面向非易失性内存的持久索引数据结构研究综述

王永锋，陈志广
中山大学计算机学院，广东 广州 510006

摘要
随着非易失性内存从理论走向实用，现代存储系统的设计与实现将迎来颠覆性变革。针对传统存储设备设计的存储系统并不能充分利用非易失性内存带来的性能红利。为了构建高吞吐、低时延、大规模的存储系统，迫切需要设计与非易失性内存硬件特性相匹配的持久索引数据结构，从而进一步提升性能。从持久索引数据结构出发，分别对B+-Tree和哈希表在非易失性内存上的设计和优化进行分析，比较其优缺点，并展望了该方向的机遇与面临的挑战。

关键词
存储系统；非易失性内存；持久索引数据结构；性能优化

中图分类号：TP311 文献标识码：A doi: 10.11959/j.issn.2096-0271.2021062

A survey of persistent index data structures on non-volatile memory

WANG Yongfeng, CHEN Zhiguang
School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou 510006, China

Abstract
With non-volatile memory becoming commercially available, the design and implementation of traditional storage systems need a fundamental change since they can not fully utilize the performance of non-volatile memory. To build a high-throughput, low-latency, large-scale storage system, there is an urgent need for efficient persistent index data structures that adapt to the characteristics of non-volatile memory. In terms of persistent index data structures, the optimizations applied for B+-Tree and Hash Table on non-volatile memory were summarized, and the pros and cons among these schemes were compared. And the future research directions with the challenges and opportunities that need to be resolved were showed.

Key words
storage system, non-volatile memory, persistent index data structure, performance optimization

1 引言

非易失性内存是一种新兴的存储介质，其具备字节可寻址、内存级别读写时延的特性，这给当前大量的存储系统带来了根本性的变革。非易失性内存正在迅速发展，现有的大部分非易失性内存（如相变内存[1]、STT-RAM[2]等）仍处于研究阶段，但由美光科技有限公司和英特尔联合研制的傲腾持久内存（基于3D XPoint[3]）已经发布，并且投入市场。由此，在存储系统中尽可能发挥非易失性内存性能优势的需求越发迫切。其中，面向非易失内存研发新型持久索引数据结构是解决该问题的关键。

在存储系统的设计与实现中，持久索引数据结构是核心之一。文件系统中文件路径到索引节点的寻址、大文件中偏移量到指定数据块的寻址、键值存储系统中根据键寻找值的数据结构、数据库中的聚集索引等，都需要持久存储的索引数据结构，且这些持久索引数据结构的实现对系统本身的性能至关重要。但这些持久索引数据结构目前大多面向传统存储设备进行优化，而不能高效利用非易失性内存的硬件特性。将这些持久索引数据结构在非易失性内存上重新设计实现，并面向非易失性内存的硬件特性进行优化，能够大大降低存储系统的时延、提升吞吐量。

近年来，已经有一些工作对非易失性内存上的存储系统进行了深入的探讨和研究。陈游旻等人[4]在大数据环境下分析了基于新型存储硬件进行存储系统构建的挑战和趋势；Liu H K等人[5]对非易失性内存下的内存架构设计、持久内存管理、性能优化和功耗管理、编程框架以及非易失性内存之上的应用进行了全面的综述；Lersch L等人[6]将基于模拟非易失性内存设计的有序索引数据结构进行分析，并在英特尔傲腾持久内存上进行了测试；Hu D K等人[7]则针对面向非易失性内存构建的哈希表进行了测试和分析。另外，邓镇龙等人[8]将非易失内存与MPI-IO相结合，大大提升了应用在高性能集群上的读写性能；杨青霖等人[9]提出的高效数据缓存方法显著提升了存储系统的读写性能。这些工作对在非易失性内存上设计存储系统具有重要的指导意义。本文从持久索引数据结构的设计出发，对不同的设计之间的差异进行细粒度的对比。

提高持久索引数据结构的性能是实现低时延、高吞吐的现代存储系统亟须解决的问题。针对大量在非易失性内存上优化持久索引数据结构的工作，笔者对其进行分类、汇总、对比，厘清索引数据结构的发展主线，总结其中的关键挑战，并对其发展趋势进行展望。

