, 并在数据库和文件系统之间进行跨层次设计. LUNAR 的架构图如图 12 所示, 具备以下 4 个设计特点.

轻量化. 图 4 表明, 基于传统文件系统的数据库由于多层抽象而具有较低的带宽利用率. 为了解决这个问题, LUNAR 通过整合数据库和文件系统层实现了跨层次设计. 它消除了文件抽象, 直接在设备上存储表结构, 同时提供了 SQL 兼容的 API, 如创建表格、插入、查询和更新. 图 13 展示了传统的 I/O 系统和 LUNAR 的 I/O 系统. 尽管这种方法引入了一些维护复杂性, 但笔者认为这是值得的, 因为它显著提高了带宽利用率, 并提供了更短的 I/O 路径. 此外, 由于 LUNAR 构建在更低的层次 (设备驱动) 之上, LUNAR 拥有对存储设备的直接控制能力, 允许设计高效的数据布局和分配器 (具体细节如下), 以适应数据库访问模式.

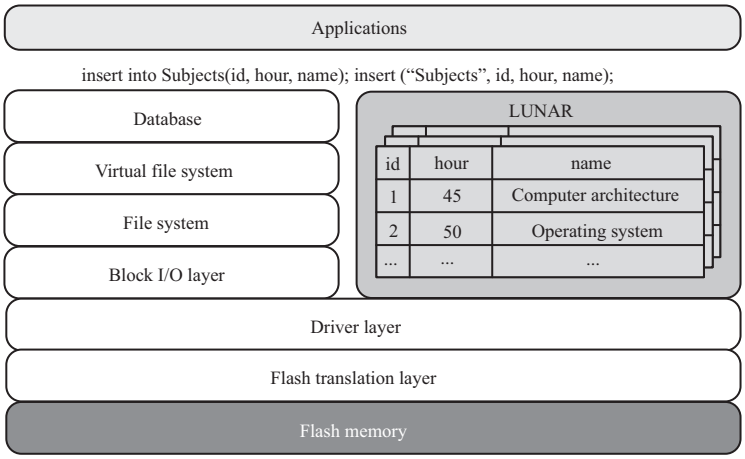

图 13　传统的与 LUNAR 的 I/O 系统

Figure 13　Traditional and LUNAR's I/O systems

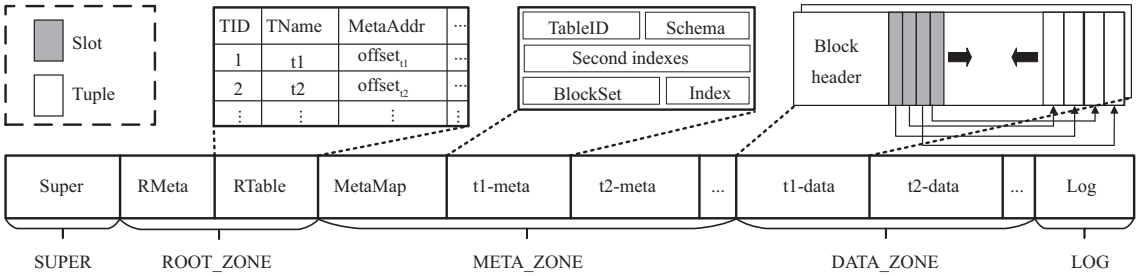

图 14　LUNAR 的存储布局

Figure 14　Storage layout of LUNAR

存储高效性. LUNAR 采用了一种新颖的类型感知存储布局, 考虑了不同数据类型的访问模式, 以优化性能. 此外, 它使用了可变大小的块分配器来减少碎片化, 最小化 RAM 和 I/O 带宽的浪费.

图 14 展示了终端存储设备上 LUNAR 数据结构的高级布局. LUNAR 将存储空间划分为 5 个 zone, 包括 SUPER, ROOT_ZONE, META_ZONE, DATA_ZONE 和 LOG. 进一步, 这 5 个 zone 根据访问特性分为 3 个不同的 region. region1 包括 SUPER, ROOT_ZONE 和 META_ZONE, 用于存储通常较小但经常随机访问的元数据, 以控制存储碎片. DATA_ZONE 属于 region2, 采用可变大小块分配器. region3 存储 LOG, 有助于提高日志记录效率和避免干扰前台任务. 不同数据类型根据访问特性分类到不同 region, 提高了处理效率并减少了区域之间的干扰, 从而提升系统性能.

与传统的定长块分配器不同, LUNAR 采用可变大小的块分配器, 以减少内存碎片并提高端侧设备上内存的利用率. LUNAR 将分配单元大小与设备提供的最小原子操作单位对齐, 数据在存储设备和 RAM 之间以扇区粒度传输, 减少了 RAM 和 I/O 带宽的浪费. 如图 15 所示, region2 的块按大小分类组织. 每个类包含两种类型的块: 管理分配状态的位图块和存储数据的数据块. 类别 0 包含最小的块, 如 512 字节, 后续的类包含大小指数级增长的块, 即在类 i 的块大小的基础上, 类 $i+1$ 的块大小增加了一倍, 默认的增长因子为 2. 特别地, 将增长因子设置为 1 则是传统的固定块大小分配器.

模块化. 为解决当前存储系统的模块化不足以及无法适应小型和低功耗端侧设备的问题, LUNAR 采用了模块化设计. 如表 3 所示, LUNAR 包括 3 类模块: 基本模块、系统优化模块和表优化模块, 其

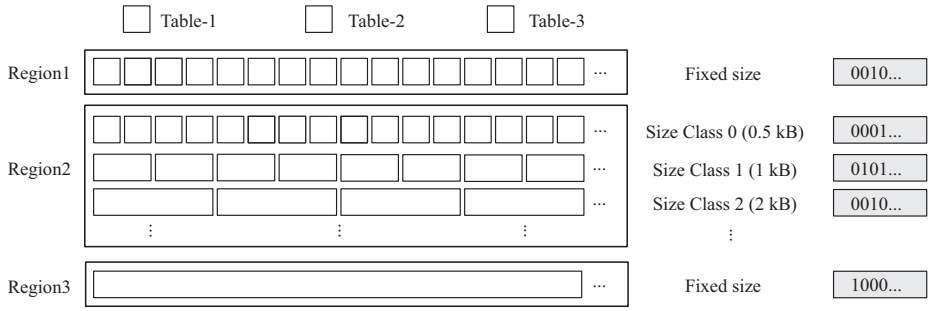

图 15 可变大小的块分配器

Figure 15 Variable-size block allocator

表 3 LUNAR 实现的模块

Table 3 Modules implemented by LUNAR

Module		ROM (kB)	RAM (kB)
Basic	Device management	20.150	0.203
	RAM management	12.358	0.114
System optimization	Prefetcher	2.834	0.025
	Buffer	1.732	0.023
Table optimization	Secondary index (B+Tree)	28.054	0.024
	Query cache	4.539	0.056

表 4 LUNAR 的一致性模式

Table 4 Consistency modes of LUNAR

Mode	Sync metadata	Metadata	Order	Sync data	Data
Disorder	×	✓	×	×	×
Metadata	×	✓	✓	×	×
Data	×	✓	✓	×	✓
Full	✓	✓	✓	✓	✓

中基本模块是必需的, 后两者是可选的. 基本模块的资源占用非常小 (ROM 32.51 kB, RAM 0.32 kB), 可以很好适应资源有限的终端设备. 基本模块是表存储引擎的核心, 提供了表引擎所需的基本存储功能, 包括 RAM 管理和基本的 I/O 管理. 系统优化模块为系统中所有表的服务, 如缓存管理器和预取器. 每个表可以独立选择表优化模块, 包括查询缓存模块和二级索引模块. LUNAR 的模块化设计, 可以适应资源有限的终端设备, 也有助于模块的升级和扩展.

