计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2021.20200389
58(2):406-417,2021

基于持久性内存和 SSD 的后端存储 MixStore

屠要峰[1,2]　陈正华[2]　韩银俊[2]　陈　兵[1]　关东海[1]
[1](南京航空航天大学计算机科学与技术学院　南京　211106)
[2](中兴通讯股份有限公司　南京　210012)
(tu.yaofeng@zte.com.cn)

MixStore: Back-End Storage Based on Persistent Memory and SSD

Tu Yaofeng[1,2], Chen Zhenghua[2], Han Yinjun[2], Chen Bing[1], and Guan Donghai[1]
[1](*Department of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing* 211106)
[2](*ZTE Corporation, Nanjing* 210012)

Abstract　Persistent memory (PMEM) has both the low-latency byte addressing of memory and the persistence characteristics of disk, which will bring revolutionary changes and far-reaching impact on the existing software architecture system. Distributed storage has been widely used in cloud computing and data centers, but the existing back-end storage engines represented by Ceph BlueStore are designed for traditional mechanical disks and solid state disks (SSD), and its original optimization design mechanism is not suitable for exerting the advantages of PMEM. In this paper, a back-end storage MixStore is proposed based on PMEM and SSD. Through the volatile range marking and to-be-deleted list technology, a Concurrent SkipList suitable for persistent memory is implemented, which is used to replace RocksDB to implement the metadata management mechanism. While ensuring transaction consistency, it eliminates the performance jitter and other issues caused by BlueStore's compaction, and improves the concurrent access performance of metadata. The storage optimization design of data objects based on the metadata management mechanism is adopted, non-aligned small blocks of data objects are stored in PMEM, and the aligned large blocks of data objects are stored in SSD. This makes full use of the byte addressing and durability characteristics of the PMEM, and the large-capacity and low-cost advantage of the SSD. The data update policy is optimized based on the delayed writing and copy-on-write (CoW) technology, eliminating the write amplification caused by the WAL log of BlueStore and improving the performance of small data writing. The test results show that under the same hardware environment, MixStore's write throughput is increased by 59% and the write latency is reduced by 37% compared with that of BlueStore, which effectively improves the system performance.

Key words　persistent memory; Concurrent SkipList; back-end storage; mixed storage; BlueStore storage engine

摘　要　持久性内存(persistent memory, PMEM)同时具备内存的低时延字节寻址和磁盘的持久化特性,将对现有软件架构体系产生革命性的变化和深远的影响.分布式存储在云计算和数据中心得到了广泛的应用,然而现有的以 Ceph BlueStore 为代表的后端存储引擎是面向传统机械盘和固态硬盘(solid

收稿日期:2020-06-07;**修回日期**:2020-08-17
基金项目:国家重点研发计划项目(2018YFB1003301);江苏省重点研发计划项目(BE2019012)
This work was supported by the National Key Research and Development Program of China (2018YFB1003301) and the Key Research and Development Program of Jiangsu Province(BE2019012).
通信作者:韩银俊(han.yinjun@zte.com.cn)

state disk, SSD)设计的,其原有的优化设计机制不适合 PMEM 特性优势的发挥.提出了一种基于持久性内存和 SSD 的后端存储 MixStore,通过易失区段标记和待删除列表技术实现了适用于持久性内存的并发跳表,用于替代 RocksDB 实现元数据管理机制,在保证事务一致性的同时,消除了 BlueStore 的 compaction 所引发的性能抖动等问题,同时提升元数据的并发访问性能;通过结合元数据管理机制的数据对象存储优化设计,把非对齐的小数据对象存放在 PMEM 中,把对齐的大块数据对象存储在 SSD 上,充分发挥了 PMEM 的字节寻址、持久性特性和 SSD 的大容量低成本优势,并结合延迟写入和 CoW (copy-on-write)技术实现数据更新策略优化,消除了 BlueStore 的 WAL 日志引起的写放大,提升小数据写入性能.测试结果表明,在同样的硬件环境下,相比 BlueStore,MixStore 的写吞吐提升 59%,写时延降低了 37%,有效地提升了系统的性能.

关键词 持久性内存;并发跳表;后端存储;混合存储;BlueStore 存储引擎

中图法分类号 TP316

近年来,随着移动通信和物联网技术的快速发展,数据的规模呈爆发式增长,传统的存储系统已经无法满足不断增长的海量数据存储的需要.为解决数据存储规模、数据备份、数据安全、服务质量、软件定义等问题,分布式存储系统应运而生.分布式存储是通过软件的方法把若干节点的存储资源,如磁盘、内存等组织起来,对外提供虚拟的统一的块、文件或对象接口的按需存储服务.

本地文件系统如 XFS(extents file system),EXT4 (fourth extended file system), btrfs (B-tree file system)等,具有成熟、稳定的优势,成为分布式存储系统访问本地存储资源的主要方法.应用广泛的分布式存储系统 Ceph 的后端存储 FileStore 也是基于 XFS 进行存储访问.这样做的优势是利用了文件和对象天然的映射关系,利用操作系统页缓存机制缓存数据,利用索引节点(inode)缓存机制缓存元数据,同时从操作系统层面保证了磁盘的隔离性.但是这种架构也有明显的缺陷[1],主要是事务一致性难以保证,元数据管理低效,以及缺乏对新型硬件的支持.鉴于此,Ceph 在 Jewel 版本引入了 BlueStore 作为后端存储.BlueStore 将对象数据的存放方式改为直接对裸设备进行指定地址和长度的读写操作,不再依赖本地文件系统提供的 POSIX(portable operating system interface of Unix)接口.同时,BlueStore 引入了 RocksDB 数据库保存元数据和属性信息,包括对象的集合、对象、存储池的 omap 信息和磁盘空间分配记录等,BlueStore 有效避免了数据的双写,提升了元数据的操作效率,同时借助 RocksDB 解决了事务一致性问题.

BlueStore 虽然改善了元数据效率和事务一致性,但在实际使用中存在一些明显的问题[1],主要包括:

1) RocksDB 使用预写式日志(write ahead logging, WAL)技术保证一致性,存在写放大.RocksDB 本身的 LSM-Tree(log-structured merge tree)[2]机制也会带来写放大.LSM-Tree 的数据压紧(compaction)机制还会带来业务的性能抖动.

2) BlueStore 的元数据信息、属性信息和小数据被存储到 RocksDB 中,并在其中多次复制和序列化,带来较大的 CPU 开销.

3) 为了保证事务一致性,对小数据的更新需要先从磁盘读取更新后保存到 RocksDB 中,同时 RocksDB 有自己的 WAL 日志,导致 Journal of Journal 的写放大问题.

上述 3 个问题和对应的系统开销,在固态硬盘(solid state disk, SSD)和机械盘场景下,相对系统整体性能提升带来的收益来说是值得的,但在持久性内存场景下则成为了不必要的负担.

