计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2018.20180258
55(9):2038-2049,2018

一个基于日志结构的非易失性内存键值存储系统

游理通　王振杰　黄林鹏
(上海交通大学计算机科学与工程系　上海　200240)
(litong.you@sjtu.edu.cn)

A Log-Structured Key-Value Store Based on Non-Volatile Memory

You Litong, Wang Zhenjie, and Huang Linpeng
(*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai* 200240)

Abstract　Non-volatile memory (NVM) technologies are promising that would change the future of storage. NVM possesses many attractive capabilities such as byte addressability, low access latency, and persistence. It provides a great opportunity for the integration of DRAM and NVM in a unified main storage space. NVM could access data through the memory bus and CPU related instructions, which makes it possible to design a fast and persistent storage system in non-volatile memory. Existing key-value stores proposed for block devices implement NVM as block devices, which conceal the performance that NVM provides. A few existing key-value stores for NVM fail to provide consistency when hardware supports (e.g., cache flush) on power failures are unavailable. In this paper, we present a non-volatile memory key-value storage system, named TinyKV, which utilizes the log structure as its core framework. We propose a static concurrent, cache-friendly Hash table implementation using the characteristics of the key-value workloads. TinyKV separates the maintenance for data log of each worker thread in order to guarantee high concurrency. In addition, we implement the log structure technology for memory management and design a multi-tier memory allocator to ensure consistency. To reduce write latency, we reduce writes to NVM and cache flushing instructions by using cache flushing instructions. Our experiments demonstrate that TinyKV outperforms traditional key-value stores in both throughput and scalability.

Key words　non-volatile memory (NVM); log structure; key-value store; Hash table; memory management

摘　要　非易失性内存(non-volatile memory, NVM)技术是非常具有应用前景的计算机内存技术,将会对计算机存储层次结构产生极大的影响.NVM具有可字节寻址、可持久存储、低访问延迟等特点,这为DRAM和NVM在统一的主存储空间中的结合提供了巨大的机会.NVM可通过内存总线以及CPU相关指令进行数据访存,这使得在非易失性内存中设计快速的持久存储系统成为可能.现有的键值存储系统将NVM作为块设备使用,未能充分发挥NVM的性能.当硬件支持出现故障(例如高速缓存刷新)时,一些现有的键值存储系统无法保证数据的一致性.提出了一种基于日志结构的非易失性内存键值存储系统TinyKV,该系统利用键值数据负载的特性提出了一个静态并发、缓存友好的Hash表实现方案.TinyKV为每个工作线程维护单独的数据日志,以实现高并发性.此外,TinyKV采用日志结构技术进

收稿日期:2018-04-06;**修回日期**:2018-06-11
基金项目:国家重点研发计划项目(2018YFB1003302);国家自然科学基金项目(61472241)
This work was supported by the National Key Research and Development Program of China (2018YFB1003302) and the National Natural Science Foundation of China (61472241).
通信作者:黄林鹏(lphuang@sjtu.edu.cn)

行内存管理,设计多层级内存分配器,以保证一致性.此外,系统通过减少对 NVM 的写入与缓存刷新指令,以降低写入延迟.实验显示:与传统的键值存储系统相比,TinyKV 具有良好的吞吐性能与扩展能力.

关键词 非易失性内存;日志结构;键值存储;Hash 表;内存管理

中图法分类号 TP391

近年来,随着大规模数据分析处理、数据密集型计算等技术的兴起和发展,内存计算也成为了更加重要的基础技术.这些技术都依赖于快速大容量的存储.除常见的磁盘、SSD 和 DRAM 外,非易失存储器(non-volatile memory, NVM)技术正在快速发展,例如相变存储器(phase-change memory, PCM)[1]、自旋矩存储器(spin-torque transfer ram, STT-RAM)[2]和阻变存储器(resistive ram, ReRAM)[3]等.这类存储介质所共有的特点包括可字节寻址能力、接近 DRAM 的读写延迟以及数据的可持久性能力.DRAM 和 NVM 可以抽象为统一的主存储空间,这种新的混合内存架构为建造更高速有效的大容量持久性存储系统(如文件系统、数据库等)带来了新的机遇.

在软件技术方面,以 LevelDB[4],Dynamo[5]为代表的键值存储系统已经是业界使用的主流系统,这类存储系统在非结构化数据存储方面有很好的性能优势.现有较为成熟的键值存储系统都是针对磁盘或 SSD 进行优化设计实现的,它们依靠文件系统来存储数据,并进一步支持数据持久化.此外,针对非易失内存已有一些文件系统(如 HMFS[6-7],PMFS[8],NOVA[9],Octopus[10]),但是依靠文件系统来存储数据,需要执行文件系统的代码,并且操作系统也要进行用户态和内核态之间的切换,这些都会带来一定的开销.由于非易失性内存的延时接近于 DRAM,这种用户态/内核态切换开销对存储系统的性能的影响是不能忽略的.在基于 DRAM 内存存储系统方面,键值存储也是研究热点,如在实际中被广泛应用的 Memcached[11]、性能非常好的 MICA[12],但针对 DRAM 设计的系统无法顾及到非易失性内存的特性,如 NVM 的读写延时的差异、DRAM 与 NVM 访问速度差异等.因此,综合 DRAM 和 NVM 各自特点开发新型的系统需要新的设计思路和实践.

在使用 NVM 设计存储系统时,传统的基于文件系统和基于磁盘、DRAM 或 SSD 的键值存储系统存在 4 个问题:

1) DRAM 和 NVM 混合架构下内存分配困难.针对 DRAM 设计的键值存储系统未考虑到非易失性内存的特性,如非易失性内存的读写延时的差异、DRAM 与非易失性内存访问速度差异等因素.

2) 现有键值存储系统数据一致性保障机制性能开销较大,且未充分利用 NVM 的非易失特性.传统架构中,多数系统只基于 DRAM 设计并维护一个数据日志,所有数据都是以写日志的方式进行组织,这样导致的一致性维护开销较大,NVM 的特性很难被充分利用.

3) 垃圾回收复杂.NVM 具有读写不对称的特性,因此 NVM 设备使用一段时间后需要垃圾回收.在日志结构的存储系统中,数据的更新采用写时复制的方式进行,旧数据保留在原始位置,形成空间碎片,不能被有效地重新利用.使用 NVM 设备进行垃圾回收时,需要整合内存碎片,整个过程复杂耗时.

4) NVM 设备的并发性能难以得到充分的利用.传统键值存储系统针对 DRAM 设计并发机制,这种情况下,频繁的读写操作会限制 NVM 的性能.

基于以上 4 点考虑,本文提出一个基于日志结构和非易失性内存构建键值存储系统的方法,并以 PCM 为代表的存储类内存(storage class memory, SCM)为存储介质实现一个键值存储系统 TinyKV,其充分考虑了键值数据工作负载等特性以及非易失性内存的特点,采用日志结构技术最大化非易失性内存的吞吐性能,并提供可扩展能力和数据一致性保证.