2 持久索引数据结构

索引数据结构是一种能够存储键值对映射的数据结构。常用的数组仅能实现高效的查询，无法实现高效的插入操作，而设计良好的索引数据结构对于查询和插入操作，都能达到对数时间复杂度，甚至是常数时间复杂度，这样的性能优势使其在内存系统和外存系统中均得到广泛应用。索引数据结构面向的场景种类繁多，针对不同的场景，索引数据结构有多种不同的实现，本文关注需要持久存储在存储设备上的索引数据结构，这样的索引数据结构被称为持久索引数据结构（以下简称索引数据结构）。

2.1 索引数据结构的分类

在不同的场景中，对索引数据结构中存储的数据有不同的假定。构建在关系型数据库中的索引数据结构往往需要处理大量的范围查询，即查询在某个区间内的所有键值对。为了高效支持范围查询，索引数据结构需要维护数据的有序性，并针对范围查询进行优化。而在一些键值存储系统中，可能仅输入指定的键，要求系统返回对应的值，不需要范围查询，此时底层的索引数据结构就不需要额外的开销来维护数据的有序性，相邻的键值对可以存放在存储设备上的任意位置。因此，根据其内部数据结构对数据有序性的维护情况，可以将索引数据结构分为有序索引数据结构和无序索引数据结构。

有序索引数据结构需要严格维护数据结构中的有序性。对于每一次写入操作，有序索引数据结构都需要根据插入的数据对整体的结构进行修改以保证有序性，因此范围查询的性能最好。哈希表完全不维护数据结构中的有序性，因此一般而言额外开销最小，但无法对范围查询进行优化。另外，在有序索引数据结构中，维护有序性会带来大量开销，因此一些面向写优化的有序索引数据结构会放松一部分对有序性的约束，从而提升写性能。

除了传统索引，新兴的学习索引（learning index）将索引任务变为一个回归问题，能够根据输入数据自适应地调整数据的存放模式。

2.2 面向非易失性内存的索引数据结构关键问题

为了能够让索引数据结构在传统的存储设备（如机械硬盘、固态硬盘）中进行持久存储，并且高效地利用硬件特性，很多研究人员进行了大量的研究工作。随着新型存储设备（如非易失性内存）渐渐成熟，研究人员分析了非易失性内存和传统存储设备的差异，并且就索引数据结构的设计提出了不少新颖的方法。具体地说，在非易失性内存上实现持久索引数据结构，需要解决以下3个问题。

- 如何减少在操作持久索引数据结构时的软件开销？
- 如何针对特定的持久化语义实现崩溃一致性保证？
- 在非易失性内存上如何利用多核架构高效并发处理读写请求？

索引数据结构的软件开销逐渐成为限制性能的关键因素。传统存储设备的持久化时延往往是微秒级甚至毫秒级，而内存的时延最多不过一百多纳秒，因此在传统存储设备上的持久索引数据结构并不需要过多地关注与内存读写相关的软件开销，而是更多地关注如何通过写聚合等方式尽可能减小持久化的开销。但在非易失性内存的背景之下，持久化开销与内存的读写相近，如傲腾内存的写时延为62 ns，而读时延为169~305 ns[10]，过去的一些略微提高软件开销、降低持久化开销的优化手段无法被直接应用在非易失性内存中，软件开销对性能的影响大大增加[11]。同时，缓存未命中、流水线停顿等体系架构层面的性能损失也会对构建在非易失性内存之上的索引数据结构有较大的影响。如为了进一步降低开销，MOD[12]将持久化所需要的内存屏障进一步降低，以提升性能。另外，为了降低开销，还需要尽可能解决读写放大的问题，有工作指出[10]，由于傲腾内存的内部读写粒度为256 byte，小于256 byte的读写均可能带来写放大，这会对索引数据结构的性能有所影响。

同时，需要重新思考崩溃一致性的实现。传统存储设备基于块设备的抽象、操作和读写都以块为单位（更具体地说，机械硬盘的读写粒度为扇区，大小一般是512 byte，固态硬盘的读写粒度为闪存页，大小一般是4 KB），只要相应的块或页被写入存储设备，即完成了持久化。但在非易失性内存中，一般的store指令原子操作粒度仅为8 byte，且该指令会由于CPU的乱序执行而难以按照开发者预想的顺序写入存储。另外，数据会首先写入CPU的L1/L2/L3缓存中，而CPU的缓存并不能保证持久化。因此，为了保证让数据在非易失性内存上持久化存储，需要在store指令后相应地加入内存屏障和刷写缓存行指令（如clflush或clwb），将缓存行从CPU缓存刷到非易失性内存里。由此，非易失性内存上需要新的方法来保证崩溃一致性。