PMEM(persistent memory)[3]也称为非易失内存(non-volatile memory, NVM)或存储级内存(storage class memory, SCM),在本文统称为 PMEM.PMEM 因其非易失、字节寻址和原地更新等特性,成为工业界和学术界研究的热点.

BlueStore 没有很好利用 PMEM 内存式高速处理和外存式持久化的双重能力.PMEM 具有非易失、字节寻址特性[4],可以通过简化日志的方式保证事务一致性,减少日志结构引起的数据整理和写放大,以及由此带来的额外系统开销和性能瓶颈.与传统基于块的存储设备相比,PMEM 提供了更高的吞吐和更低的延迟.但是,PMEM 比 SSD 容量小且单位存储成本更高,而且目前商用的 PMEM 读写具有不对称性,例如通过 Intel MLC[5]测得的 Intel Optane DC Persistent Memory 读带宽最高约为 6 GBps,而写带宽则约为 2 GBps.因此,PMEM 更适合用来

存储元数据和小数据,而 SSD 更适合存储大块的数据对象.充分发挥 PMEM 和 SSD 存储介质的优势特性,设计高性能的后端存储,将为解决传统存储的不足带来了新的机遇.

本文提出了一种基于 PMEM 和 SSD 的分布式后端存储 MixStore,通过针对 PMEM 和 SSD 特性的组合优化设计,构建性能更优的本地存储系统,解决了 Ceph 现有后端存储 BlueStore 的写放大以及 compaction 等问题.

1 相关工作

Ceph 现有的 BlueStore 被设计为一个面向 SSD 和 HDD(hard disk drive)的存储引擎,致力于提供快速的元数据操作、避免对象(Object)写入时的一致性开销,同时解决日志双写问题.BlueStore 通过将元数据保存到 RocksDB 来实现快速的元数据操作.通过 Space Allocator 进行磁盘空间管理,将 Object 直接写入块设备,去除对文件系统的依赖.同时,BlueStore 实现了写时复制机制(copy-on-write, CoW)方式的 Object 更新,从而避免在日志中包含完整的 Object 数据,解决双写问题.但是基于 RocksDB 管理元数据存在写放大问题,RocksDB 的 compaction 操作也会导致性能抖动,影响系统吞吐率.另一方面,在数据规模较大以及 EC(erasure code)等应用场景中,基于 RocksDB 的元数据管理仍然难以满足性能要求.即使更换持久性内存等更高性能的硬件,所获得的提升也有限.

以 LevelDB[6] 和 RocksDB[7] 为代表的 LSM-Tree 存储引擎凭借其优异的写性能成为众多分布式组件的存储基石.在大型生产环境中,例如 BigTable[8],Cassandra[9],HBase[10] 等广泛部署了各种基于 LSM-Tree 的本地存储.LSM-Tree 把小的随机写通过合并操作变成连续的顺序写,因此对 HDD 硬盘的写入性能有很好的优化.相比 HDD 来说,SSD 的顺序写入和随机写入性能差别较小,所以 LSM-Tree 对 SSD 的写入性能提升有限.为此,WiscKey[11] 提出一种基于 SSD 优化的键值(key-value, KV)存储,核心思想是把 key 和 value 分离,只有 key 被保存在 LSM-Tree 中,而 value 单独存储在日志中.这样就显著减小了 LSM-Tree 的大小,使得查找期间数据读取量减少,并减轻了索引树合并时不必要的数据移动而引起的写放大.WiscKey 保留了 LSM-Tree 的优势,减少了写放大,但数据访问时需要进行多次 MAP 映射,且依然存在 LSM-Tree 固有的 compaction 过程.SLM-DB[12] 基于 PMEM 和磁盘对 LSM-Tree 进行改进,将磁盘上的数据从多级树减少到 1 级,在 PMEM 上构建 B 树加速对磁盘数据的访问,具有低写入放大和近乎最优的读取放大特性.但在磁盘上数据依然需要做 compaction 操作,同时因为数据存放在磁盘,小数据的访问性能较差.NoveLSM[13] 是一款基于 PMEM 的 LSM-Tree 存储结构的 KV 系统,旨在利用 PMEM 为应用提供高吞吐、低延迟的 KV 存储,在 WAL 日志满的时候,或在 compaction 受阻时,引入 PMEM 进行加速,是一种改良的 LSM-Tree,但没有完全解决写放大和 compaction 的问题.

针对 PMEM,NOVA[14] 把 DRAM 和 PMEM 相结合,索引存放在 DRAM 中,日志放在 PMEM 中,每个 inode 都有自己的日志,日志通过链表进行组织,这样的设计充分发挥了 PMEM 的字节寻址特性,但因为单位存储成本较高不适合作为大容量后端存储.Ziggurat[15] 提供了一个分层文件系统,将 PMEM 和慢速磁盘结合在一起,通过在线预测应用的访问模型,把同步的、小的数据写入 PMEM,把异步的、大的数据写入磁盘,这种分层放置的策略对有多样性的访问模型有效,但对于单一数据类型的应用将不能充分发挥 PMEM 和磁盘的各自的特性.NV-Booster[16] 提出了一种基于 PMEM 的文件系统来加速存储节点的对象访问,通过 PMEM 中的高效命名空间管理,实现了快速的对象搜索以及对象 ID 和磁盘位置之间的映射,其优化是针对 Ceph 的 FileStore,未解决 BlueStore 所面临的问题.此外,以上这些工作都提供基于内核态的本地文件系统,由于系统调用、内存拷贝和中断而导致较大的性能开销,并不适合作为分布式存储的后端.Octopus[17] 提出了一种基于 PMEM 和 RDMA(remote direct memory access)的分布式持久存储文件系统,利用了 RDMA 可以直接读写 PMEM 的特性,将文件数据设置为全局可见,直接执行远程读写,提高性能;而文件元数据则设置为私有,对应的操作由服务端来执行,从而保证安全性和系统数据一致性.但该系统缺少对大容量 SSD 的集成,不适合作为分布式存储的后端.

KVell[18] 是基于 NVMe SSD 设计的高效 KV 存储,核心设计思想是简化 CPU 的使用,数据在 SSD 上不排序,每个分区的索引在内存中是有序的.这种设计在每次启动时需要进行索引的重构,无法

满足快速重启动的工程需求.KVell 使用页缓存和 LRU(least recently used)算法负责磁盘上的空闲空间申请和释放,省去了磁盘上空闲空间回收和整理的过程.但是当磁盘容量快被用完,特别是被覆盖写时,大概率触发 SSD 的擦除→拷贝→修改→写入过程,实际 IO 性能会大幅下降.Pmemkv[19]是针对持久性内存优化的本地键值数据存储,其底层包含了 cmap、csmap、tree3、stree 等多种存储引擎,其中 csmap 引擎基于 SkipList 实现,put、get、count 等操作的性能随线程数扩展,但其 remove 操作使用了全局锁,限制了并发性.