本文的主要贡献总结有 3 个方面:

1) 存储系统功能分区与索引结构的设计.通过对非易失性内存的抽象,只需要用户重新定义并实现与设备有关的数据结构,就可使用存储系统,从而实现存储系统与设备模型的解耦.通过针对键值数据特性的分析,设计与实现 Hash 表的结构,使用地址连续的数据结构高效充分利用缓存,使用 Hash 表对所有的数据对象进行索引.对整个设备空间进行功能分区,确定并记录各区域的范围功能.

2) 非易失性内存分配器的实现.结合非易失性内存的功能分区,设计多层次的非易失性内存分配器,通过不同层次的功能分离,在不同层次使用不同的数据结构,实现一个能够充分结合非易失性内存和 DRAM 优势的内存分配器.

3) 存储系统垃圾回收算法的设计与实现. 存储系统垃圾回收算法以块为单位对日志中的剩余空间进行回收. 通过存储功能区的块状态能够生成垃圾回收算法所需数据结构,把这个数据结构存放到DRAM中,以提高系统运行效率并减少对NVM的占用.

1 相关工作

键值存储系统是传统的关系型数据库的简化形式,通过去除不必要的特性,键值存储系统可提升自身性能,降低数据间的耦合度.

1.1 传统键值存储系统

LevelDB是Google开发的一套针对块设备进行优化的存储系统,其核心使用了日志结构合并树(log-structured merge tree, LSM-Tree)[13]的数据结构. LevelDB的优点是写操作性能很强,缺点是读操作性能表现不佳且容易造成严重的写放大. MICA是一个具有很强扩展能力的内存键值存储系统,通过Hash索引结构把所有的键进行分区,每个处理器核心负责一个分区的读写工作. MICA的内存分配器使用了非严格的循环日志结构的分配方式,并对垃圾回收工作进行了新的解释. 在MICA提供的存储模式中,它使用类似于分离适配(segregated fit)的方式管理内存,具有较高的内存空间使用率. 当内存碎片较多时,MICA可能满足不了大内存的需求. PebblesDB[14]介绍了一种基于碎片化的日志结构合并树(fragmented log-structured merge trees, FLSM)的键值存储系统,其通过设计新型数据结构FLSM,降低了传统基于LSM-Tree的键值存储系统写放大严重的问题,有效地提高了系统整体的读写性能.

1.2 基于非易失性内存的键值存储系统

Echo[15]是一个针对非易失性内存设计的持久性键值存储系统. 它利用DRAM和非易失性内存的两级存储结构,把数据缓存到每个线程的DRAM中. 每个工作线程可以把数据提交到主线程的队列里,由主线程负责把DRAM中的数据存储到非易失性内存的存储系统中,当读写比较频繁时,可能会有多个主线程同时运行. 它通过快照隔离的方式保证数据一致性,并假设计算机硬件可以提供数据持久化的帮助. UDORN[16]是一个基于NVM的持久化内存键值存储数据库的新型设计框架,在UDORN中,NVM上的持久数据库被用来完成常规内存数据库和持久化存储图像的功能. 在运行时期间,持久性数据库被映射到进程地址空间,这些操作通过相应的地址空间直接在NVM上执行. 与传统基于NVM Library的Redis系统相比,应用UDORN的Redis获得了6倍的性能提升. NVMKV[17]是一个闪存感知键值存储器,它依靠闪存转换层(flash translation layer, FTL)[18]功能在关键值存储处实现最少的数据管理,还有一些针对非易失性存储器优化的键值系统. NVMcached[19]是非易失性存储器的键值缓存,它尝试通过使用校验和来剔除损坏的数据以避免大多数缓存溢出. 它充当后备缓存,并且任何丢失的键值对象都需要被重新插入其中. HiKV[20]在NVM与DRAM的混合内存架构中构建混合索引,利用Hash索引和B^+树索引的特性,提供更有效的键值操作,以保持其固有的快速索引搜索能力,避免长时间NVM写入以保持2个索引的一致性. 此外,HiKV将差异并发方案应用于混合索引,并采用有序写入一致性来确保崩溃一致性. LibreKV[21]使用静态Hash表和动态Hash表来实现系统性能和内存利用率之间的平衡. 其采用基于校验和的一致性机制来保证数据一致性和永久存储在NVM上. 与传统的键值存储系统相比,LibreKV独立工作,不依赖于底层文件系统,简化了读写I/O操作. NVHT[22-23]是一个基于Hash表结构设计的键值存储系统,它针对NVM进行了多项优化,包括非易失指针结构的设计、NVM数据一致性的undo日志解决方案以及结合磨损均衡(wear-leveling)算法的内存分配器设计.

1.3 基于非易失性内存的文件系统

目前许多文件系统的研究工作都致力于在NVM上提供不同的抽象, Mnemosyne[24], NV-Heaps[25]和NVML[26]提供了通用编程接口和持久性内存和事务机制以实现故障恢复;PMFS,NOVA和Octopus是针对非易失性内存优化的文件系统,它们允许通过POSIX文件接口对传统的基于文件的内存进行访问. PMFS使用具有不同CPU指令的多粒度原子更新和用于元数据一致性的细粒度日志记录以及用于数据一致性的写入时复制. SIMFS[27]是一个基于NVM的可持续内存文件系统,其充分利用了文件访问路径上的内存映射硬件. SIMFS利用一个新型设计框架:文件虚拟地址空间(file virtual address space),进行文件读写操作处理. 在这个框架内,当文件被打开时,其文件虚拟地址空间会被嵌入所对应的进程虚拟地址空间,之后的文件访问由内存映射硬件处理.

2 TinyKV 系统架构

本文提出了一个基于非易失性内存的键值存储系统 TinyKV.其充分考虑了键值系统对象数据比较小并且大小分布比较集中等特性,对存储系统进行优化设计,通过日志结构的内存管理方式对非易失性内存的磨损均衡也起到了一定的帮助作用.

TinyKV 的整体架构如图 1 所示.为了支持高并发的数据服务, TinyKV 允许多个线程同时访问键值对象.为减少线程间的资源干扰,TinyKV 为每个工作线程设置一个单独的分配器及数据日志.这些分配器被放在 DRAM 上以支持快速分配.所有线程共享一个全局 Hash 表来索引键值对象.Hash 表使用动态分配的数组来解决 Hash 冲突.使用数组而不是列表可以通过 CPU 的硬件预取来加速针对键值对象的访问.TinyKV 通过全局静态 Hash 表来索引数据,并提供插入、读取和删除 3 个应用接口来管理数据.

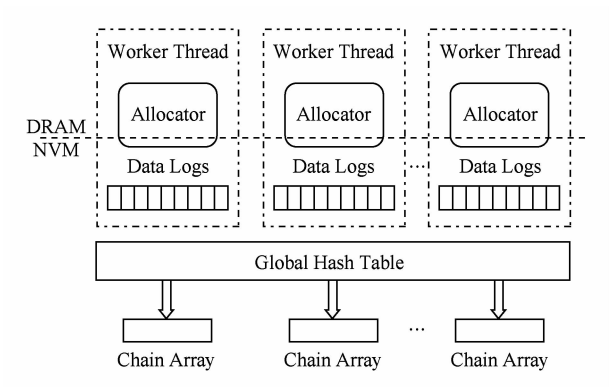

Fig. 1 Architecture of TinyKV

图 1 TinyKV 架构图

在内存分配器设计中,TinyKV 中的每个线程都有自己的内存分配器,这可以减少同步原语引起的开销.TinyKV 使用多级内存分配模型来有效地管理内存,并使用原子写操作保证每一个数据元素的写操作是一次独立的事务,从而保证操作的原子性[8,28],整个存储空间首先采取分块的方式进行管理,然后把分配的块按功能进行划分.