另外，面对海量的读写请求，需要设计适合多核架构的索引数据结构。由于极低的时延以及字节可寻址的特性，与传统存储设备相比，构建在非易失性内存之上的索引数据结构的吞吐量有多个数量级的优势，在多核架构上非易失性内存的优势将更加显著。尽管如此，由于前面提到的两个问题，能够在动态随机存取存储器（dynamic random access memory，DRAM）上使用的并行索引数据结构并不能直接用于非易失性内存上。另外，由于非易失性内存额外引入的刷写缓存行操作，以及傲腾内存在线程数过多时带宽反而会下降[10]，需要对索引数据结构上的并发读写做进一步的优化，才能够充分适应非易失性内存的特性。

针对上述3个问题，目前已有很多研究给出了针对非易失性内存的索引数据结构设计。本文主要分析其中两种常见的索引数据结构（B/B+树（B/B+-Tree）和哈希表），并总结了它们在非易失性内存上的发展。

3 有序索引数据结构在非易失性内存上的实现

有序索引数据结构能够高效地处理范围查询任务，其中的一种实现——B/B+-Tree能够显著地减少磁盘I/O的次数，已经被广泛应用到InnoDB等存储引擎中。针对B/B+-Tree在非易失性内存上的实现，笔者总结了下面的工作，并分析了其优劣。其中2011—2018年的工作都是在模拟的非易失内存模拟器上完成的，之后的工作才开始在真实的傲腾内存上实现。

2011年Venkataraman S等人[13]首次提出了针对非易失性内存设计的CDDS B-Tree。他们使用mfence（内存屏障）和clflush（刷写缓存行）的组合指令来保证数据按顺序写入非易失性内存中，在B-Tree的基础上，使用多版本机制实现更新操作，另外通过写时复制实现节点的分裂和合并，从而减少了额外的写入，无须通过写日志保证崩溃一致性。但是使用多版本和写时复制的代价是需要后台线程来进行垃圾回收，这会带来额外的性能开销。

2015年Yang J等人[14]对非易失性内存上的B+-Tree进行了改进，提出了能够进一步降低维护一致性开销的NV-Tree。通过深入分析，他们发现在叶子节点维护顺序存放的键值需要刷写多个缓存行，另外还需要维护B+-Tree的内部节点的崩溃一致性，这些引入了大量开销。为了进一步优化，该文章提出可以让叶子节点存放的键值对乱序，具体的实现是使用日志结构写入。另外，由于内部节点可以重建，不需要额外维护内部节点的崩溃一致性。但由于叶子节点没有维护顺序，这种方法对读操作的性能造成了一定的影响。

2015年Chen S M等人[15]基于参考文献[14]进一步优化了B+-Tree在非易失性内存上的实现，提出了wB+-Tree。如果叶子节点没有维护键值对的顺序，就会影响读操作的性能，因此该文章在叶子节点中使用位图记录槽的分配情况，并进一步增加槽数组（slot array）用于记录键值的顺序，优化读操作。另外针对崩溃一致性的实现，wB+-Tree在插入操作和更新操作中，都先在节点中寻找空的或无用的槽写入并保证持久化，然后通过一次8 byte的原子写入和持久化修改元数据，从而完成操作。这样的实现使用非易失性内存上的8 byte原子写入指令保证崩溃一致性，但节点分裂操作依然使用了传统的重做日志方法，带来了额外的写入。

2016年Oukid I等人[16]结合参考文献[14]提到的分析，在非易失性内存上进一步优化了B+-Tree，提出了FPTree。基于内部节点可以通过叶子节点重建的原理，FPTree将所有内部节点都放在DRAM里，只将叶子节点持久化存放在非易失性内存中，减小维护崩溃一致性的开销。同时FPTree在叶子节点中存放了每个键各1 byte的指纹，用于快速判断指定键是否在该叶子节点中，从而降低叶子节点无序存放键值对读操作的影响。为了进一步降低软件开销，优化并发，FPTree将分配内存的开销分摊到多个节点上，并结合硬件事务内存降低了并发访问的开销。