在并发访问控制和事务一致性方面,Harris 等人[20]利用指针标记的方法解决了 DRAM 上 CAS(compare-and-swap)算法在并发执行节点插入和删除操作时,删除前置节点会导致插入节点丢失的问题.David 等人[21]基于此研究提出一种无锁数据结构,实现了适用于 PMEM 的 CAS 算法,当使用 CAS 指令更新前置节点的 *next* 指针时,同步为该指针添加未持久化标记,并在完成持久化后去除此标记.其他并发的更新操作在检测到未持久化标记时,则首先执行持久化,以保证更新的线性执行.但上述方法中,保存在 PMEM 中的指针包含了未持久化标记,后续的标记移除操作则可能因故障而未被持久化,导致故障恢复后新的插入操作重复执行指针持久化,产生不必要的性能开销.PMwCAS[22]通过无锁持久高效的多字段 CAS 技术,解决了多字段更新的原子性问题,大大降低了构建无锁索引的复杂性,但未提供节点删除时资源回收的解决方案.TLPTM[23]通过分配感知和基于压缩算法的日志优化策略设计了一种基于微日志的持久性事务内存系统,虽然减少了日志的写入量,但本质上还是一种基于日志的事务系统,没有解决前述的数据双写问题.

肖仁智等人[24]从持久索引层面、文件系统层面和持久性事务等方面对面向 PMEM 的数据一致性相关工作进行了综述和总结,并指出在开发新的一致性的持久内存索引方面,未来更多的工作将集中在 LSM-Tree 和 SkipList 上.

综上,现有后端存储 BlueStore 固有的写放大和 compaction 问题在 PMEM 上更加突出,业界针对 SSD 和 PMEM 下的 LSM-Tree 提出了许多改进,但仍然避免不了 compaction 带来的性能抖动.在构建基于持久性内存的存储系统方面业界提供了一些解决方案,但大多是内核态的实现,存在较多的上下文切换和数据拷贝的开销,并且基于本地文件系统的后端存储在事务一致性保证和高效元数据访问方面并不友好,同时 PMEM 本身由于容量和成本原因并不适合单独构建大规模分布式存储系统,因此基于 PMEM 和 SSD 的特性联合设计后端存储存在切实的需求.

2 MixStore 架构设计

针对 PMEM 和 SSD 的硬件场景,本文设计了 MixStore 替换原有的 BlueStore,作为 Ceph 的后端本地存储.其最大的挑战是事务一致性处理和充分发挥新型硬件的性能,事务一致性要求一次 IO 更新操作要么全部成功,要么什么也没做,确保系统异常崩溃时数据的正确性.通常基于本地文件系统实现后端存储时,需要先写 WAL 日志,再更新数据,但这样存在严重的写放大而影响到系统的性能.与此同时,由于本地文件系统在管理文件时使用了多层级 inode 来管理文件数据和元数据的存放,但后端存储也有自己的元数据,会导致后端存储元数据到本地文件系统多级元数据和数据的映射的开销,所以直接基于裸盘设计后端存储更具有性能优势.

MixStore 利用 PMEM 的字节寻址和持久性特性,把元数据和小块数据存放在 PMEM 中,基于 SSD 裸盘,把对齐的大块数据对象存储在 SSD 上.PMDK(persistent memory development kit)[25]是 Intel 提供的适用于持久性内存的开源库,专门针对 PMEM 进行了优化.基于 PMDK 事务机制可以确保在节点故障后,所有内存操作都回滚到最后提交的状态,PMDK 实现了线程内的原子性,但没有提供隔离性.MixStore 提出了一种基于 PMEM 的 Concurrent SkipList 索引管理机制,基于 PMDK 事务、区段标志位以及待删除列表的方式保证节点修改的事务一致性、原子性和并发性.同时提出了一种高效的数据对象更新管理方法.针对 PMEM 和 SSD 各自的特性,优化数据对象的访问性能.

MixStore 的系统架构如图 1 所示.作为用户态后端存储,MixStore 对外提供 ObjectStore 接口,内部分为元数据管理和数据对象的管理.MixStore 的核心创新是在 PMEM 上实现了基于 Concurrent SkipList 的索引替换原有 RocksDB 所实现的元数据管理功能,去除了对 RocksDB 的依赖;通过直接管理 Object 元数据,避免了 WAL 和 LSM-Tree 机制带来的写放大和 compaction 时的性能抖动;同时避免了序列化和反序列化带来的 CPU 开销;小块

的数据存放于 PMEM 中,并优化了小 IO 的事务一致性处理方法,避免了 BlueStore 的 Journal of Journal 的问题,对于大块的数据对象,首尾非最小可分配大小(minimal allocate size, MAS)对齐的部分存放在 PMEM 中,中间 MAS 对齐部分使用 CoW 机制存储到 SSD 中.得益于 PMEM 的字节寻址特性,对于非对齐写和小 IO 有更好的优化.

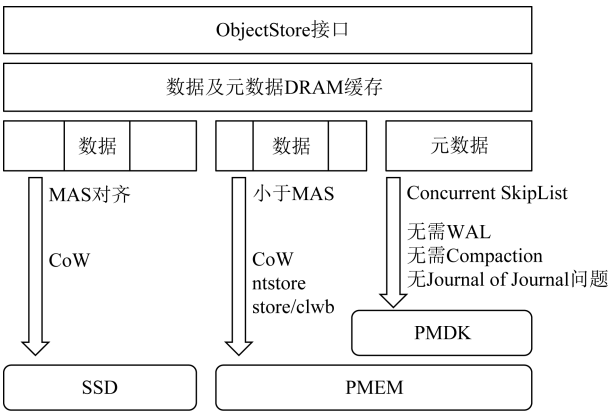


Fig. 1 System architecture of MixStore

图 1 MixStore 系统架构图

在索引结构的选择上,现有的 LSM-Tree 在 WAL 日志更新,多层级查找及序列化方面开销巨大[13],且无法利用 PMEM 的字节寻址和低延迟特性,显然不适用于 PMEM.有序链表因为 $O(N)$ 的较高时间复杂度消耗亦不适合用作索引结构.对于树形结构,数据量的变化通常需要对树结构重新平衡,重新平衡操作可能会影响树结构的较大范围,这将需要在较多树节点上使用互斥锁.相比树结构,SkipList 具有更好的局部性,更适合并发访问/修改[26-27].对 SkipList 的操作只会影响到节点本身以及前后插入的节点,在更改 SkipList 结构时不需要锁住和同步整个 SkipList 数据,具有更好的并发性.MixStore 选择 SkipList 作为索引结构,同时基于 PMEM 实现了事务一致性和并发性.

在写放大方面,PebblesDB[28] 显示,插入 5 亿个键值对时,平均键值大小 100 B,共写入 45 GB 数据,RocksDB 实际写入了 1 868 GB 数据,写放大率达 42 倍.对于 SkipList 来说,只需要额外一个指针,加上中间层次节点,平均一个键值对额外增加 16 B,写放大为 0.16 倍,远远少于 RocksDB 的写放大.对于每个 100 B 键值加上 SkipList 管理开销 16 B,可以管理 NVMe SSD 上的 4 KB 的数据块,实际元数据最多消耗仅为数据量的 2.8%.