灵活性. 终端设备通常资源严格受限, 不同泛在应用对一致性的需求也不尽相同. LUNAR 在资源消耗和一致性之间进行权衡. 一般来说, 较低一致性产生较低的资源消耗和更好的性能. 如表 4 所示, LUNAR 提供 4 种一致性模式: disorder, metadata, data 和 full.

在 disorder 模式中, LUNAR 只维护元数据日志. 这种模式一致性最弱, 适用于运行在资源极度有限的端侧设备, 且一致性需求低的泛在应用. 该模式下, 如果系统在元数据日志持久化后崩溃, 可能导致恢复后检索到垃圾数据. LUNAR 在 disorder 模式的基础上确保 metadata 模式 (默认). metadata 模式强制执行一个顺序约束, 要求必须在写入元数据日志之前完成与事务相关的数据写入. 此模式保

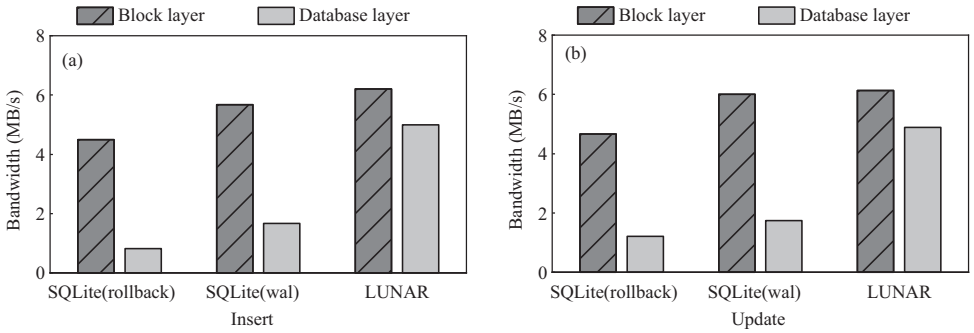

图 16 Mobibench 在带宽方面的性能表现

Figure 16 Mobibench performance on bandwidth. (a) Insert workload; (b) update workload

证了数据和元数据的持久化顺序, 以防止检索到垃圾数据. 由于不记录数据日志, 如果系统在写入数据时崩溃, 不会影响系统一致性, 但数据操作可能只完成部分. data 模式在 metadata 模式基础上提供一致性保证. data 模式不仅记录元数据日志, 还记录数据日志, 因此不会出现部分数据操作. full 模式提供最强的一致性保证. 在 data 模式基础上, full 模式还确保所有操作都是同步的.

5.1.2 实验分析

为了验证 LUNAR 的高效性, 本文在一块 Kendryte K210 开发板上进行了实验[8]. 该平台配备有 2 核 64 位 RISC-V 处理器、6 MB 内存和 32 GB SD 卡. 所有系统运行在 XiUOS 之上 [5, 9], 其中内核采用 RT-Thread. 本文运行了 Mobibench, YCSB (Yahoo! cloud serving benchmark) 和 SQLite-bench. Mobibench 使用 SQLite 作为测试数据库, 包含 10000 行记录 (每行 4 kB), 并执行同步的插入/更新操作. 对于 YCSB, 我们变化了两个参数: (1) 读/写比例: 只写 (100% 写入)、读重 (90% 读取/10% 更新)、平衡 (50% 读取/50% 更新), 以及更新重 (10% 读取/90% 更新); (2) Zipf 分布的偏斜度: 无 (偏斜度 = 0)、低 (偏斜度 = 0.5) 和高 (偏斜度 = 1). 对于 SQLite-bench, 我们比较了系统在读取和写入工作负载下的性能, 包括随机和顺序访问模式. 此外, 将 LUNAR 与广泛使用的 SQLite[9] 以及最新的数据库 FlashDB[10] 进行了比较. SQLite 是一款轻量且高速的数据库引擎, 已成为目前最广泛部署的数据库引擎. FlashDB 专注于为嵌入式产品提供数据存储解决方案.

(1) 基本性能测试. 本文使用 Mobibench 分析数据库如何利用底层存储设备: 执行了 100k 个 SQL 请求, 并测量了两种带宽类型, 包括数据库层 (即 SQLite 和 LUNAR) 和块层带宽. 如图 16 所示, 在使用 SQLite 执行插入操作时, 与块层相比, 带宽损失为 70.5%~81.7%. SQLite 的显著带宽损失主要是由于文件系统和 SQLite 之间协调不足的相互作用引起的, 由于多层抽象而导致的严重 I/O 放大. 相比之下, LUNAR 相对于块层, 实现了有 79.8%~81.6% 的带宽利用率. LUNAR 高带宽利用率主要得益于其消除了文件抽象, 大大减轻了 I/O 放大.

(2) 宏观基准测试. 本文运行了 YCSB 基准测试, 针对不同的读/写比例和偏斜度. 如图 17 所示, 在各种偏斜度的只写工作负载下, LUNAR 的平均延迟分别为 SQLite 和 FlashDB 的 49.7% 和 57.2%. 此外, 本文使用 SQLite-bench 来测试在随机和顺序模式下读取和写入工作负载的性能. 如图 18 所示, 在顺序和随机写入下, LUNAR 的吞吐量是 SQLite 和 FlashDB 的 1.3~4.1 倍. LUNAR 的高性能源于

8) Max Bit. https://wiki.sipeed.com/hardware/en/maix/index.html/.
9) SQLite. https://www.sqlite.org/.
10) FlashDB. https://github.com/armink/FlashDB/.

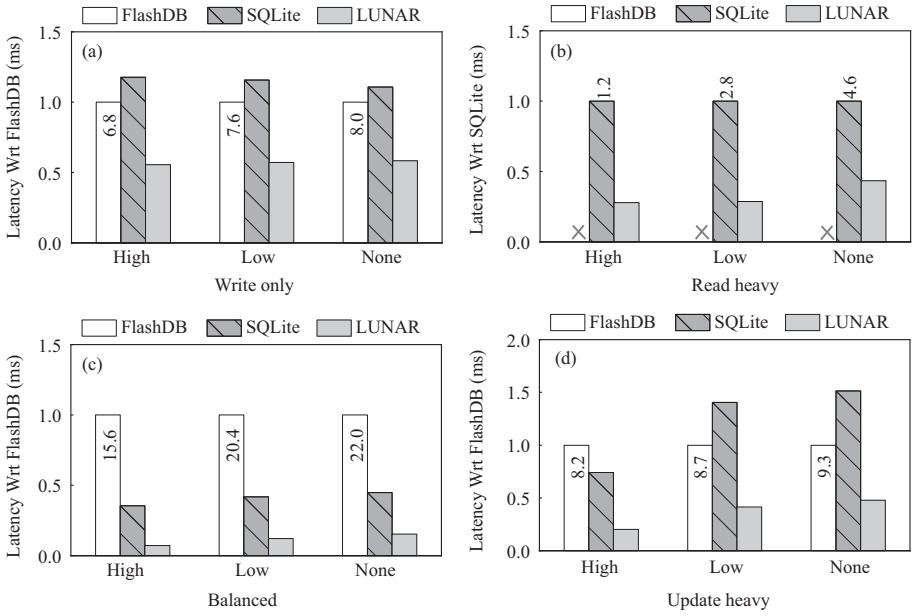

图 17 YCSB 在时延方面的性能表现

Figure 17 YCSB performance on latency. (a) Write-only workload; (b) read-heavy workload; (c) balanced workload; (d) update-heavy workload