TinyKV 日志和 Hash 表都存储在 NVM 中.日志是用来保障系统的数据一致性.它首先将键值对象添加到日志记录中;然后才将它们插入到 Hash 表中,以完成真正的持久化过程.TinyKV 可以在故障恢复时扫描日志元数据信息来了解日志空间的使用情况.

此外,TinyKV 在 NVM 中存储系统全局静态 Hash 表和动态链.传统的动态 Hash 表的大小根据记录的数量适度增长和收缩,当 Hash 表需要扩容时,这个过程会导致一定的时间与空间开销.根据对 Facebook 负载数据特点的分析[29],可以得知键值数据的大小分布比较集中并且比较小,例如存储 SYS(系统数据)的键(key)的大小有 90%小于 40 B,值(value)的大小有 80%在 500 B 左右.TinyKV 根据设备空间的大小,估算预测出整个设备能够容纳多少键值对象.根据对象数目计算 Hash 表的初始大小,通过大 Hash 表减小了载荷因子,使 Hash 表产生冲突的可能性减少,减少 Hash 表扩容的次数,以规划合适大小的 Hash 表空间给数据对象.凭借这一手段,TinyKV 可以使用静态 Hash 表作为存储数据的索引.为了减少静态分配的 Hash 表的大小,系统设定 Hash 表存储区的基本组成单位是大小为 64 b 的字.每个字包含一个指向键值对象或 Hash 表链的指针.TinyKV 所维护的全局 Hash 表,在没有 Hash 冲突发生时直接访问键值对象,而当冲突发生时,它则将动态数组分配为数组链.

3 TinyKV 设计与实现

本节主要介绍 TinyKV 的设计考虑和具体的实现技术.首先,我们介绍 TinyKV 的存储区功能布局;接下来介绍一种支持并发并且缓存友好的 Hash 表,是 TinyKV 设计的重点;然后介绍内存分配器的详细设计与 TinyKV 多线程中的应用;最后,给出 TinyKV 的垃圾回收算法.

3.1 存储区功能布局

图 2 是 TinyKV 对非易失性内存的功能分区.类似于操作系统,所有的内存空间以块为基本管理

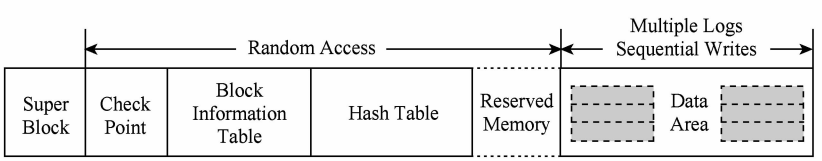

Fig. 2 TinyKV memory layout

图 2 TinyKV 非易失性内存功能分区

单位,然后把这些块根据不同功能进行划分.每个块的大小是 2 MB,每个块只允许存储一种类型的信息,如 Hash 表的数组链或键值对象.存储数据的块使用顺序写的方式,存储 Hash 表的块使用随机写的方式.

1) 超级块区(super block)记录了 TinyKV 的参数信息和对功能区的划分,如存储空间的大小、块的大小、块的数量、各个分区的起始块编号和块数目.这些内容在第 1 次创建时确定并且不能被修改,它只会在 TinyKV 启动时被读取.

2) 系统状态区(check point)记录了 TinyKV 的运行状态的数据结构,如上次启动时间、上次运行时的线程个数、每个线程日志所在的块编号、Hash 表大小、空闲块数等.每次系统启动都会产生一个新的状态,这些状态保留在一个循环数组中.在系统启动时,TinyKV 会读取最新的状态信息,了解 Hash 表是否经过了扩容,扫描每个线程日志以保证数据的一致性,最后产生一条新的状态信息,新的状态信息可能会覆盖最旧的状态信息.

3) 块状态区(block information table)描述了非易失性内存的每一个块的信息.块状态区中的元素包含与其对应块的信息,例如键值对象或数组链的数量、是否为空闲块以及块修改的时间.系统将设置一个链表把所有的空闲块链接起来,如果当前块为空闲状态,那么它将被链接至链表中.为了减少块状态区对内存的占有率,一些存储区域将会被复用,并将块被访问时的状态信息只存留在 DRAM 中.每个块状态大小为 8 B,每个块的大小为 2 MB,其对存储内存的占有率为 $8/2^{21}=0.0004\%$.

4) Hash 表区(Hash table)是 TinyKV 的索引结构,每个桶包含指向键值对象的指针、一个标志位记录是否有线程在修改这个桶以及桶里对象的个数等信息.当桶中包含多个对象时,它的指针是一个数组,这个数组包含了所有属于这个桶的键值对象的地址.在 TinyKV 中的存储地址都是一个 48 b 的无符号整数,它是对象所在位置相对于内存起始地址的偏移量,运行时地址可以通过存储地址加上内存起始地址的偏移量得到,Hash 表的大小是 2^N,这样可以根据位运算来计算键值对象将要插入的位置.

5) 保留内存区(reserved memory)紧跟在 Hash 表后面,通过它可以很方便地对 Hash 表扩容.当数据存储区内存不足时,它还可以用来存储数据对象.对 Hash 表扩容是从保留内存区的块从前向后分配空间,当用来存储数据对象时,保留内存区的块从后向前开始分配.

6)数据存储区(data area)是为了存储键值对象,或者包含键值对象地址的数组.数据存储区的每个块只存储一种类型的数据.当存储区域的内存不足时,可以向保留内存区申请空间.

3.2 静态并发、缓存友好的 Hash 表

图 3 展示了 TinyKV 的 Hash 表结构.在图的最左边是 Hash 表区域的结构,可以将其看成一个数组.最右边是 TinyKV 存储的键值对象.中间是为了解决 Hash 冲突而动态分配的数组(称为数组链),这些数组里存储的是键值对象的地址.TinyKV 通过计算出 key 的 Hash 值找到对应的桶,然后根据桶里的内容确定键值对象或者数组链的地址.如果桶里只有一个键值对象,那么它包含的是键值对象的地址;如果桶里有多个对象,为了解决冲突,它包含的地址是数组链的地址,数组链则包含相应对象数据的地址.