在时间效率方面,SkipList 通过维护一个多层次的链表,且与下面一层链表元素的数量相比,每一层链表中的元素的数量更少.算法首先在最稀疏的层次进行搜索,直至需要查找的元素在该层 2 个相邻的元素中间.这时,算法将跳转到下一个层次,重复刚才的搜索,直到找到需要查找的元素为止[29].SkipList 与平衡二叉树一样可提供时间复杂度为 $O(\log N)$ 查找、插入和删除操作[30],而无需像 B 树、红黑树或 AVL 树所要求的那样,进行复杂的树平衡或页面拆分,并且实现起来更加简单和简洁.

在数据布局方面,根据 DRAM,PMEM,SSD 各自特性和优势,合理搭配使用以充分挖掘系统的性能.SSD 容量最大,用于存储具体的数据内容,PMEM 随机读写性能好,且具有持久性,用于存储元数据信息、磁盘空间分配信息、属性信息等,同时还存储非 MAS 对齐的临时的数据内容.而对于 DRAM,性能相比 PMEM 更好,用作热点数据和元数据的缓存.对于 MAS 对齐的数据首先写入 DRAM,然后持久化到 SSD 中,但在 DRAM 中仍然保留,直至后续被 LRU 淘汰.

在事务一致性方面,PMDK libpmemobj 库实现了事务机制,提供更新原子性,以及崩溃一致性保证.通过 Concurrent SkipList 设计,基于 PMDK 实现了强一致的元数据管理,而不必额外使用日志机制,从而避免了 Journal of Journal 问题.但是 PMDK 未提供足够的隔离性,当多个线程并发执行事务操作时,对共享资源的访问仍然需要额外的隔离机制.MixStore 通过区段标志位方式进行并发控制访问,解决了多线程隔离性问题.对于 PMEM 中非 MAS 对齐的小数据更新使用 ntstore 或 clwb 指令通过 CoW 方式直接写入.因为大块数据使用 ntstore 指令具有更好的性能表现[31],所以 MixStore 对于大于等于 256 B 的数据采用 ntstore 的方式进行存储,而对于小于 256 B 的数据采用 clwb 的方式进行存储.

3 关键技术

本节详细描述 MixStore 在元数据管理和数据对象管理方面的关键技术.

3.1 基于 Concurrent SkipList 的元数据管理

MixStore 的元数据通过基于 PMEM 的 SkipList 进行管理,内存结构的 SkipList 可以基于 CAS 指令实现数据的并发更新机制,但在 PMEM 上应用 CAS 算法面临新的挑战.

现有的 Cache Coherency 模型是针对 DRAM

的,当 CAS 操作完成时,数据在 CPU Cache 间保持一致,但对于 PMEM,仍然需要额外的 clflush 操作,才能完成持久化.因此,操作原子性被破坏,如果在断电时仅完成了指针的更新,而没有持久化到 PMEM,则在多线程并发更新访问时,会导致数据结构的损坏,产生内存泄漏和一致性问题.

3.1.1 元数据插入机制

与 David 等人[21]提出的无锁数据结构不同,MixStore 使用一种易失区段标记来解决并发操作一致性问题,即将 SkipList 分为多个区段,并在 DRAM 中保存每个区段的更新标记.更新操作在完成前置节点查找后,通过 CAS 指令将对应区段设置为更新中,以与同区段的其他更新操作互斥.更新标记保存在 DRAM 而不是 PMEM 中,以提升访问效率,并简化故障恢复处理.区段级而不是节点级的更新标记则可以缩减内存使用量,并有助于解决插入和删除并发时的节点丢失问题.为了解决跨区段更新的问题,在 SkipList 中引入 *dummy* 节点作为区段边界,以保证前置节点和待插入节点总是在一个区段中.此时,仅同区段的更新操作需要互斥,不同区段的更新操作可以并发执行,而查找操作不需要互斥,完全是并发执行的.

以一个 2 区段的 SkipList 为例,新元素的插入操作如图 2 所示,首先根据待插入元素 d 所属区段设置对应的更新标记,并开启 PMDK 事务,然后将待插入元素 d 的 *next* 指针指向 x(x 为区段中第 1 个元素时,则指向该区段的起始 *dummy* 节点),通过 CAS 操作将元素 c 的指针指向元素 d.此时,CPU cache 中的插入操作完成,元素 d 对查找操作可见.接下来,提交 PMDK 事务将更新操作持久化到 PMEM 中,并复位 SkipList 的区段更新标记.此时,CPU cache 和 PM 中的数据视图达成一致,整个插入操作完成.对于待插入元素的高度超过 1 的情况,由于上层 list 在故障恢复时可以通过底层的数据进行重建,因此可以仅通过 CAS 操作进行更新,无需主动执行持久化操作.

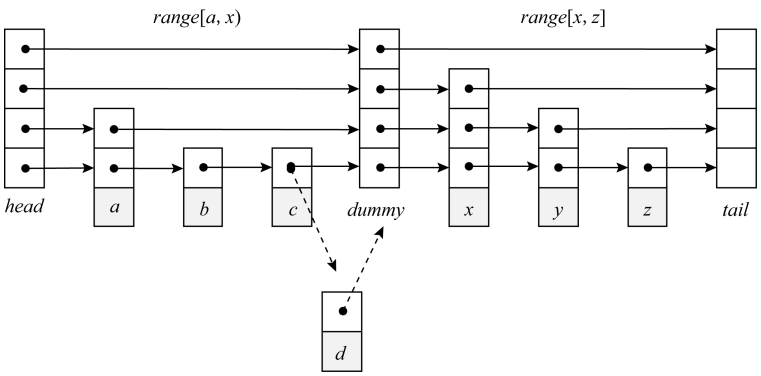


Fig. 2 Metadata insertion mechanism

图 2 元数据插入机制

为了进一步减少同区段的并发冲突,插入时的查找操作可以使用二次确认(double check)的方式,即首先遍历 SkipList 确定新元素的插入位置,然后设置区段更新标记,并再次检查前置节点的 *next* 指针,确定是否需要重新执行查找操作.完整的节点插入流程描述如算法 1.

在节点插入时需要同时更新本节点的 *next* 指针和前一节点的 *next* 指针,把这 2 个指针更新放在同一 PMDK 事务中执行,确保系统的崩溃一致性,避免断电导致的各类异常,例如数据部分持久化导致的内存区域泄漏、SkipList 链表断裂等情况.

算法 1. 插入节点 *Insert*(*node*).