2018年Arulraj J等人[17]基于PMwCAS在非易失性内存上实现了能够无锁并发的BzTree。比较并交换（compare and swap，CAS）指令是实现无锁并发算法的关键指令，其能够对单个字节进行原子的比较和交换操作，而PMwCAS[18]将该操作扩展到多个字节且保证非易失性内存上的持久化。通过PMwCAS提供的原子性，开发者可以避免非易失性内存带来的编程细节，使用通用的方法在非易失性内存上实现支持崩溃一致性且无锁并发的BzTree。

2018年Hwang D等人[19]在非易失性内存上基于FAST（failure-atomic shift）&FAIR（failure atomic in-place rebalancing）算法实现了可容忍瞬时不一致的B+-Tree。由于一些不一致的情况可以通过修改读操作来容忍，避免读取错误的内容，作者首先通过FAST算法保证在节点内部维护有序性时，移动键值对产生的不一致是可容忍的，然后FAIR算法能够以类似的保证进一步处理节点分裂或合并的情况。这个方法无须任何日志就能够保证B+-Tree在任何操作中都处于不一致但可容忍的状态，同时由于内部节点也存在于非易失性内存中，基于该方法实现的B+-Tree崩溃后能够马上恢复，而无须重建索引。

2020年Chen Y M等人[20]发现非易失性内存上的B+-Tree有较严重的长尾时延，经过深入分析后，他们认为在非易失性内存中对叶子节点的结构进行改变的操作（排序和节点平衡）以及并发线程之间相互等待访问非易失性内存是造成长尾时延的根本原因。基于这样的分析，他们提出uTree。uTree的内部节点组织与一般B+-Tree相同，存放在内存上，而叶子节点分成内存中的数组层与存放在非易失性内存上的链表层。该设计能让造成长尾时延的操作在内存上完成，而在非易失性内存上可以并行操作链表的个别元素。这样的设计缓解了长尾时延，但也增加了索引数据结构的内存占用。

2020年Liu J H等人[21]在3D XPoint上优化了B+-Tree，提出了LB+Trees。他们充分利用了3D XPoint上内部介质读写粒度为256 byte和持久化粒度为64 byte之间的差异，发现影响性能的是CacheLine

的写入，在CacheLine写入数量相同的情况下，CacheLIne内部脏字写入是没有影响的，进而提出可以通过节点内部键值对的移动来减少内部介质读写。同时为了保证崩溃一致性，他们在wB+-Tree的实现上进一步扩展，基于类似的思路，利用8 byte的原子写保证了包括节点分裂和聚合在内的所有操作的崩溃一致性，无须写日志，并且还能够通过分布式头元数据的方式扩大叶子节点的大小（256 byte的倍数）而不牺牲崩溃一致性。

B/B+-Tree在非易失内存上的实现及优缺点见**表1**。除了B/B+-Tree之外，在非易失性内存上进行优化的有序索引数据结构还包括基数树（radix tree）以及其变体，如WORT[22]、P-ART[23]、HART[24]、DPTree[25]、ROART[26]等。这些工作从不同方面对上述3个关键问题给出了不同的解决方案。从这些工作可以发现，为了高效地利用非易失性内存的性能，减小软件开销、通过避免日志写入实现崩溃一致性、通过避免锁的使用来实现高效的并发、结合处理器体系架构和非易失性内存的硬件特性来进行优化，已经成为主流的方法。

4 哈希表在非易失性内存上的实现

随着NoSQL存储系统逐渐发展，无须维护数据有序性从而具备更小开销和更简洁实现的哈希表目前在各种存储系统中得到越来越广泛的应用，如MongoDB、HBase、Memcached等。在哈希表的设计中，需要解决的两个核心问题分别是哈希表扩容和哈希冲突。哈希表扩容指的是当哈希表的容量不足以容纳用户需要写入的所有数据时，哈希表需要进行容量扩大，并根据需要将旧的数据重哈希到新的哈希表上。哈希冲突指的是哈希表中的单个桶（bucket）没有空闲位置放置键值对的情况。

现有的针对哈希表的研究可分为两类：动态哈希和静态哈希。在静态哈希中，当哈希表容量需要进行扩大时，一般需要创建一个更大的哈希表，并将旧哈希表中的数据重新插入新的哈希表中，其扩容的开销与哈希表本身包含的数据量成正比。关于这种哈希表在非易失性内存上的实