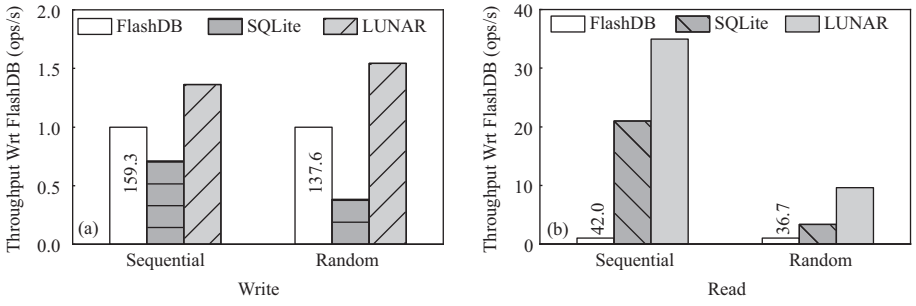

图 18 SQLite-bench 吞吐量性能

Figure 18 SQLite-bench performance on throughput. (a) Sequential write and random write; (b) sequential read and random read

两个主要因素. 首先, LUNAR 采用跨层设计, 通过消除文件抽象和建立更短的 I/O 路径, 减少了软件冗余. 相比之下, SQLite 和 FlashDB 的表建立在文件抽象之上, 导致了许多小、随机和同步 I/O. 其次, LUNAR 仅维护表元数据, 而 SQLite 和 FlashDB 维护表和文件的两种元数据, 加剧了 I/O 放大.

(3) 微观基准测试. 表 5 显示了几个系统的 ROM 和 RAM 使用情况. LUNAR (basic) 仅启用基本模块. 与 FlashDB 相比, LUNAR (basic) 具有更丰富的表级存储能力, 其 ROM 和 RAM 消耗分别减少了 8.2% 和 62.5%. 与 SQLite 相比, ROM 和 RAM 分别减少了 91.2% 和 95.6%. 对于启用了基本模块以及所有系统级和表级优化模块的 LUNAR (all) 来说, 相对于 SQLite 的 ROM 和 RAM 消耗分别降低了 81.1% 和 93.9%. 总之, LUNAR 集成了数据库和文件系统层, 导致资源消耗更低. 此外, 由于其模块化架构, LUNAR 可以适应极度资源有限的嵌入式设备, 有助于升级现有模块并添加新模块, 以满足不同应用的存储需求.

表 5 资源消耗对比

Table 5 Comparison of resource consumption

System	ROM (kB)	RAM (kB)
LUNAR (basic)	32.508	0.317
LUNAR (all)	69.667	0.445
VFS+FatFs	29.745	0.846
FlashDB+VFS+FatFs	35.417	0.847
SQLite+VFS+FatFs	369.227	7.293

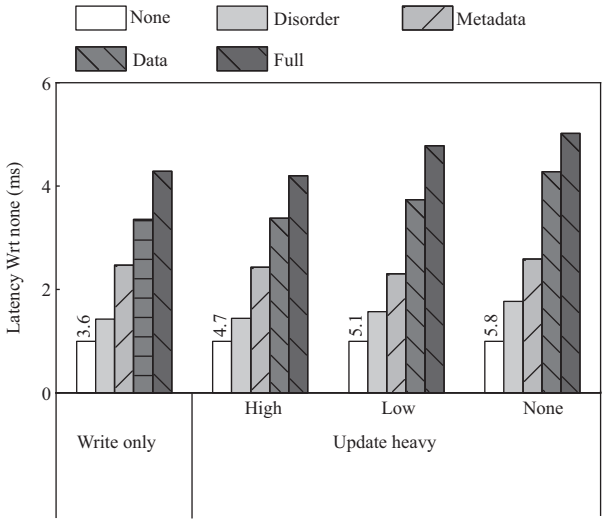

图 19 不同一致性模式下的性能

Figure 19 Performance with different consistency modes

本文使用 YCSB 来研究不同一致性模式对性能的影响. 如图 19 所示, None 表示在 LUNAR 中未启用日志. 与 None 相比, 在只写工作负载下, disorder, metadata, data 和 full 的延迟分别增加了 30.1%, 59.5%, 70.2% 和 76.7%. 总的来说, 对于这 4 种一致性模式, disorder 提供了最佳性能, 并且资源消耗最少, 因为它不需要同步持久性或数据日志记录, 同时提供了最弱的一致性保证. 与 disorder 相反, full 提供了最强的保证, 但消耗了最多的存储资源, 并且性能最差.

5.2 面向边/云侧的原生表存储引擎

边/云侧的服务器具有较强算力, 可以配置新型存储硬件以满足高性能存储需求. 如 4.2 小节所述, 在边/云侧部署新型存储设备既带来了机遇, 也带来了新的挑战. 一方面, 借助新硬件有望打破目前的存储性能瓶颈; 另一方面, 存储软件需要做轻量化的处理, 同时结合新硬件的特性以发挥其全部潜力. 接下来, 本文首先探讨边/云侧硬件友好性的设计, 然后介绍面向新型存储设备的轻量化表存储, 最后分别对基于 NVMe SSD 和 PM 的两款原生表存储引擎进行实验分析.

5.2.1 硬件友好性设计

NVMe SSD 和 PM 等新型高性能存储设备凭借其性能优势会逐渐成为泛在计算时代的主流. 面向泛在操作系统的存储系统在管理这些新型设备时需考虑新硬件的新特性, 以释放硬件全部潜能, 满

Algorithm 1 Queue allocation

1: $Q_{\min} \leftarrow$ qpair[0];
2: $S_{\min} \leftarrow \sum_{j=1}^{\text{qstate[0].length}}$; ($\lambda_j$·Size$_j$);
/* Query the load of each queue in turn to obtain the most idle I/O queue */
3: **for** i = 1 to SYSTEM_IO_QUEUES **do**
4: score $\leftarrow \sum_{j=1}^{\text{qstate}[i].\text{length}}$;
5: **if** score $< S_{\min}$ **then**
6: $S_{\min} \leftarrow$ score;
7: $Q_{\min} \leftarrow$ qpair[i];
8: **end if**
9: **end for**
/* Append the I/O task to $Q_{\min}$.SQ */
10: $Q_{\min}$.length $\leftarrow$ $Q_{\min}$.length + 1.

足泛在应用的存储需求.

NVMe SSD 支持高达 64k 个 I/O 队列, 队列深度理论上高达 64k. 如何高效调度丰富的 I/O 队列、更好地结合多核和多 I/O 队列的优势以发挥新硬件的高并发性是新型存储系统需要考虑的问题. 由于 I/O 数量和操作类型的差异, 随机地或按顺序分配设备 I/O 队列可能会导致负载不平衡. 笔者实时监控每个 I/O 队列的任务执行情况并始终将负载最轻的 SQ 分配到工作线程. 如算法 1 所示, 其中, 针对不同 I/O 类型及访问模式赋予了不同权重 (例如 $\lambda_r = 1$, $\lambda_{\text{sw}} = 2$, $\lambda_{\text{rw}} = 3$). 此外, 由于数据量 (Size$_j$) 和操作类型 (λ_j) 对 I/O 任务的执行时间有很大影响, 我们对每个 SQ 进行评分 (第 3∼9 行) 并将 I/O 任务提交到相对空闲的 SQ 中 (第 10 行). 上述设置是系统的默认设置, 可以根据泛在应用的需求而设置不同的优先级, 比如让延迟敏感应用可以独占部分队列.