① *pre*←*SearchPrevious*(*node.key*);
② *MarkRange*(*node.key*);
③ if *pre.next.key*<*node.key* then
④ *pre*←*SearchPrevious*(*node.key*);
⑤ end if
⑥ TX BEGIN /* pmdk transaction */
⑦ *node.next*←*pre.next*;
⑧ *pre.next*←*node*;
⑨ TX COMMIT /* pmdk transaction */
⑩ *UnmarkRange*(*node.key*).
⑪ function *MarkRange*(*key*)
⑫ *range*←*KeyToRange*(*key*);
⑬ do
⑭ *marked*←*CAS*(*range*,0,1);
⑮ while *marked*≠TRUE
⑯ end function

⑰ function *UnmarkRange*(*key*)
⑱ *range*←*KeyToRange*(*key*);
⑲ *CAS*(*range*,1,0);
⑳ end function

3.1.2 元数据删除及查询机制

元素删除操作与插入操作类似,需要通过区段更新标记进行互斥.与插入操作不同的是,元素从 *SkipList* 移除后,由于可能有其他线程正在执行查找操作,其内存区域不能立即释放.同时,在等待释放期间,为了避免异常断电导致持久内存泄漏,待删除的元素需要进行妥善的跟踪管理.

MixStore 使用待删除列表解决上述问题.如图 3 所示,首先为每个线程维护一个 *SkipList* 访问事件时戳,线程每次完成 *SkipList* 的访问后,时戳加 1.同时,每个线程有一个保持在持久内存中的待删除元素列表,该列表实际由 2 个数组和对应的 2 个时戳向量组成,在当前数组满时互相交换,并在符合特定条件时清空备用数组.时戳向量中记录了特定时刻所有线程的时戳值,线程时戳和时戳向量均无需持久化,在启动时初始化为 0 即可.元素从 *SkipList* 中删除时,会被暂存到待删除列表,同时更新此刻的时戳向量.

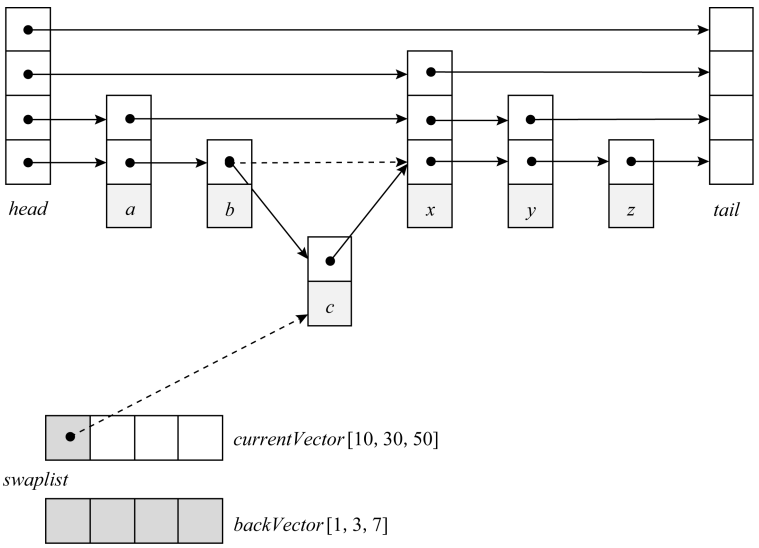


Fig. 3 Metadata deletion mechanism

图 3 元数据删除机制

以删除元素 *c* 为例,首先设置区段更新标记,并开启 PMDK 事务,然后使用 CAS 操作将元素 *b* 的 *next* 指针指向元素 *x*.此时,元素 *c* 对后续的查找操作不可见,但仍然可能有查找过程正在访问元素 *c*.然后,元素 *c* 被加入待删除列表,提交 PMDK 事务完成持久化,同时复位区段更新标记.接下来,更新时戳向量,记录当前时刻所有线程的时戳值.显然,删除操作发生在此时戳向量之前,当后续某一时刻的时戳向量严格大于该值时,即每个线程的时戳都大于旧值时,元素 *c* 的内存可以被释放.具体而言,在更新当前时戳向量后,将当前时戳向量与备用数组的时戳向量进行比较,并确定是否清空备用数组.完整的节点删除流程描述如算法 2 所示.

算法 2. 删除节点 *Remove*(*key*).

① *pre*←*SearchPrevious*(*key*);
② *MarkRange*(*key*);
③ if *pre.next.key*≠*key* then
④ *pre*←*SearchPrevious*(*key*);
⑤ end if
⑥ *curr*←*pre.next*;
⑦ TX BEGIN / * pmdk transaction * /
⑧ *pre.next*←*curr.next*;
⑨ *AddToSwaplist*(*curr*);
⑩ TX COMMIT / * pmdk transaction * /
⑪ *UnmarkRange*(*key*);
⑫ *UpdateCurrentVector*();
⑬ if *currentVector*>*backVector* then
⑭ *ClearBacklist*();
⑮ end if

在进程重启后的故障恢复阶段,由于待删除列表中的元素仍然可能会被其他线程的恢复过程访问,故保留待删除列表中的元素,并将 2 个时戳向量重置为 0.此时,待删除列表中的元素被延迟到后续删除操作中进行清理.

与插入和删除不同,查询操作不需要关注区段标记.节点插入时,链表指针的更新通过原子操作完成,从链表中删除的节点则首先保存在额外的待删除列表中,直到查询操作完成后相关内存才被释放.因此,查询操作是完全并发的,不存在等待时间,故而具备较高的性能和延迟一致性表现.

3.2 数据对象管理

MixStore 充分利用 PMEM 的字节寻址特性,来优化 Object 的写入性能.一方面,Object 的元数据使用存储在 PMEM 中的 SkipList 进行管理,另一方面,对于非对齐 IO 和随机小 IO,也利用 PMEM,通过不同的写入策略进行优化.

虽然 SSD 有较好的随机读写性能,但相比于顺序读写仍有所降低.为此,设定 SSD 的最小分配大小 *MAS*,对于普通的 SATA SSD 来说,*MAS*=16 KB.对于 NVMe SSD 来说,因为有更好的随机读写性能,我们设定 *MAS*=4 KB.通过数据写入策略的设定,有效减少了 SSD 的随机读写,进一步优化系统的性能.

Object 更新时,小于 *MAS* 的数据被直接写入 PMEM,后续小 IO 写入也直接在 PMEM 中进行追加更新和合并,从而有效减少了 SSD 的磨损和闪存转换层(flash translation layer, FTL)频繁垃圾回收(garbage collection, GC)导致的性能抖动.对于此类数据,如图 4 左侧部分所示,BlueStore 采用 DeferredWrite 策略,即首先从 SSD 读取旧数据与新数据进行合并,获得 block 对齐的待更新数据块,然后将待更新数据块和元数据作为 WAL 写入 RocksDB.最后,异步执行数据的 In-Place 更新,并在完成数据写入后删除 WAL.这种方式导致在构建 WAL 时即需要执行 SSD 数据读取,且增加了 PMEM 的写入数据量.与 BlueStore 不同,MixStore 对异步写入的数据不使用 In-Place 更新策略,如图 4 右侧部分所示,MixStore 将非对齐小 IO 直接写入 PMEM,然后异步执行 SSD 读取操作并进行 CoW 异地更新.MixStore 面向更高性能的 PMEM 和 SSD 设计,由于数据的不连续分布对 SSD 设备的读性能影响要远小于 HDD,因此选择 CoW 异地更新而不是 In-Place 更新,从而避免了构建 WAL,降低了 PMEM 写入数据量;此外,PMEM 的高性能也可以更好的支撑不连续的读取操作,使得 MixStore 可以使用更惰性的数据回写策略,从而提升非对齐小 IO 的合并概率,减少最终的 SSD 写入次数和数据的碎片化.