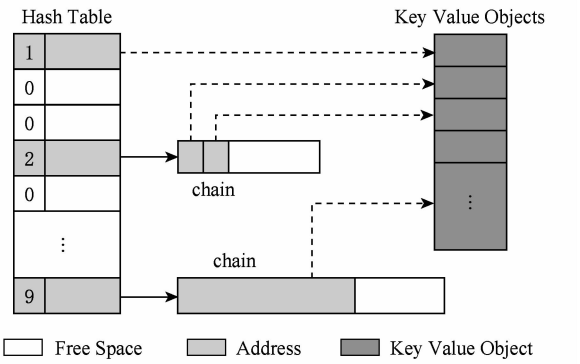

Fig. 3 TinyKV Hash table structure

图 3 TinyKV Hash 表结构图

为了减少 Hash 表区域的大小,TinyKV 对桶的大小做了限制,其定义如图 4 所示.Hash 表中每个存储桶是大小为 64 b 的字.其中,*has_writer* 表示是否有线程在修改这个桶.TinyKV 允许多个线程进行读操作的同时,最多有一个线程可以修改同一个桶,它由原子操作设置或清除.在并发访问 key-value 的模式下,读事务的顺序具有一定的随机性,如果一个线程 A 修改 *key*1 的同时,另一个线程 B 进行读 *key*1,它保证线程 B 读到的数据是新数据或是修改前数据,但如果线程 A 在修改 *key*1 之后继续读取 *key*1,其读取的数据是新数据.每个写线程在访问任一个桶的同时,它必须检查 *has_writer* 是否为 0.如果为 0,它把 *has_writer* 置 1,然后再访问该桶;如果为 1,它就等待.为了保证只有一个线程

可以修改同一个桶,必须对 *has_writer* 进行保护. TinyKV 使用 CAS 原语保证互斥,它相当于一个自旋锁的作用,读线程是不需要读写这个锁的. TinyKV 还使用了事务机制,保证写操作是原子操作,从而保证读的数据只能是修改前或修改后的数据,从而可以保证读操作的一致性.

```
struct HashTable{
① uint64_t has_writer:1;
② uint64_t version:3;
③ uint64_t reserved:4;
④ uint64_t nr_objects:8;
⑤ uint64_t nvm_addr:48;
⑥ } g_htable[0].
```

Fig. 4 Definition of TinyKV Hash table

图 4 TinyKV Hash 表的定义

第 3 个字段 *reserved* 表示位于存储桶中的键值对象的数量. *version* 是对桶分裂次数的统计. 当 Hash 表进行扩容时,会有一个变量记录它扩容的次数. 每次对 Hash 表进行扩容,在 Hash 表里的桶就会产生一次分裂. 当它到达最大值或者保留内存区剩余资源不足时,扩容操作会失败. *nr_objects* 表示桶里包含的数据对象个数,每个桶最多容纳 255 个元素. *nvm_addr* 是非易失内存内的一个对象的地址. 当 *nr_objects*=1 时,它指向一个键值对象;当 *nr_objects*>1 时,它指向一个数组链,数组链包含键值对象的地址.

数组链是一个动态分配的数组,其大小与起始地址均为 *cache_line* 的整数倍,它的内存是从数据存储区分配的. 数组中的每个元素指向一个键值对象. 元素的数据格式如图 5 所示. 每个元素大小是 64 b,其中低 48 b 的 *nvm_addr* 存储键值对象的地址,高 16 b 的 *tag* 是对 *key* 进行数字描述,它可以用 *key* 的 Hash 值高 16 b 来表示. 在访问键值进行匹配时,需要对 *key* 的内容进行逐一匹配. 当 TinyKV 进行查找时,它会先匹配 *tag*,然后再根据 *nvm_addr* 访问键值对象. 若 2 个键值对象具有相同的 *tag*,再继续对其 *nvm_addr* 进行对比,直至找到所需对象数据. 2 个不同的 *key* 具有相同 *tag* 的概率是 $1/2^{16}=0.0015\%$. 故首先比较 Hash 值高16 b的 *tag*,将会极大地减少了读取匹配字符串的次数.

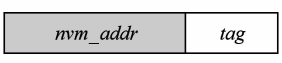

Fig. 5 Layout of a chain element

图 5 数组链数据结构图

为了遍历一个链表,CPU 要先把数据从内存中加载到高速缓存中,因为链表是结点不连续的存储结构,CPU 访问内存的次数是不确定的,使用地址连续的数组,可以将 CPU 访问内存的次数降到最少. 若硬件预取生效,会进一步降低 CPU 访问内存的次数. TinyKV 没有明确对桶中的元素数量进行限制,当桶中元素数超出原始设定大小,TinyKV 从数据存储区分配一个新的数组,其大小为原始数组的大小与一个 *cache_line* 之和,分配完毕后,将原始数组的数据复制到新的数组中,并把相应的桶的 *nvm_addr* 设置为新数组的地址,虽然这可能会造成某些桶元素较多,但这个问题可以通过现有的比较成熟的 Rehash 机制来解决[30].

3.3 内存分配器

TinyKV 对 NVM 进行分块管理,其优势是支持高并发分配. 如果对整块 NVM 进行统一管理,则 NVM 块元数据信息只有一份,则每一次 NVM 分配都将会锁定整块 NVM 空间的元数据,而不能同时分配多块 NVM 区域给多个用户,因此,键值对象不可跨块存储. 系统采用分块管理后,每块 NVM 空间有自己独立的元数据信息,从而可以并发地进行修改. 基于对 Facebook 负载数据特点的分析,可以得知,一般的键值对象所占用空间较小,内存分配器只需分配一个块,即可满足键值对象的存储空间要求. 由于分配内存过大或过小都会造成较大的瓶颈,分块过大,每个块内的数据元素较多,难以支持高并发的访问;分块过小,块元数据所占的空间开销较大. 因此,TinyKV 将分配的块的大小限制在 2 MB 以内(块的大小). 当块空间不足时,需要重新分配新块,旧块剩余空间可以继续用于分配给之后存储的较小数据元素,这样虽然会造成一定程度上的内存碎片,但由于键值对象本身所占用空间较小,所以内存分配过程中空间浪费情况不是很严重.

TinyKV 中的每个线程都有自己的内存分配器,这可以减少由同步原语引起的开销. TinyKV 使用多级内存分配模型来有效地管理内存. 如图 6 所示,TinyKV 的内存分配器采用 3 层结构进行描述,一方面能够独立定义各层的功能,另一方面各个层次的结构可以根据需要放在 DRAM 或非易失性内存中. 第 1 层为块管理层,它保留一个空的内存块池,分配和释放块;第 2 层是日志分配层,每个线程都有各自的日志分配器. 分配器在分配新日志时需要底层的块,如果块变空,它会将块释放到底层;第 3 层是对象分配层,该层是为链或键值对象分配内存的对象分配器,这些对象分配器无法重复使用日

志尾部之前的空间,只能利用来自日志尾部的内存.因此,在删除键值对象时,日志中存在许多不可复用的碎片.为了复用这些内存碎片,分配器需要执行垃圾收集.垃圾收集机制将被废弃的日志中的一些尚在使用的键值对象移动到新的日志中,并将废弃日志块释放到块管理层.