CPU 多核和 NVMe SSD 多 I/O 队列之间的绑定关系对性能也有较大影响, 两者之间可以有 4 种绑定模式, 即默认模式、成对绑定、分离绑定和偏斜绑定 [29]. 默认模式是每个 I/O 队列均未明确绑定到某个 CPU 核心. 在成对绑定模式下, 每一对队列与指定 CPU 核心绑定. 区别于以队列对为绑定单位的成对绑定模式, 分离绑定将一对队列的 SQ 和 CQ 分别绑定到不同的 CPU 核心. 偏斜绑定模式中系统会把每一个 SQ 与不同 CPU 核心进行绑定, 但所有 CQ 仅与同一个 CPU 核心进行绑定. 表 6 总结了这 4 种模式的优缺点, 可以根据实际的泛在计算场景选择合适的绑定模式.

PM 作为一个新型的具有颠覆性的存储设备, 拥有诸多独特的硬件特性, 在进行面向泛在操作系统的存储系统设计时, 应综合考虑 PM 的访问粒度、设备内预取器、NUMA、数据交错等特性进行协同设计. CPU 访问粒度是 64 字节的缓存行, 而 PM 以 256 字节的粒度 (被称为 XPLine) 访问数据, 为了减轻读写放大问题, 系统采用日志结构组织数据. 通过将元组打包到 256 字节对齐的日志条目中来提高 PM 的带宽利用率. 为利用 PM 内部的预取器以加速数据处理, 采用预取器友好的 PM 分配器, 尽可能提供顺序 I/O, 此外应尽可能分配 NUMA-Local 的 PM 空间, 避免跨 NUMA 访问导致的低性能. 如图 20 所示, 由于 PM 的数据交错特性, 通过将存储系统的 Block 对齐到 PM 的交错页面边界来利用 PM 的条带化. 这大大减少并行访问期间对 PM 的争用, 实现并行处理的性能最大化, 释放硬件潜在的并行潜能.

5.2.2 轻量化表存储

在泛在操作系统场景中, 对于结构化数据存储, 传统的堆叠式软件栈不仅会限制硬件性能的释放, 成为潜在的性能瓶颈, 也无法满足泛在应用对 I/O 时效性的需求. 垂向优化存储软件栈不应局限于单

表 6　不同绑定模式的优缺点

Table 6　Strengths and weaknesses of each CPU affinity mode

<table>
<tr><th></th><th colspan="3">Strengths</th><th colspan="2">Weaknesses</th></tr>
<tr><td>Simple</td><td colspan="3">Improving the balance of CPU resource utilization</td><td colspan="2">Enhancing the cache miss and thread migration</td></tr>
<tr><td>Pair-binding</td><td rowspan="3">Optimizing cache usage and thread management</td><td>–</td><td>–</td><td rowspan="3">Some CPU cores are unbound and idle</td><td>Competition for CPU resources between SQ and CQ bounds to the same CPU core</td></tr>
<tr><td>Separated-binding</td><td rowspan="2">Improving CPU utilization by reducing SQ-CQ resource conflicts</td><td>–</td><td>Consuming more CPU resources for polling</td></tr>
<tr><td>Skew-binding</td><td>Optimizing CPU allocation for enhanced computational and data processing capabilities</td><td>When all CQs utilize only one CPU core for polling, the check for each I/O task completion is slower compared to the separated-binding</td></tr>
</table>

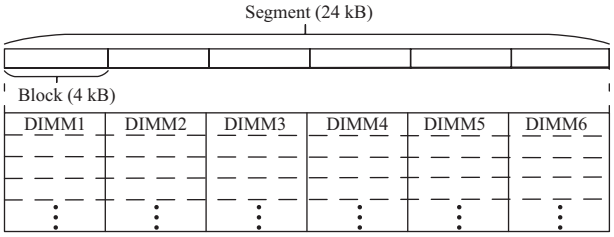

图 20　PM 的存储布局

Figure 20　Layout of PM

一层次 (数据库层或者文件系统层), 需要从软件全栈的角度重构现有软件架构, 提升系统性能.

如图 21(a) 所示, 对于传统的结构化数据存储, 往往依赖于数据库系统. 用户请求 (例如, 选择) 在存储栈中需要自上而下依次经过数据库层、文件系统层、操作系统层 (主要是块设备层) 和设备驱动层, 最后到达硬件设备. 我们将数据库层到设备驱动层之间的软件堆栈定义为全栈 (fullstack), 从文件系统层到设备驱动层的软件栈定义为 IO 栈 (IOstack). 图 21(b) 展示了笔者提出来的新型存储栈, 称为表存储栈. 它对数据库层、文件系统层和块层进行了垂向整合优化, 消除了软件冗余. 表存储栈的核心是 “表存储”. 与块存储、文件存储和对象存储一样, 表存储是一种基本存储模式, 它对二维数据库表的数据服务进行了抽象. 表存储秉承 “everything is table” 的设计理念, 而不是 “everything is file”. 在表存储格式上构建出来表存储栈, 进一步, 基于表存储栈可以搭建出面向边/云侧新型存储设备的原生表存储引擎. 其中, LATTE [29] 是一款专为 NVMe 设备设计的原生表存储引擎, 其绕过了传统的厚重软件栈, 用户态直接访问 NVMe 设备, 无需经过多层缓存和块设备层中的软件处理队列. LATTE [29] 将 NVMe 设备的 I/O 队列和 CPU 核心进行绑定, 以充分利用多核和多队列, 从而提高 I/O 的并行性. Hyte 是一款专为 PM 设计的原生表存储引擎, 它集成了文件系统和数据库的存储引擎层来实现较短的 I/O 路径, 并利用 PM 的交错特性设计存储布局, 减少并行访问期间对 PM 的争用. 面向边/云侧的原生表存储引擎具有以下 4 个主要特点:

• 低冗余. 精心布局存储栈中的每一层, 简化并整合各层之间的重复功能, 去除文件抽象并直接将 “表” 存储到设备中, 避免功能冗余和不必要的软件开销, 在用户空间中提供了更短的 I/O 路径以实现极低的 I/O 延迟.

• 硬件友好. 结合新型存储硬件的特性进行协同设计, 比如将 NVMe 设备的 I/O 队列和 CPU 核

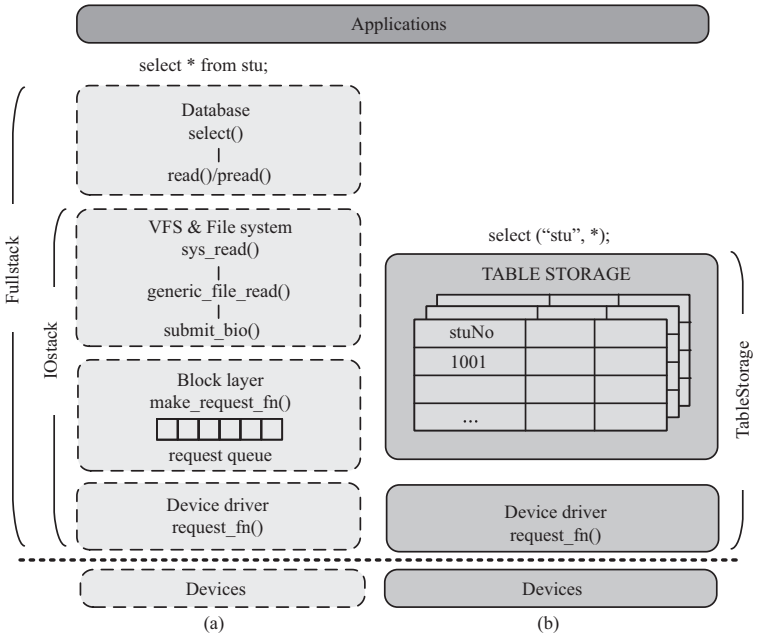

图 21 (a) 传统的存储栈和 (b) 表存储栈

Figure 21 (a) Traditional storage stack and (b) table storage stack

心进行绑定, 使得多核和多队列能得到充分的利用以提升 I/O 的并行性, 利用 PM 的交错特性设计存储布局, 减少并行访问期间对 PM 的争用, 实现并行处理的性能最大化.