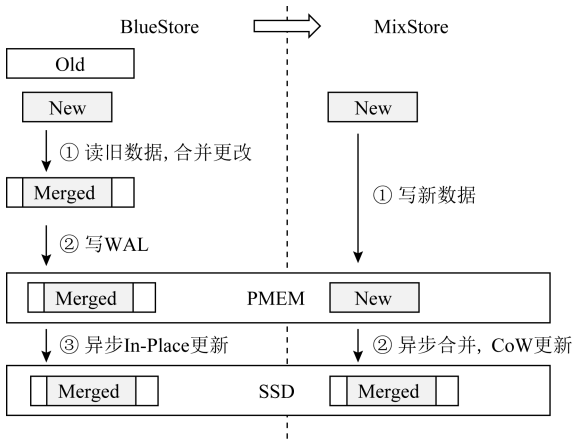


Fig. 4 Comparison of unaligned IO processing

图 4 小 IO 数据更新流程对比

对于大于 *MAS* 的更新操作,则首先进行 *MAS* 对齐,将其切分为 3 个部分,即首尾非 *MAS* 对齐的部分和中间 *MAS* 对齐的部分.对于首尾非对齐部分使用与小 IO 相似的方式处理,对齐部分则使用 CoW 策略,在写入 SSD 后更新元数据信息.具体而言,整个更新操作分为 2 个阶段,即首先将中间 *MAS* 对齐部分异地写入 SSD,首尾非对齐部分写入 PMEM,然后将所有的元数据修改在一个 PMDK 事务中完成.

Object 读取时,通过访问 SkipList 获取数据存放位置信息,首选从 SSD 中读取本次 IO 涉及的数据区域,然后从 PMEM 中读取非对齐的小 IO 增量,并进行必要的合并即可.得益于 PMEM 的随机访问性能,对数据的碎片化有更高的容忍度,读性能仍然可以保持较高的水平.

MixStore 根据 PMEM 的可用容量和数据的碎片化情况决定是否执行合并操作,PMEM 读取 4 KB 数据的延迟通常不超过 5 μs,而 NVMe SSD 的 4 KB 读延迟通常在 100 μs 左右.MixStore 在读操作中检查数据的碎片化情况,当碎片数量超过阈值以后,触发合并操作,合并后的数据使用 CoW 的形式进行更新,以避免 In-Place 更新可能的一致性问题.碎片阈值是动态的,在 PMEM 容量充足时,碎片阈值较高,并随着可用容量的减少而逐步降低.

4 实验及分析

4.1 实验环境

本实验采用的硬件配置信息如表 1 所示,实验在一个由 40 GbE 以太网交换机连接的 4 节点集群

上完成,测试使用的持久内存为 Intel 最新推出的 Optane DC 持久性内存,单条容量为 256 GB,NVMe SSD 为 Intel 的 P4610,单盘容量为 1.6 TB.PMEM 可以配置为 App-Direct 或 Memory 这 2 种工作模式,由于本实验都是把 PMEM 作为持久存储来使用,所以采用 App-Direct 模式.本实验对比 BlueStore 和 MixStore 测试时所采用的硬件环境相同.

Table 1 Experiment Configuration

表 1 实验环境配置

项	值
CPU	Intel® Xeon® Gold 6230N×2
DRAM	32 GB×12
PMEM	Intel® Optane™ PMem 256 GB×4
NVMe SSD	P4610 1.6TB×4
Network	XL710 for 40 GbE QSFP+

所有主机均安装 Fedora release 29,内核版本升级到 4.18.16-300,以提供对 PMEM 的支持.测试基于 Ceph Luminous 12.2.12 或其修改版本完成,使用 FIO 3.10 进行负载模拟.

Ceph 集群部署在其中 3 台安装了 PMEM 和 NVMe 的主机上,PMEM 配置为 fsdax 模式.另一台主机作为客户端运行 FIO 程序,该主机上实际未安装 PMEM 和 NVMe 设备.

4.2 性能对比

测试 BlueStore 时,PMEM 作为块设备使用,用于存储 WAL 和 DB 数据,SSD 用于保存对象数据.测试 MixStore 时,PMEM 格式化为 ext4-dax,然后通过 MMAP 转换为持久内存使用,PMEM 用于存储索引和小块数据,大块对象数据保存在 SSD 上.

本次测试使用 FIO 工具的 rbd ioengine 访问 Ceph RBD 接口,后端为 2 副本存储池.为了更接近真实负载,FIO 的队列深度被设置为 32,而不是更大的值.每次测试使用 20 个 FIO 客户端分别对 20 个大小为 50 GB 的 RBD Image 进行读写,测试过程持续 5 min.

4.2.1 写性能

本次测试覆盖了小 IO 随机写入和大 IO 顺序写入 2 类场景.对于随机写入,为了进一步确认 BlueStore 和 MixStore 针对不同数据大小的写入策略差异,分别测试了 2 KB,4 KB,16 KB 的数据写入.对于顺序写入,分别测试了 64 KB,128 KB,256 KB 的数据写入.

每组数据测试完成后,重启全部 OSD(object storage daemon)节点,以排除缓存等因素的干扰.

小 IO 随机写入的性能如图 5 所示,可以看到,对于所有随机写入场景 MixStore 均有优于 BlueStore 的表现,其中 4 KB 写入高出约 59%,这显然得益于 MixStore 的高效原数据管理和事务日志的消除.对于 2 KB 随机写,BlueStore 和 MixStore 均使用 PMEM 进行数据暂存,不同的是,BlueStore 需要执行 read-merge-write 操作,因此最终写入 PMEM 的为包含 4 KB 数据块的事务日志,而 MixStore 则直接将 2 KB 的数据以及相关元数据写入 PMEM.测试过程中,通过 iostat 或 pcm-memory.x 可以观察到 MixStore 消除了 NVMe 读操作,PMEM 写带宽约为 BlueStore 的一半.但由于 NVMe 磁盘实际负载较小,最终 MixStore 测试结果仅表现出与 4 KB 类似的优势,领先 BlueStore 约 56%.对于 16 KB 的随机 IO,MixStore 领先约 34%,一方面,MixStore 更高效的元数据操作释放了 NVMe 的性能,另一方面,对于 16 KB 的数据,NVMe 写操作在整个请求处理开销中的占比上升,导致最终的性能优势不如 4 KB 数据明显.MixStore 的平均延迟和尾延迟也呈现出和 IOPS 相同的优势,平均延迟约为 BlueStore 的 63%~75%,尾延迟约为 BlueStore 的 72%~84%.