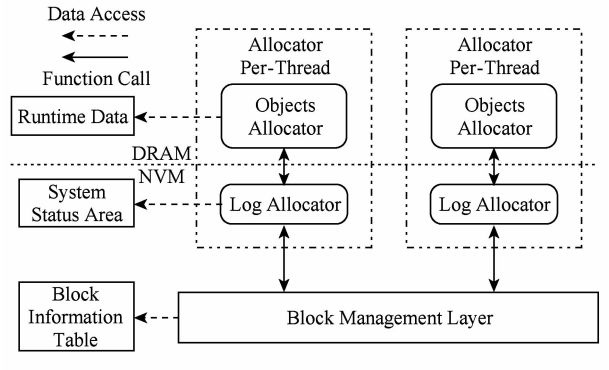

Fig. 6 NVM memory allocators

图 6 NVM 内存分配器

TinyKV 内存分配器的 3 层结构详细描述如下:

1) 块管理层.它负责整个非易失性内存的块区域内存分配管理.它的主要数据结构是由块状态区的数组组成,所以对它的修改要保存到非易失性内存中.所有线程共享一个块管理层,它负责记录块的类型、块内有效数据的大小、自由链表的维护等.它提供的功能有:分配块、释放块、修改块状态.由于块管理层的数据结构可以由所有线程同时修改,所以需要一种同步机制保证数据的一致性,TinyKV 使用 CAS 操作来替代互斥锁中以提高并发的性能.

2) 日志管理层.一个日志最大的大小是一个块的大小.每个线程都有一个独立的日志分配器,日志分配器所维护的数据只能被所有者线程进行修改,所以不需要互斥锁进行数据同步.由于数据存储区存储键值对象和数组链对象 2 种数据,所以日志也有 2 种类型,一种是键值数据日志,另一种是数组链对象日志.在系统状态区有记录各个线程正在使用的日志的块编号的信息.

3) 对象管理层.它在日志尾部为键值对象或者数组链分配内存空间.这些对象分配器,无法重复使用日志尾部之前的空间,只能利用来自日志尾部的内存.因此,在删除键值对象时,日志中存在许多不可复用的内存碎片.为了重用这些内存碎片,分配器需要执行垃圾回收.垃圾回收机制将被废弃的日志中的一些尚在使用的键值对象移动到新的日志中,并将废弃日志块释放到块管理层.

对于每个块,块信息表(block information table, BIT)中都有一个记录块使用情况的元素.对于写操作密集型工作负载,块使用情况经常发生变化.为了减少对 NVM 的写操作次数,日志分配器将块区域信息复制到 DRAM,因此块使用的更新情况保存在 DRAM 中.当块区域存储达到上限时,日志分配器将块信息转存到 NVM,并使用 *clwb* 或 *clflushopt* 指令将相关日志数据持久保存至 NVM,并要求内存池中新的块区域启动新日志.分配内存时,TinyKV 使用 Lock-Free 的更新操作.由于每个线程日志,没有多个线程将键值对象附加到同一个日志中;但是,当 2 个工作线程想要删除同一个日志中的键值对象时,块使用信息会被这 2 个线程同时修改.所以块信息数据的更新操作必须是原子的,以此保证一致性.

3.4 多线程应用

TinyKV 可以通过多线程技术对系统处理用户请求的能力进行扩展.在多线程编程中一个最常见的问题是数据同步,比较常用的数据同步技术有互斥锁、条件变量以及读写锁等.一般来说,当发生数据竞争时,锁的使用会带来较大的系统开销.例如 *futex* 会在可能发生数据竞争时,使应用程序发生用户态和内核态的转换.由于 TinyKV 的数据更新使用的日志结构,所以数据竞争的情况是不会发生的.但由于 Hash 表的数据是原地更新,因此要对 Hash 表的结构进行数据同步操作,对 Hash 表的每个桶进行锁保护.

TinyKV 实现了一种允许多个读者和一个写者同时对一个桶的数据进行访问的操作.它对读者不加约束,而写者在修改桶里的数据时,需要申请写锁.由于 TinyKV 的数组链存储的是指向键值对象的地址,每个地址是 48 b 的,而 CPU 对数据对齐的 64 b 字的更新是原子的.当一个写者与多个读者同时对一个 key 进行访问时,读者看到的要么是写者更新后的数据,要么是写者更新前的数据.

由于 TinyKV 的日志结构,若未立即删除旧的数据,不会出现悬垂指针.与锁相关的信息可以放在 DRAM 中,也可以放在非易失性内存中.如果将它们放到 DRAM 中,可以提高运行效率,但是当 Hash 表很大时,将占用大量的 DRAM.如果将它们放入到非易失内存中,加锁信息可能会持久存储到 NVM 中,下次系统启动时,需要检测并释放所有存储的加锁信息,因为存储加锁信息没有意义.

3.5 垃圾回收算法

在日志结构的存储系统中,回收被删除对象空

间的有效方法是垃圾回收算法,垃圾回收算法的时机和策略对性能的影响同样重要. TinyKV 使用垃圾收集线程来回收键值对象或链式数组被删除时累积在日志中的空闲内存. TinyKV 启动后,线程扫描 BIT 以了解块的使用情况,然后使用与 LFS[31] 类似的成本收益方法来选择废弃的日志. 根据日志中的可用空间量和日志的修改时间选择需要废弃的日志. 对于每个所选日志,垃圾收集线程都会扫描存储在日志中的键值对象,将活动对象复制到新日志并将其重新插入到 TinyKV 中,然后它将被废弃的日志内存空间释放到内存池中,使得其所占用的内存可用于新日志.

TinyKV 的垃圾回收算法,包括存储系统内垃圾的产生及特点、影响垃圾回收性能的因素、垃圾回收的时机以及垃圾回收算法. 在 TinyKV 中,当数据更新时,新数据被追加至日志的尾部,而旧数据的引用在 Hash 表中被替代,使得旧数据成为失效数据,因此产生了垃圾数据. 为了减少垃圾回收的代价,有效数据的占比越小越好. 随着时间的推移,旧日志中的数据被修改的可能性越大,从而有效的数据量越来越少. TinyKV 触发垃圾回收时机有 2 个:1)当日志内的有效数据为零时;2)当 TinyKV 中剩余块不足 10%时. 当日志内有效数据为零时,块在代价很小的情况下立即被回收,回收的块被插入到剩余块的链表尾部. 在第 2 种情况下,TinyKV 扫描整个块状态表,根据块的有效数据量和块上次被修改时间计算花费收益比[30],根据收益比的数值建立一个最小堆. 针对那些日志内有效数据为零的块,TinyKV 的垃圾回收机制可以对其进行异步回收,从而提升垃圾回收的效率. 由于在回收的过程中,回收块中的有效数据非常少,因此 TinyKV 的垃圾回收效率较高.