• 兼容性. 考虑到二维表是结构化存储中的基本数据结构, 表存储系统为应用提供与传统数据库系统中读写兼容的表级操作接口. 为了避免引入上层应用修改的负担, 拟提供的接口与传统数据库系统中接口一致, 兼具超低延迟和向多维表的扩展功能.

• 可扩展性. 支持组件的深度集成. 由于许多 DBMS 子系统已经实现, 并且可以用于集成 [28]. 具体来说, 可以集成事务和执行引擎以满足复杂的事务和查询处理需求.

泛在场景下的结构化数据通过表存储系统直接将表结构存储到设备上, 而无需经过文件系统、操作系统等多层数据转换和拷贝, 大大减少了软件冗余以满足泛在化应用的 I/O 时效性需求. 表存储系统将设备上的存储空间划分为连续的海量数据块, 这些数据块是最小的读写单元. 为便于管理表中数据, 在数据块的上层可以构建一个逻辑层, 即将一定数量的数据块定义为一个段 (segment). 每个段仅属于一张用户表. 段中的数据块由段管理器 (segment manager) 维护和管理. 对于段中的每个数据块, 段管理器会实时更新其块号 (BlockID)、占用标记位 (IsOccupied) 等属性信息. 块号是数据块的逻辑编号, 占用标记位表示该数据块是否被工作线程占用. 块内的元组基于日志结构进行组织管理, 主要用于降低一致性开销, 规避写前日志 (write-ahead logging, WAL) 的双写问题.

5.2.3 实验分析

(1) NVMe SSD. 为验证表存储架构在 NVMe 设备上的优势, 本文对 LATTE 原型系统进行了一系列的实验. 实验使用一台配备了 3.2 TB 英特尔 DC P3608 系列的 NVMe SSD、375 GB 英特尔傲腾 DC P4800X NVMe SSD、双十核至强 E52630 v4 CPU 和 224 GB RAM 的服务器.

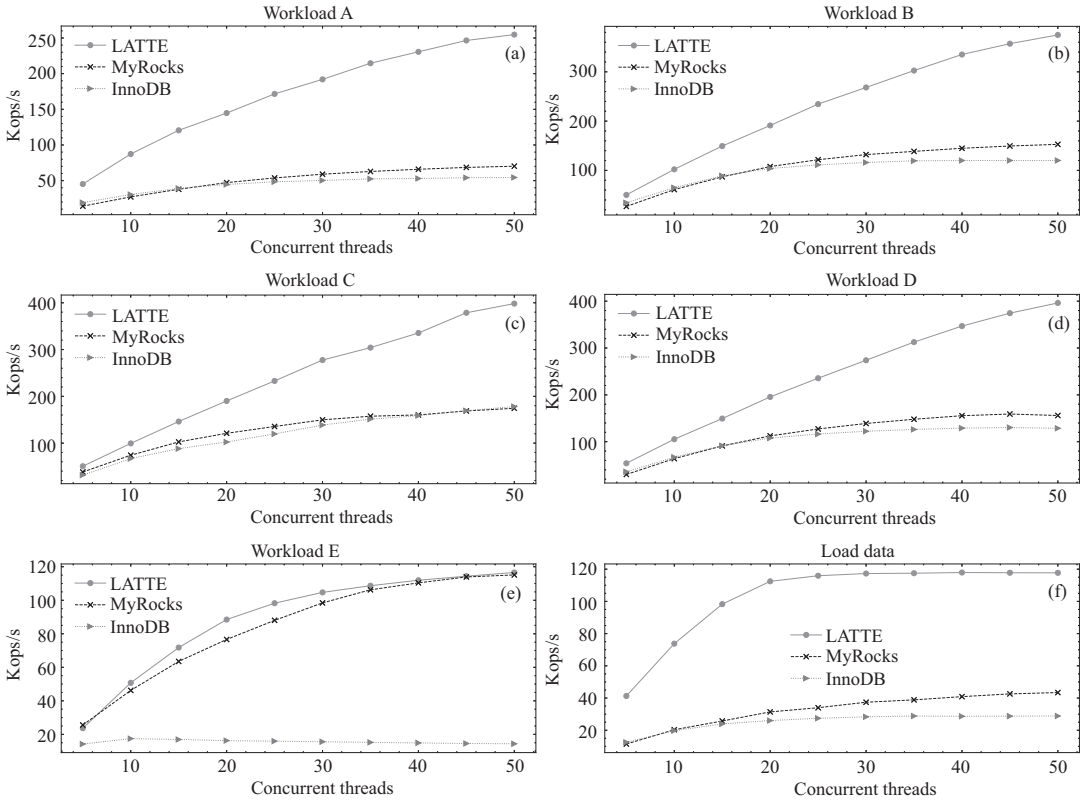

图 22 LATTE, MyRocks 和 InnoDB 的吞吐量对比

Figure 22 Throughput comparison of LATTE, MyRocks, and InnoDB. (a) Workload A: 50% read/50% update; (b) workload B: 95% read/5% update; (c) workload C: 100% read; (d) workload D: 95% read/5% insert; (e) workload E: 95% scan/5% insert; (f) load data: 100% insert

本文使用雅虎云服务基准测试 (YCSB) 将它们与 LATTE 进行对比. 如图 22 所示, LATTE 的性能明显优于 InnoDB 和 MyRocks, 吞吐量高达 InnoDB 的 6.6 倍和 MyRocks 的 3.6 倍. LATTE 性能的显著提高主要归功于 LATTE 的并行队列调度策略. 它充分利用了 NVMe 固态硬盘更高的带宽和 NVMe 设备的多个 I/O 队列, 因此读写操作可以尽可能地得以并行执行. 而且, 较短的 I/O 路径减少了每个 I/O 任务的处理时间, 从而使得队列的利用率更高.

图 23(a) 展示了使用 strace 工具追踪 LATTE 和 MySQL 中单条插入操作的具体时间. MySQL 中的 InnoDB 引擎平均写入延迟为 1007 μs, 而 LATTE 执行单条写入操作仅需 21 μs. 整个写入过程可以分为 3 个阶段: pwrite, fsync 和 write. pwrite 和 fsync 是写入实际数据的阶段, 而 write 阶段是往系统日志文件中追加日志的子阶段. 图 23(b) 和 (c) 分别描述了在 YCSB 负载下的平均延迟和尾延迟. LATTE 的延迟的急剧下降主要是因为 LATTE 在用户空间中的 I/O 路径更短, 从而消除了在文件系统、操作系统块设备层之间不必要的开销. 另外, 通过轮询 CQ 进行检查, 主机可以及时接收已完成的 I/O 请求.