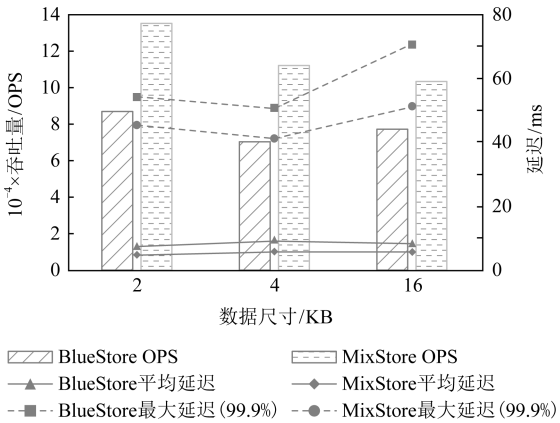


Fig. 5 Comparison of random write

图 5 随机写测试对比

如图 6 所示,对于大 IO 顺序写入,MixStore 和 BlueStore 基本表现出相同的性能,平均延迟也基本相同.这是因为对于 64 KB 以上的写请求,2 个系统的处理策略基本相同.较大的数据尺寸也导致请求处理的主要开销来自于 SSD 写入操作,软件栈的差异被进一步削弱.即便如此,由于具备更好的元数据操作并发性,MixStore 仍然表现出了更好的尾延迟,约为 BlueStore 的 54%~91%.

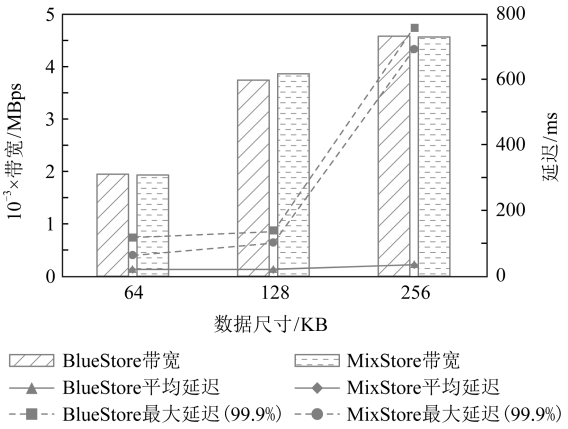


Fig. 6 Comparison of sequential write

图 6 顺序写测试对比

4.2.2 读性能

对于读操作,本次测试选择了与写操作相同的数据特征,即针对 2 KB,4 KB,16 KB 的随机读取,以及 64 KB,128 KB,256 K 的顺序读取场景进行测试.同样,每组测试完成后重启所有 OSD,以消除缓存的干扰.

对于小 IO 随机读,如图 7 所示,MixStore 和 BlueStore 有基本相同的 IOPS 和平均延迟,这是因为在排除缓存干扰后,读操作的开销主要来自于磁盘数据读取.而较大的写入总量也导致读操作更多的是从 SSD 而不是 PMEM 读取数据,因此不同数据大小的测试之间并未体现出差异.但得益于 MixStore 元数据访问的高并发性,在 16 KB 读取测试中,MixStore 的尾延迟表现出了 23%的降低.

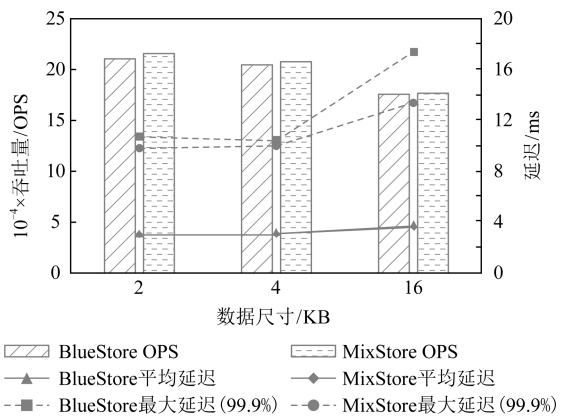


Fig. 7 Comparison of random read

图 7 随机读测试对比

对于大 IO 顺序读,如图 8 所示,受限于本次测试的网络环境,BlueStore 和 MixStore 未能体现出差别,仅 64 KB 顺序读 MixStore 体现出约 21%的尾延迟降低.128 KB 及 256 KB 的性能制约则更多来自于网络传输.

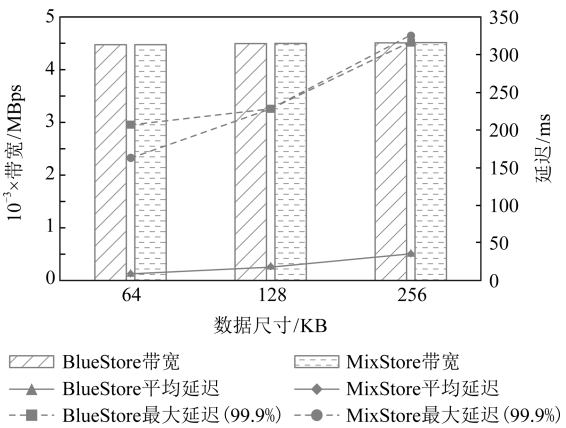


Fig. 8 Comparison of sequential read

图 8 顺序读测试对比

5 结 论

现有的分布式存储的后端存储存在着明显的写放大和 compaction 消耗.持久内存具备字节寻址,接近 DRAM 性能,掉电不丢数据等特性.但持久内存容量较小,更适合存放数据的元数据或小的数据的缓存,相反 SSD 容量较大,顺序读写性能更优,更适合存放大块的数据.MixStore 充分利用持久内存和磁盘各自的特性,把两者结合起来,通过构建 Concurrent SkipList 进行精细的索引元数据组织,以及通过惰性的 CoW 的数据存放机制,提供了高性能的后端存储,有效减少了写放大和避免了 compaction 带来的消耗.实验显示,相比 BlueStore,MixStore 的写吞吐提升 59%,写时延降低了 37%,有效地提升了系统的性能,并且具有更好的读写尾时延表现.展望未来,随着新型器件的发展和持久内存产业化水平的提升,计算、存储和网络能力都将显著提升,可能会出现基于持久内存等新型器件构筑的全新存储环境.我们需要重新审视现有的设计和优化机制,并设计新的算法来适配新的硬件环境.