在扫描全局块状态表,建立起最小堆后,选择堆顶的块,把其中的有效数据移动到垃圾回收线程自己的日志中. 垃圾回收算法通过扫描被选中块的所有数据对象,计算对象的 Hash 值,通过 Hash 表进行查询数据对象是否还被引用. 如果正在被引用,说明它是有效数据;否则说明它是垃圾对象. 由于日志是不允许修改的,并且数据对象不能使用反向指针指向引用它的桶或数组链,故在数据被修改或删除时,TinyKV 不能通过标记来记录数据修改或删除的信息,所以在扫描时不能单纯依靠数据本身识别它是否为有效数据. 最后更新 Hash 表中关于更新被移动数据的索引信息的操作,流程大致与插入操作相似,不同点在于要比较 Hash 表中的关于这个数据对象的索引信息是否相同. 若不同,说明在垃圾回收的过程中,这个数据对象已经被其他线程进行了修改,那么被移动的对象即可删除;若相同,把索引信息更新到数据对象所在的当前位置.

3.6 数据持久化

目前一种比较有效的方式是把非易失内存和 DRAM 一样映射成写回的方式(write-back),通过特殊的 CPU 指令来保证修改的数据能够最终到达非易失性内存中. *clflush* 是比较传统的刷新高速缓存行的方式,它把数据从高速缓存中写回内存中并使高速缓存行失效,多个 *clflush* 的顺序是固定的;*clflushopt* 是对非易失性内存进行优化而提出的指令,会使高速缓存行失效,但这个指令是无序的. 通常情况下,*clflushopt* 的性能比 *clflush* 要好,适用于在比较大的数据中使用. *clwb* 同样是对非易失性内存进行优化而提出的指令,不同于 *clflushopt* 的是它不会使高速缓存行失效. 不过,这些指令不能保证数据能够到达非易失性内存上,必须使用内存屏障原语(*sfence*)明确地表示要等待这些指令的完成. 在 TinyKV 中,只有当日志满时,才会使用 *clflushopt* 或 *clflush* 对整个日志的数据持久化;当修改一个数据对象时,持久化必要的元数据信息,同时计算存储数据的校验码,以便在系统恢复时能够回退到一致状态.

当 TinyKV 正常关闭时,为了快速恢复数据,TinyKV 会将所有当前活动日志和一些数据结构(例如分配器)转移至 NVM. 在 TinyKV 异常关闭的情况下(例如系统崩溃),TinyKV 必须恢复到一致状态,并通过扫描数据日志来重建一些数据结构,这个过程较为迅速. TinyKV 快速恢复的过程具体描述为:1)检测 TinyKV 日志的大小,正常情况下,TinyKV 日志的大小不会大于块的大小;2)当日志已满时,日志中的键值对象通过内存分配器将数据持久化保存到 NVM,但可能会存在尚未来得及回收的日志;3)TinyKV 为每个可能的未回收的日志启动一个恢复线程,其会扫描其日志中的所有键值对象,记录第 1 个损坏对象,将日志的尾部设置为该对象的地址,并将剩余对象故障前的旧地址压入恢复堆栈;4)通过恢复线程,所有堆栈中的对象将会回滚至故障前状态.

4 实验与结果

4.1 实验环境

实验设备如表 1 所示,本次实验中,实验所用的

计算机是 Dell-R730,操作系统内核版本是 Linux 2.6.32.机器装备了 2 个 10 核的 CPU,其型号参数是 Intel® Xeon E5-2650-V3,2.3 GHz,Ivy Bridge EP.每个 CPU 的 L3 缓存是 25 MB,每个核的 L2 缓存是 10×256 KB.机器上装备了 128 GB 的 DDR3 DRAM,使用 DRAM 模拟 NVM.

Table 1 Experiment Configuration

表 1 实验环境配置

OS	CPU	L2 Cache/MB	L3 Cache/KB	DDR3 DRAM/GB	Model
Linux 2.6.32	Intel® Xeon E5-2650-V3, 2.3 GHz	25	10×256	128	Dell-R730

1) 对比系统.本次实验对 TinyKV,LevelDB 和 Masstree[32] 的性能做了比较.LevelDB 需要文件系统存储数据,我们在内存中创建一个文件系统,并把 LevelDB 的数据文件存放到这个文件系统中.Masstree 是一个键值内存存储系统,使用 B^+ 树作为系统的索引结构,类 Trie 的数据结构对 B^+ 树进行连接.为了减少数据访问的延迟,它大量采用了数据预取技术.

2) 测试数据集.本次数据采用的数据集是用 YCSB[33] 性能测试工具集生成的.数据集的类型和大小等信息如表 2 所示.数据集的类型根据数据的分布特征可分为 2 种:均匀(uniform)分布和偏斜(skewed)分布.均匀分布的数据是等概率随机生成的;偏斜分布的数据是根据 Zipfian 分布生成的,其数据分布的特征是某些数据比大多数数据的出现次数要高很多.根据大小和数据分布,我们生成了 4 个数据集,其特点如表 2 所示:

Table 2 Workload Figures

表 2 测试数据集特征

Workload	Key Size/B	Value Size/B	Operation Count
Small Uniform	16	100	64×10^6
Big Uniform	128	512	8×10^6
Small Skewed	16	100	64×10^6
Big Skewed	128	512	8×10^6

4.2 读写性能

本节将对比各系统单线程的读写性能.对于每一类型的数据集合,在进行写性能测试时,每一次测试,我们都重新建立一个存储系统,通过持续不断地插入数据,得出实验结果.在进行读性能测试时,所有的存储系统都预先存储了 6 400 万个数据,通过连续地进行数据查询,得出实验结果.系统的单线程写性能测试结果如图 7 所示:

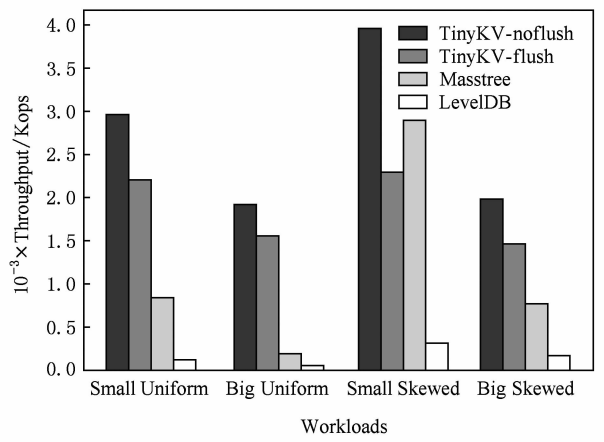

Fig. 7 Single thread write throughput

图 7 写性能测试

在 TinyKV-flush 模式中,TinyKV 通过 *clflush* 和内存屏障原语对元数据进行持久化.在一个日志空间写满时,对整个日志的数据进行持久化,针对未来得及持久化的日志内的数据,依靠系统恢复时检错并回滚到一致性状态.在 TinyKV-noflush 模式中,不使用刷新高速缓存的指令进行数据持久化.通过对比 2 种模式的实验结果揭示数据持久化对系统性能的影响.在测试 LevelDB 和 Masstree 时,我们不使用相关指令对数据进行持久化操作.在 TinyKV-noflush 模式下,TinyKV 的写性能优于 LevelDB 和 Masstree.当进行数据持久化操作时,即在 TinyKV-flush 模式中,TinyKV 的系统写性能有很大的损失.在数据对象比较小时,吞吐量减少了 40%左右;在数据对象比较大时,吞吐量减少了 20%左右.