(2) PM. 为了验证表存储架构在 PM 设备上的优势, 本文使用标准的基准测试对 Hyte 原型系统进行了一系列的实验. 实验机器采用双插槽的 Linux 服务器, 配备有 36 颗 Intel(R) Xeon(R) Gold 6240M CPU 核心, 192 GB (6x32GB) DRAM, 以及 1.5 TB (6x256GB) 的 Intel Optane DCPMM. 对于比较系统, 我们安装了最先进的 SplitFS 和 PMDK (v1.11) 以映射 PM 文件. 为了避免 NUMA 效应,

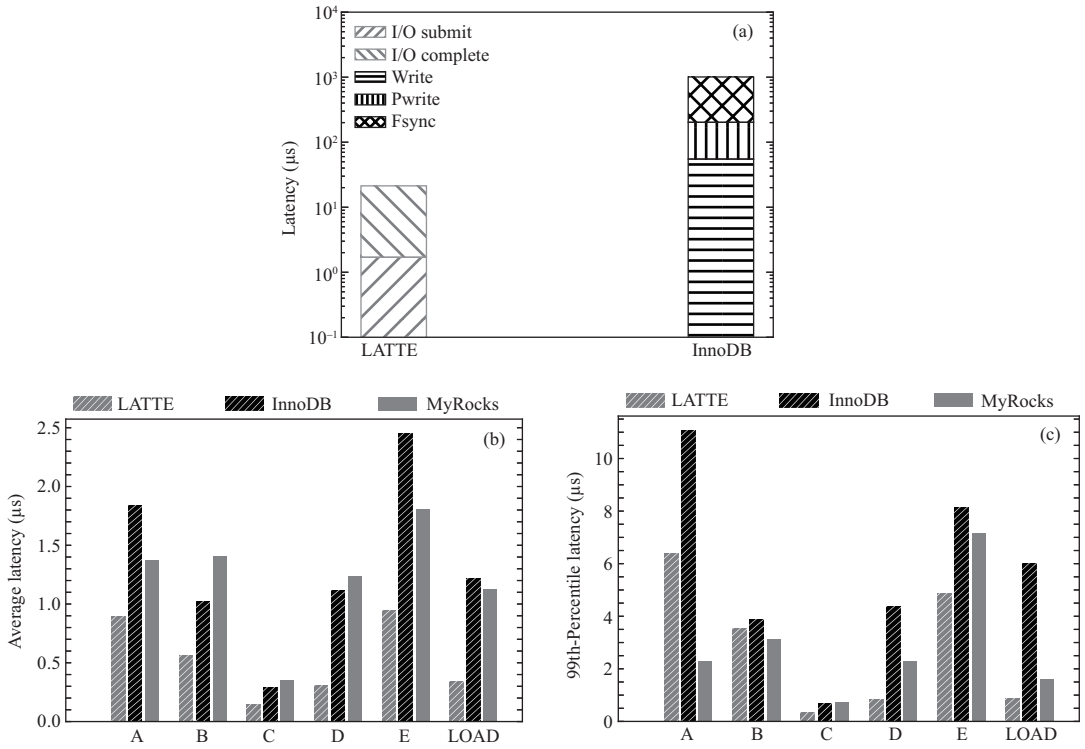

图 23 LATTE, InnoDB 和 MyRocks 的延迟对比

Figure 23 Latency comparison of LATTE, InnoDB, and MyRocks. (a) Break-down of an insert; (b) YCSB average latency; (c) YCSB tail latency

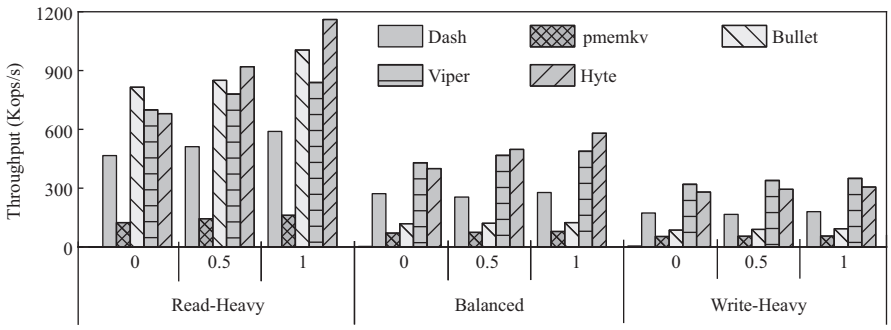

图 24 不同的读写比例和偏斜度

Figure 24 Different read/write ratio and skewness

专门在一个 NUMA 节点上进行实验.

本文使用 YCSB 基准测试, 并在工作负载中变化了两个参数: (1) 读/写比例: 读密集型 (90% 读/10% 写)、平衡型 (50% 读/50% 写) 和写密集型 (10% 读/90% 写); (2) Zipfian 分布的偏斜度, 从 0 到 1 不等. 如图 24 所示, Hyte 相对于对比系统有着 1.6~5.5× 的吞吐量的提升, 这主要得益于 Hyte 去除了文件抽象, 提供更短的 I/O 路径以减少了 I/O 处理时间, 并结合 PM 的条带化特性设计了高效的存储布局. 此外, 本文还进行了可扩展性研究, 以验证 PM 感知分配的益处, 采用了一个平衡型、偏斜度为 0.5 的工作负载. 如图 25 所示, Hyte 和 Viper 因访问多个 DIMMs 以充分利用并行性而具有

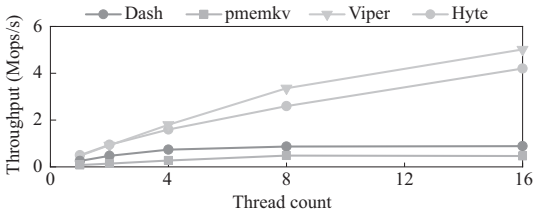

图 25 YCSB 的可扩展性

Figure 25 YCSB scalability

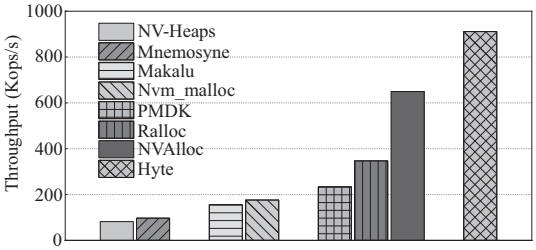

图 26 YCSB 在吞吐量方面的性能表现

Figure 26 YCSB performance on throughput

良好的可扩展性. 因访问冲突, Dash 和 pmemkv 的可扩展性受到了限制.

为了更全面地评估原生表存储引擎 Hyte 的性能, 本文将其与使用 PM 的 7 个最新的存储系统进行了对比分析. 笔者采用 YCSB 作为基准测试, 通过执行 100000 次插入和 100000 次删除操作来模拟实际工作负载, 并据此评估各系统的性能表现. 实验结果如图 26 所示, Makalu [45] 和 Nvm_malloc [46] 相比于 NV-Heaps [43] 和 Mnemosyne [44] 在吞吐量上有了 1.6~2.0 倍的提升. 这种改进主要得益于 DAX 技术的引入, 它提供了更高效的内存映射文件的方法, 从而简化了内存语义访问的实现过程. 而在此之前, 开发者需要手动实现复杂的内核模块来支持类似的功能. 此外, PMDK, Ralloc [47] 和 NVAlloc [48], 与 Makalu [45] 和 Nvm_malloc [46] 相比, 吞吐量提升了 1.3~3.7 倍. 这是因为后两者是基于模拟器的系统, 模拟器往往无法完全模拟真实设备的硬件特性和性能特征, 导致系统性能表现不佳. 最后, Hyte 与 PMDK, Ralloc [47] 和 NVAlloc [48] 相比, 实现了 1.5~3.9 倍的吞吐量提升. 这一显著优势主要得益于 Hyte 从全栈角度进行跨层的优化设计, 通过去除文件抽象并直接在设备上构建表结构, Hyte 消除了诸如 mmap 系统调用、文件元数据、冗余日志等开销, 轻量化了软件栈, 从而实现了系统性能的大幅提升.