参 考 文 献

[1] Aghayev A, Weil S, Kuchnik M, et al. File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution[C] //Proc of the 27th ACM Symp on Operating Systems Principles. New York: ACM, 2019: 353-369

[2] ONeil P, Edward C, Gawlick D, et al. The log-structured merge-tree [J]. Acta Informatica, 1996, 33(4): 351-385

[3] Intel. Intel Optane persistent memory [OL]. [2020-05-28]. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[4] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory [C] //Proc of the 22nd Symp on Operating Systems Principles. New York: ACM, 2019: 133-146

[5] Intel. Intel memory latency checker v3.9 [OL]. [2020-05-28]. https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html

[6] Google. LevelDB [OL]. [2020-05-28]. https://web.archive.org/web/20110816070240/http://www.readwriteweb.com/hack/2011/07/google-open-sources-nosql-data.php

[7] Facebook. RocksDB [OL]. [2020-05-28]. http://rocksdb.org

[8] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data [C] //Proc of the 7th Symp on Operating Systems Design and Implementation (OSDI'06). New York: ACM, 2006: 205-218

[9] Lakshman A, Malik P. Cassandra: A decentralized structured storage system [J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40

[10] Harter T, Borthakur D, Dong Siying, et al. Analysis of HDFS under HBase: A facebook messages case study [C] //Proc of the 12th USENIX Symp on File and Storage Technologies. Berkeley, CA: USENIX Association, 2014: 199-212

[11] Lu Lanyue, Pillai T S, Arpaci-Dusseau A C, et al. WiscKey: Separating keys from values in SSD-conscious storage [C] //Proc of the 14th USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2016: 133-148

[12] Kaiyrakhmet O, Lee S, Nam B, et al, Slm-db: Single-level key-value store with persistent memor [C] //Proc of the 17th USENIX Conf on File and Storage Technologies(FAST'19). Berkeley, CA: USENIX Association, 2019: 191-205

[13] Kannan S, Bhat N, Gavrilovska A, et al. Redesigning LSMs for nonvolatile memory with NoveLSM [C] //Proc of the 27th USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2018: 993-1005

[14] Xu Jian, Swanson S.NOVA: A log-structured file System for hybrid volatile/non-volatile main memories [C] //Proc of the 14th USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2016: 323-338

[15] Zheng Shengan, Hoseinzadeh M, Swanson S. Ziggurat: A tiered file system for non-volatile main memories and disks [C] //Proc of the 17th USENIX Conf on File and Storage Technologies (FAST 19). Berkeley, CA: USENIX Association, 2019: 207-219

[16] Wei Qingsong, Xue Mingdi, Yang Jun, et al. Accelerating cloud storage system with byte-addressable nonvolatile memory [C] //Proc of the 21st Int Conf on Parallel and Distributed Systems. Piscataway, NJ: IEEE,. 2015: 354-361

[17] Lu Youyou, Shu Jiwu, Chen Youmin, et al. Octopus: An RDMA-enabled distributed persistent memory file system [C] //Proc of the 26th USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2017: 773-785

[18] Lepers B, Balmau O, Gupta K, et al. KVell: The design and implementation of a fast persistent key-value store [C] //Proc of the 27th ACM Symp on Operating Systems Principles. New York: ACM, 2019: 447-461

[19] Intel. pmemkv [OL]. [2020-05-28]. https://github.com/pmem/pmemkv

[20] Harris T L. A pragmatic implementation of non-blocking linked-lists [C] //Proc of the 15th Int Symp on Distributed Computing. Berlin: Springer, 2001: 300-314

[21] David T, Dragojevic A, Guerraoui R, et al. Log-free concurrent data structures [C] //Proc of the 10th USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2018: 373-385

[22] Wang Tianzheng, Levandoski J, Larson P. Easy lock-free indexing in non-volatile memory [C] //Proc of the 34th IEEE Int Conf on Data Engineering. Piscataway, NJ: IEEE, 2018: 461-472

[23] Chen Juan, Hu Qingda, Chen Youmin, et al. A tiny-log based persistent transactional memory system [J]. Journal of Computer Research and Development, 2018, 55(9): 2029-2037 (in Chinese)
(陈娟, 胡庆达, 陈游旻, 等. 一种基于微日志的持久性事务内存系统[J]. 计算机研究与发展, 2018, 55(9): 2029-2037)

[24] Xiao Renzhi, Feng Dan, Hu Yuchong, et al. A survey of data consistency research for non-volatile memory [J]. Journal of Computer Research and Development, 2020, 57(1): 85-101 (in Chinese)
(肖仁智, 冯丹, 胡燏翀, 等. 面向非易失内存的数据一致性研究综述[J]. 计算机研究与发展, 2020, 57(1): 85-101)

[25] Intel. Persistent memory development kit [OL]. [2020-05-28]. https://pmem.io/pmdk/

[26] Fraser K, Harris T. Concurrent programming without locks [J]. ACM Transactions on Computer Systems, 2007, 25(2): 1-61

[27] Fraser K. Practical lock-freedom [D]. Cambridge: University of Cambridge, 2004

[28] Raju P, Kadekodi R, Chidambaram V, et al. PebblesDB: Building key-value stores using fragmented log-structured merge trees [C] //Proc of the 26th ACM Symp on Operating Systems Principles. New York: ACM, 2017: 497-514

[29] Hanson E. The interval skip list: A data structure for finding all intervals that overlap a point [C] //Proc of the 2nd Workshop on Algorithms and Data Structures. Berlin: Springer, 1991: 153-164

[30] Prout A. The story behind memSQL's skiplist indexes [OL]. [2020-05-28]. https://www.memsql.com/blog/what-is-skiplist-why-skiplist-index-for-memsql

[31] Yang Jian, Kim J, Hoseinzadeh M, et al. An empirical guide to the behavior and use of scalable persistent memory [C] // Proc of the 18th USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2020: 169-182

Tu Yaofeng, born in 1972. PhD candidate, senior member of CCF. His main research interests include cloud computing, big data and machine learning.

屠要峰,1972 年生.博士研究生,CCF 高级会员.主要研究方向为云计算、大数据和机器学习.

Chen Zhenghua, born in 1983. Engineer. His main research interests include storage systems and distributed systems.

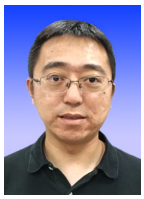
陈正华,1983 年生.工程师. 主要研究方向为存储系统和分布式系统.

Han Yinjun, born in 1977. Senior engineer. His main research interests include non-volatile memories, file system, and database system.

韩银俊,1977 年生.高级工程师.主要研究方向为非易失性内存、文件系统和数据库系统.

Chen Bing, born in 1970. PhD, professor and PhD supervisor. senior member of CCF. His main research interests include cloud computing, wireless communications and cognitive radio networks.

陈　兵,1970 年生.博士,教授,博士生导师,CCF 高级会员.主要研究方向为云计算、无线通信和认知无线网络.



Guan Donghai, born in 1981. PhD, associate professor. Member of CCF. His main research interest include machine learning, big data, and social computing.

关东海,1981 年生.博士,副教授,CCF 会员.主要研究方向为机器学习、大数据和社会计算.