我们比较在 TinyKV-flush 模式下的 TinyKV 和 LevelDB,Masstree.如图 7 所示,在均匀分布的数据集上 TinyKV 的性能比 LevelDB 和 Masstree 要好.在小数据对象的数据集测试中,TinyKV 的吞吐量是 LevelDB 的 10 倍,是 Masstree 的 2.6 倍;在大数据对象的数据集测试中,TinyKV 的吞吐量是 LevelDB 的 30 倍,是 Masstree 的 7 倍.通过大小数据对象的测试性能差异,可以看出 TinyKV 的索引结构是比较有效率的.另一方面,如图 7 所示,在偏斜分布的数据集上,当数据对象比较小时,TinyKV 的写性能要比 Masstree 要小,这是数据持久化带来的负面影响.在偏斜分布的数据集中,由于某些数据出现的次数要比大部分数据要多,所以如果能对这些数据做缓存则系统的性能会有很大的提升.这是

系统在偏斜分布数据集测试性能比均匀分布数据集优异的原因.但 TinyKV 使用刷新高速缓存的指令做数据持久化时,会与 Hash 表和数据链有关的高速缓存行失效.这样,对 TinyKV 来说,偏斜分布数据集上的数据空间局域性荡然无存.如果使用 *clwb* 替代 *clflush*,这种情况或许能得到改善.

图 8 展示了 TinyKV,Masstree 以及 LevelDB 的读性能.读性能测试不涉及数据的持久化过程,故 TinyKV 的运行模式是 TinyKV-flush.如同 Masstree 一样,TinyKV 追加一个与 key 相关联的 value 的指针.当存储系统中不存在 key 时,返回一个空指针.在不同的数据集上,TinyKV 的性能都比其他系统要好.正如系统的写测试的实验结果,这些系统在偏斜分布数据集的测试性能比均匀分布数据集的测试性能更加优异.

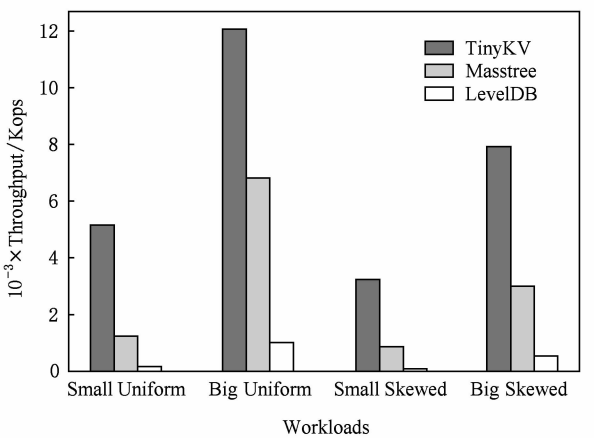

Fig. 8 Single thread read throughput

图 8 读性能测试

4.3 并发性能

在 TinyKV 系统中,数据查询不需要加锁,当不对系统的数据进行修改时,CPU 的高速缓存不会由于缓存一致性而失效,系统多线程查询性能良好.

图 9 展示了不同系统在小对象均匀分布的数据集上的多线程写性能.可以看出 TinyKV 和 Masstree 的扩展性比较好.LevelDB 的曲线比较平缓,其吞吐量基本维持在 0.02 Mops 左右;Masstree 在单线程时它的吞吐量是 0.11 Mops,当使用 10 个线程时,它的吞吐量也达到了 0.21 Mops.TinyKV 在单线程时的吞吐量是 2.1 Mops,在 4 线程时它的吞吐量达到了 5.4 Mops,在 10 线程时它的吞吐量达到了 6.7 Mops.这一方面得益于 TinyKV 每个线程使用了专有的日志及 Hash 表的桶有独立的写锁,另一方面随着线程的增多,写锁资源的竞争会激烈.

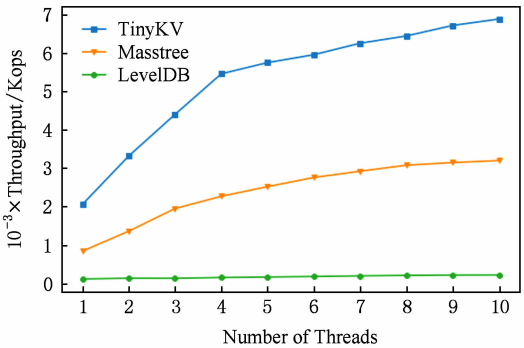

Fig. 9 Write throughput of multi-threads

图 9 多线程性能测试

现有的基于 NVM 的键值存储系统依赖于一些主流的索引数据结构,如 B 树、RB 树等,并在数据更新时保证这些数据结构的一致性.NVTree,FPTree 工作是针对 B^+ 树做了并发一致性方面的工作,在范围查询方面,B^+ 树可以表现出更高的性能.TinyKV 采用 Hash 索引方式是为了保证系统的轻量级特性:支持高吞吐率的 PUT/GET 操作,这在一定程度上牺牲了范围检索的性能.

一些键值存储系统,如 MICA,HiKV 等,同样采用了 Hash 结构的键值索引方式.但是它们与 TinyKV 的设计理念不同:MICA 是一种高速内存键值数据库,但没有结合 NVM 去保证数据持久性.因此,若要支持持久性,现有的 MICA 需依赖于传统的技术方案:写日志到磁盘或定期做检查点.这会导致以页(4 KB)为单位的数据同步产生严重的 I/O 开销.HiKV 使用一种混合索引结构,既支持 Hash 索引也支持 B^+ 树索引.这使得 HiKV 可以支持复杂的负载数据处理,例如范围检索+特定数据集合.然而,在简单的负载数据处理上,TinyKV 仍旧更有优势.这得益于 TinyKV 全局 Hash 的索引结构设计,以及多线程并发机制的支持.

由于时间原因和硬件条件的限制,例如:MICA 对服务器和客户端均做了一定程度的硬件假设,如 NUMA 结构支持、多核支持等;HiKV 暂时不开源,复现工作需要较多时间.因此,我们没有展开实验进行说明,敬请谅解.

5 总 结

本文设计并实现一种基于日志结构的非易失性内存键值存储系统 TinyKV,采用分块技术对非易失性内存进行划分,以块为单位对存储空间进行管

理;使用日志结构的数据修改方式,既满足数据一致性的存储要求又能提高存储空间的利用率;通过设计多日志的存储结构,减少多线程的竞争,提高系统扩展能力;设计多级的内存分配器,在底层进行块管理,中间层进行日志管理,上层进行 DRAM 数据对象空间分配;实现了一个高效的 Hash 表作为存储系统的索引结构,它通过估算 Hash 表的大小分配比较大的初始空间,从而减少 Hash 表的冲突概率并减少 Hash 表的扩容次数;使用动态数组作为解决 Hash 冲突的存储方式,对 Hash 表实现了允许多读单写的读写方式;在垃圾回收算法中,通过扫描块状态表在 DRAM 中构建运行时需要的数据信息,避免把垃圾回收算法产生的数据存储到非易失性内存中.实验测试结果证明,TinyKV 具有良好的吞吐性能和扩展能力,在使用均匀分布的小对象数据集进行多线程写性能测试时,TinyKV 单线程的吞吐量为 2.1 Mops,4 线程的吞吐量为 5.4 Mops,TinyKV 的吞吐量是 LevelDB 的 10 倍,是 Masstree 的 2.6 倍;在大对象数据集测试中,TinyKV 的吞吐量是 LevelDB 的 30 倍,是 Masstree 的 7 倍.