5.3 “端 – 边 – 云” 协同架构

本文进一步研制存储、查询和处理解耦的 “端 – 边 – 云” 一体化协同架构. 如图 27 所示, 该协同架构主要由 3 部分组成: (1) 终端. 在终端设备上部署节点级表存储, 终端收到的结构化数据直接以表格式存储到本地, 降低冗余的表和文件的转换, 以及由于二者之间数据块粒度和并发控制粒度等不一致而引入的性能损失. (2) 边缘端. 在边缘端部署小型的分布式表存储系统, 在节点级表存储基础上通过分布式一致性协议进行扩展, 并保持上层接口不变. 其接收终端传来的表数据, 直接在所收到的表数据上进行处理, 将产生的结果返回给终端或者进一步传至云端进行深度分析. (3) 云端. 在云端部署大规模分布式表存储系统, 并在分布式表存储系统基础之上, 基于远程直接内存访问 (remote direct memory access, RDMA) 高速互联引入负载均衡机制. 从边缘端收到的表数据可以与上层的大数据分析框架无缝集成, 完成数据全链分析处理.

总的来说, 统一的结构化 (表) 存储方式将贯穿终端数据存储、边缘端数据管理和云端大数据分析, 规避数据冗余和转换处理等问题, 敏捷适配 “端 – 边 – 云” 的泛在操作系统对数据存储的新需求.

6 总结和展望

泛在计算的新模式和新场景需要一类新型操作系统, 即泛在操作系统, 以高效管理海量的异构资

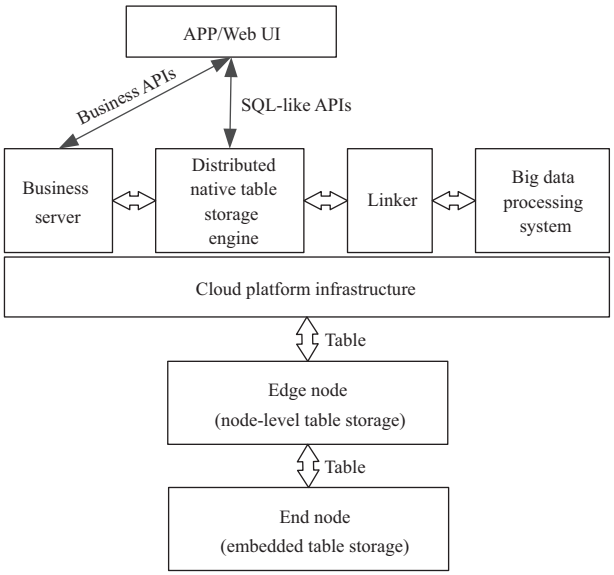

图 27 “端 – 边 – 云”一体化协同架构

Figure 27 Integrated collaborative architecture for “end-edge-cloud”

源, 满足泛在应用需求. 基于泛在操作系统设计轻量化、高性能和动态可适配的存储系统是推进泛在操作系统发展的必要举措. 本文在梳理计算机系统发展趋势的基础上, 总结泛在操作系统的基本形态及最新研究进展, 然后, 深入剖析了现有结构化存储方案在 “端 – 边 – 云” 场景下面临的挑战. 最后, 提出面向泛在操作系统的原生表存储系统, 并从端侧和边/云侧两个场景深入分析其架构优势.

从泛在操作系统和结构化存储两个层面对未来发展趋势进行展望. 泛在操作系统仍处于探索成型期, 需在不同的领域内垂直整合、共性凝练, 构成不同领域内的泛在操作系统实例. 进一步地, 在领域间做共性凝练, 生成恒定的泛在内核, 形成其他泛在操作系统的构建基础, 从而逐步构建起泛在操作系统的生态环境. 对于泛在内核, 微内核架构可能更具吸引力. 微内核几乎不提供任何服务: 它只是硬件的一个薄包装器, 足以确保多路复用硬件资源的安全. 微内核系统使用受保护的过程调用机制来提供宏内核操作系统所实现的服务. 在泛在计算环境下, 微内核架构的高度模块化优势使其具备了灵活的动态裁剪能力, 以满足不同场景的需求. 在面向泛在操作系统的结构化存储层面, 存储硬件和系统软件的发展是相辅相成的, 要充分利用硬件优势以消除软件瓶颈满足泛在化应用需求, 比如在分布式环境下结合 RDMA 的硬件特性对分布式一致性协议进行优化. 相应地, 软件栈仍有进一步优化的空间, 这也会助力硬件的进步, 此外, 缓存管理是存储系统中的重要机制, 在泛在场景下对缓存机制低空间时间开销的要求会显得尤为突出, 有必要着眼于设计新的管理策略以最小化数据在存储层次之间的拷贝.

参考文献

1 Weiser M. The computer for the 21st century. Sci Am, 1991, 265: 94–105
2 Liu Z, Wang J. Human-cyber-physical systems: concepts, challenges, and research opportunities. Front Inform Technol Electron Eng, 2020, 21: 1535–1553
3 Mei H, Guo Y. Toward ubiquitous operating systems: a software-defined perspective. Computer, 2018, 51: 50–56
4 Mei H, Cao D G, Xie T. Ubiquitous operating system: toward the blue ocean of human-cyber-physical ternary ubiquitous computing. Sci Bull Acad Sin Sci, 2022, 37: 30–37 [梅宏, 曹东刚, 谢涛. 泛在操作系统: 面向人机物融合

泛在计算的新蓝海. 中国科学院院刊, 2022, 37: 30–37]
5 Cao D G, Xue D L, Ma Z Y, et al. XiUOS: an open-source ubiquitous operating system for industrial Internet of Things. Sci China Inf Sci, 2022, 65: 117101
6 Liu S, Heller J. A record oriented, grammar driven data translation model. In: Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, 1974. 171–189
7 Mei H, Guo Y. Network-oriented operating systems — current situation and challenges. Sci China Inf Sci, 2013, 43: 303–321 [梅宏, 郭耀. 面向网络的操作系统 —— 现状和挑战. 中国科学: 信息科学, 2013, 43: 303–321]
8 Lee C, Sim D, Hwang J, et al. F2FS: a new file system for flash storage. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies, 2015. 273–286
9 Cao D G, Xue D L, Ma Z Y. XiUOS: a ubiquitous operating system for industrial Internet of Things scenarios. Commun CCF, 2021, 17: 78–81 [曹东刚, 薛栋梁, 麻志毅. 矽璘: 一款面向工业物联场景的泛在操作系统. 中国计算机学会通讯, 2021, 17: 78–81]
10 Zhang R, Liu D, Shen Z, et al. Bridging mismatched granularity between embedded file systems and flash memory. IEEE Trans Comput-Aided Des Integr Circ Syst, 2020, 40: 2024–2035
11 Lee C, Sim D, Hwang J, et al. F2FS: a new file system for flash storage. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies, 2015. 273–286
12 Oh S, Kim W H, Seo J, et al. Doubleheader logging: eliminating journal write overhead for mobile DBMs. In: Proceedings of the 36th International Conference on Data Engineering, 2020. 1237–1248
13 Ji C, Chang L P, Pan R, et al. Pattern-guided file compression with user-experience enhancement for log-structured file system on mobile devices. In: Proceedings of the 19th USENIX Conference on File and Storage Technologies, 2021. 127–140
14 Fan X P, Yan S, Huang Y C, et al. LUNAR: a native table engine for embedded devices. In: Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems, 2023. 122–133
15 Yang J, Kim J, Hoseinzadeh M, et al. An empirical guide to the behavior and use of scalable persistent memory. In: Proceedings of the 18th USENIX Conference on File and Storage Technologies, 2020. 169–182
16 XU Q, Siyamwala H, Ghosh M, et al. Performance analysis of NVMe SSDs and their implication on real world databases. In: Proceedings of the 8th ACM International Systems and Storage Conference, 2015. 1–11
17 Kim H J, Lee Y S, Kim J S. NVMeDirect: a user-space I/O framework for application-specific optimization on NVMeSSDs. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems, 2016
18 Guz Z, Li H, Shayesteh A, et al. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In: Proceedings of the 10th ACM International Systems and Storage Conference, 2017. 1–9
19 Eisenman A, Gardner D, AbdelRahman I, et al. Reducing DRAM footprint with NVM in Facebook. In: Proceedings of the 13th European Conference on Computer Systems, 2018. 1–13
20 Son Y, Kim S, Yeom H Y, et al. High-performance transaction processing in journaling file systems. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies, 2018. 227–240
21 Joshi K, Yadav K, Choudhary P. Enabling NVMeWRR support in Linux block layer. In: Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems, 2017
22 Xiang L, Zhao X, Rao J, et al. Characterizing the performance of Intel Optane persistent memory: a close look at its on-dimm buffering. In: Proceedings of the 17th European Conference on Computer Systems, 2022. 488–505
23 Nam M, Cha H, Choi Y, et al. Write-optimized dynamic hashing for persistent memory. In: Proceedings of the 17th USENIX Conference on File and Storage Technologies, 2019. 31–44
24 Chen Y, Lu Y, Yang F, et al. Flatstore: an efficient log-structured key-value storage engine for persistent memory. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, 2020. 1077–1091
25 Wang Q, Lu Y, Li J, et al. Nap: a black-box approach to NUMA-aware persistent memory indexes. In: Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation, 2021. 93–111
26 Xu J, Kim J, Memaripour A, et al. Finding and fixing performance pathologies in persistent memory software stacks. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, 2019. 427–439
27 Xu J, Swanson S. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In: Proceedings

of the 14th USENIX Conference on File and Storage Technologies, 2016. 323–338
28 Petrov I, Koch A, Hardock S, et al. Native storage techniques for data management. In: Proceedings of the 35th International Conference on Data Engineering, 2019. 2048–2051
29 Chu J, Tu Y, Zhang Y, et al. LATTE: a native table engine on NVMe storage. In: Proceedings of the 36th International Conference on Data Engineering, 2020. 1225–1236
30 Gabriel H, Viktor L. What modern NVMe storage can do, and how to exploit it: high-performance I/O for high-performance storage engines. In: Proceedings of the 49th International Conference on Very Large Data BasesMorgan, 2023. 2090–2102
31 Baptiste L, Oana B, Karan G, et al. KVell: the design and implementation of a fast persistent key-value store. In: Proceedings of the 27th Symposium on Operating Systems Principles, 2019. 447–461
32 Wang L, Zhang Z N, He B S, et al. PA-Tree: polled-mode asynchronous B+ tree for NVMe. In: Proceedings of the 36th International Conference on Data Engineering, 2020. 553–564
33 Tu Y F, Han Y J, Chen Z H, et al. URFS: a user-space raw file system based on NVMe SSD. In: Proceedings of the 26th International Conference on Parallel and Distributed Systems, 2020. 494–501
34 Qian L, Tang B, Ye B L, et al. Stabilizing and boosting I/O performance for file systems with journaling on NVMe SSD. Sci China Inf Sci, 2022, 65: 132102
35 Zhou X, Arulraj J, Pavlo A, et al. Spitfire: a three-tier buffer manager for volatile and non-volatile memory. In: Proceedings of the ACM SIGMOD Conference, 2021. 2195–2207
36 van Renen A, Leis V, Kemper A, et al. Managing non-volatile memory in database systems. In: Proceedings of the ACM SIGMOD Conference, 2018. 1541–1555
37 Chen Y, Lu Y, Zhu B, et al. Scalable persistent memory file system with kernel-userspace collaboration. In: Proceedings of the 19th USENIX Conference on File and Storage Technologies, 2021. 81–95
38 Kadekodi R, Lee S K, Kashyap S, et al. SplitFS: reducing software overhead in file systems for persistent memory. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019. 494–508
39 Li R, Ren X, Zhao X, et al. ctFS: replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In: Proceedings of the 20th USENIX Conference on File and Storage Technologies, 2022. 35–50
40 Neal I, Zuo G, Shiple E, et al. Rethinking file mapping for persistent memory. In: Proceedings of the 19th USENIX Conference on File and Storage Technologies, 2021. 97–111
41 Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory. In: Proceedings of the 9th European Conference on Computer Systems, 2014. 1–15
42 Kadekodi R, Kadekodi S, Ponnapalli S, et al. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In: Proceedings of the 28th Symposium on Operating Systems Principles, 2021. 804–818
43 Coburn J, Caulfield A M, Akel A, et al. NV-heaps: making persistent objects fast and safe with next-generation. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011. 105–118
44 Volos H, Tack A J, Swift M M. Mnemosyne: lightweight persistent memory. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011. 91–104
45 Bhandari K, Chakrabarti D R, Boehm H J. Makalu: fast recoverable allocation of non-volatile memory. SIGPLAN Not, 2016, 51: 677–694
46 Schwalb D, Berning T, Faust M, et al. Nvm_malloc: memory allocation for NVRAM. In: Proceedings of the 41st International Conference on Very Large Data BasesMorgan, 2015. 61–72
47 Cai W, Wen H, Beadle H A, et al. Understanding and optimizing persistent memory allocation. In: Proceedings of the ACM SIGPLAN International Symposium on Memory Management, 2020. 60–73
48 Dang Z, He S, Hong P, et al. Nvalloc: rethinking heap metadata management in persistent memory allocators. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022. 115–127

Structured storage for ubiquitous operating systems

Xiaopeng FAN, Song YAN & Chuliang WENG*

School of Data Science and Engineering, East China Normal University, Shanghai 200062*, China*
* Corresponding author. E-mail: clweng@dase.ecnu.edu.cn

Abstract The emerging scenarios and paradigms of ubiquitous computing in the era of human-cyber-physical fusion demand novel operating systems, namely, ubiquitous operating systems. As storage management constitutes a core function of ubiquitous operating systems, designing lightweight, high-performance, and dynamically adaptable storage systems is an essential initiative in advancing the development of ubiquitous operating systems. However, in the ubiquitous "end-edge-cloud" scenarios, structured data is widespread, and traditional storage solutions face significant challenges such as severe I/O amplification, excessive size of integrated systems, and redundant software stacks, making it difficult to meet the demands of ubiquitous applications. To address these challenges, this paper conducts original research from a holistic system perspective and proposes a native table storage system. The paper begins by reviewing the historical evolution of computer systems. Subsequently, it analyzes the emerging requirements in the era of ubiquitous computing, outlines the fundamental characteristics of ubiquitous operating systems, and presents the latest research. It then delves into the challenges faced by existing structured storage solutions in the "end-edge-cloud" scenarios. Furthermore, it proposes a native table storage system for ubiquitous operating systems and provides an in-depth analysis of its architectural advantages in "end-edge-cloud" scenarios. Finally, we summarize the paper and offer insights into future development trends.

Keywords ubiquitous computing, ubiquitous operating system, native table storage, software stack, end-edge-cloud