参 考 文 献

[1] Zhang Hongbin, Fan Jie, Shu Jiwu, et al. Summary of storage system and technology based on phase change memory [J]. Journal of Computer Research and Development, 2014, 51(8): 1647-1662 (in Chinese)
(张鸿斌, 范捷, 舒继武, 等. 基于相变存储器的存储系统与技术综述[J]. 计算机研究与发展, 2014, 51(8): 1647-1662)

[2] Shu Jiwu, Lu Youyou, Zhang Jiacheng, et al. Research progress on non-volatile memory based storage system [J]. Science & Technology Review, 2016, 34(14): 86-94 (in Chinese)
(舒继武, 陆游游, 张佳程, 等. 基于非易失性存储器的存储系统技术研究进展[J]. 科技导报, 2016, 34(14): 86-94)

[3] Shen Zhirong, Xue Wei, Shu Jiwu. Research on new non' volatile storage [J]. Journal of Computer Research and Development, 2014, 51(2): 445-453 (in Chinese)
(沈志荣, 薛巍, 舒继武. 新型非易失存储研究[J]. 计算机研究与发展, 2014, 51(2): 445-453)

[4] Ghemawat S, Dean J. Level DB, v1.14.0 [OL]. [2015-03-20]. https://github.com/google/leveldb,http://leveldb.org

[5] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store [C] //Proc of the 21st ACM SIGOPS Symp on Operating Systems Principles. New York: ACM, 2007: 205-220

[6] Yan Cairong, Li Tie, Huang Yongfeng, et al. HMFS: Efficient support of small files processing over HDFS [C] //Proc of the 14th Int Conf on Algorithms and Architectures for Parallel Processing. Berlin: Springer, 2014: 54-67

[7] Zheng Shenan, Huang Linpeng, Liu Hao, et al. HMVFS: A hybrid memory versioning filesystem [C] //Proc of the 32nd Symp on Mass Storage Systems and Technologies. Piscataway, NJ: IEEE, 2016

[8] Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory [C] //Proc of the 9th European Conf on Computer Systems. New York: ACM, 2014: 15

[9] Xu Jian, Steven 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

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

[11] Fitzpatrick B. Memcached: A distributed memory object caching system [OL]. [2018-06-06]. https://memcached.org/

[12] Lim H, Han D, Andersen D G, et al. MICA: A holistic approach to fast in-memory key-value storage [C] //Proc of the 11th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2014: 429-444

[13] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385

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

[15] Bailey K A, Hornyack P, Ceze L, et al. Exploring storage class memory with key value stores [C] //Proc of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads. New York: ACM, 2013

[16] Chen Xianzhang, Sha E H M, Abdullah A, et al. UDORN: A design framework of persistent in-memory key-value database for NVM [C] //Proc of the 6th Non-Volatile Memory Systems and Applications Symp. Piscataway, NJ: IEEE, 2017

[17] Marmol L, Sundararaman S, Talagala N, et al. NVMKV: A scalable and lightweight flash aware key-value store [C] //Proc of the 6th USENIX Workshop on Hot Topics in Storage and File Systems. Berkeley, CA: USENIX Association, 2014

[18] Sengupta S, Debnath B K, Li Jin. Flash memory cache including for use with persistent key-value store: USA Patent 9,298,604 [P]. 2016-03-29

[19] Wu Xingbo, Ni Fan, Zhang Li, et al. NVMcached: An nvm-based key-value cache [C] //Proc of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems. New York: ACM, 2016

[20] Xia Fei, Jiang Dejun, Xiong Jin, et al. HiKV: A hybrid index key-value store for DRAM-NVM memory systems [C] //Proc of 2017 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2017: 349-362

[21] Liu Hao, Huang Linpeng, Zhu Yanmin, et al. LibreKV: A persistent in-memory key-value store [OL]. [2018-07-26]. https://ieeexplore. ieee. org/document/8239872/

[22] Zhou Jie, Shen Yanyan, Li Sumin, et al. NVHT: An efficient key-value storage library for non-volatile memory [C] //Proc of the 3rd IEEE/ACM Int Conf on Big Data Computing, Applications and Technologies. New York: ACM, 2016: 227-236

[23] Huang Kaixin, Zhou Jie, Huang Linpeng, et al. NVHT: An efficient key-value storage library for non-volatile memory [OL]. [2018-06-07]. https://doi. org/10. 1016/j. jpdc. 2018. 02. 013

[24] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight persistent memory [J]. ACM SIGARCH Computer Architecture News, 2011, 39(1): 91-104

[25] Coburn J, Caulfield A M, Akel A, et al. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories [J]. ACM Sigplan Notices, 2011, 46 (3): 105-118

[26] Carns P H, Jenkins J, Cranor C D, et al. Enabling NVM for data/intensive scientific services [OL]. [2018-07-26]. https://www. usenix. org/system/files/conference/inflow16/inflow16/paper/carns. pdf

[27] Sha E H M, Chen Xianzhang, Zhuge Qingfeng, et al. A new design of in-memory file system based on file virtual address framework [J]. IEEE Trans on Computers, 2016, 65(10): 2959-2972

[28] Li Bojie, Ruan Zhenyuan, Xiao Wencong, et al. KV-Direct: High-performance in-memory key-value store with programmable NIC [C] //Proc of the 26th Symp on Operating Systems Principles. New York: ACM, 2017: 137-152

[29] Atikoglu B, Xu Yuehai, Frachtenberg E, et al. Workload analysis of a large-scale key-value store [J]. ACM SIGMETRICS Performance Evaluation Review. 2012, 40 (1): 53-64

[30] Sun Yuanyuan, Hua Yu, Feng Dan, et al. A collision-mitigation cuckoo hashing scheme for large-scale storage systems [J]. IEEE Trans on Parallel and Distributed Systems, 2017, 28(3): 619-632

[31] Rosenblum M, Ousterhout J K. The design and implementation of a log structured file system [J]. ACM Trans on Computer Systems, 1992, 10(1): 26-52

[32] Mao Yandong, Kohler E, Morris R T. Cache craftiness for fast multicore key-value storage [C] //Proc of the 7th ACM European Conf on Computer Systems. New York: ACM, 2012: 183-196

[33] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB [C] //Proc of the 1st ACM Symp on Cloud Computing. New York: ACM, 2010: 143-154

You Litong, born in 1994. PhD candidate. His main research interests include non-volatile memories, file system, and key value store system.

Wang Zhenjie, born in 1992. Master candidate. His main research interests include non-volatile memories and key value store system.

Huang Linpeng, born in 1964. PhD, professor and PhD supervisor. His main research interests in clude distributed systems and in-memory computing.