表1 B+-Tree 在非易失性内存上的实现与优缺点

对比项	研究工作	主要贡献	缺点
非易失内存模拟器上的B/B+-Tree	CDDS B-Tree[13]	提出使用clflush和mfence保证持久性	需要后台线程进行垃圾回收
	NV-Tree[14]	提出了选择性持久化，仅维护核心数据的崩溃一致性，另外放松叶子节点的顺序约束，允许无序存放	叶子节点的无序性会影响范围查询的性能
	wB+-Tree[15]	使用8 byte原子写实现崩溃一致性，无须日志	节点分裂需要写入重做日志，带来额外写入开销
	FPTree[16]	将可重建的内部节点放到内存中，并使用硬件事务内存降低并发事务的开销	由于内存中的数据会丢失，需要较长时间进行崩溃后重建
	BzTree[17]	使用PMwCAS实现B-Tree，降低了无锁编程的难度	PMwCAS操作带来较多的性能开销
	FAST&FAIR[19]	无须任何日志，基于可容忍的瞬间不一致实现崩溃一致性，无须额外写入	需要依赖特定的内存模型和硬件特性
傲腾内存上的B/B+-Tree	uTree[20]	通过存放于内存的叶子节点，避免在非易失性内存上进行操作导致的长尾时延	需要占用较多的内存
	LB+Trees[21]	减少CacheLine写，无须日志保证崩溃一致性	需要硬件特性（硬件事务内存）实现事务并发

现，目前的研究成果如下。

Zuo P F等人[27]提出的Path-Hashing针对非易失性内存环境下的哈希表，指出以往的哈希表设计会带来大量的额外写，这会极大地影响哈希表在非易失性内存上的性能。因此他们提出了一个基于路径的哈希表。该哈希表通过位置共享技术解决哈希冲突问题，优化了对CPU缓存的使用且不会带来额外的写。为了进一步提高负载因子和读写性能，Path-Hashing还能够使用两个哈希函数构建两条搜索路径，并进行路径压缩，在缓解哈希冲突的同时，减小路径的深度。

Zuo P F等人[28]进一步优化了哈希表的写入操作，提出了Level-Hashing。他们不仅提出了一种写优化的层次哈希结构，通过双哈希函数和冲突后一次键值对的移动来提升哈希表的负载因子，同时还实现了一种原地扩容机制，这让Level-Hashing只需要对1/3的内容进行重哈希，将处于底层的内容重新写到顶层的哈希表中，即可完成扩容。另外Level-Hashing还能够以无须日志的方式保证插入、修改、删除、扩容操作的一致性，只有更新操作可能需要写日志。

Chen Z Y等人[29]提出的CLevel-Hashing对哈希表的并发性能进行了大量改进。一方面，CLevel-Hashing能够在后台异步扩容和重哈希，而不会阻塞操作，从而避免了等待扩容导致的长尾时延；另一方面，过去哈希表一般使用锁来实现并发，他们针对读取、插入、更新、删除操作均基于CAS原语实现了无锁并发算法，从而避免锁竞争，提升哈希表的扩展性。

相对于静态哈希，动态哈希能够根据用户插入的键值对的数量灵活地扩大或缩小哈希表的容量，每次扩容时只需要常数时间复杂度的开销。

Nam M等人[30]在非易失性内存中引入了可扩展哈希CCEH（CacheLine-conscious extendible hashing）。基于原始的可扩展哈希设计，他们根据非易失性内存的特性进行了一些改进，包括将桶的大小设置为缓存行的整数倍，以及增加一层段（segment）来减少空间占用。同时他们还提出了一种崩溃恢复算法，使其能够保证在修改CCEH和进行段分裂时无须任何日志即可保证崩溃一致性。

Lu B T等人[31]进一步提出了一种扩展性更好的哈希表Dash。他们认为不仅要优化非易失性内存上的写入，还要尽量减少非易失性内存上的读取操作。因此他们对Dash做了两方面的优化。从哈希表本身的设计上，他们提出基于指纹对键的存在与否进行快速的判断以优化读取速度，同时在哈希冲突的解决上，他们使用哈希桶间自平衡的方法来提高哈希表的负载因子。同时Dash还能够让插入和更新操作均不需要用日志来保证崩溃一致性，仅在段分裂时需要使用重做日志。另外，Dash还使用了优化的并发控制访问，读取不需要锁的参与，写时使用原子变量实现互斥访问。

Zou X M等人[32]提出了HMEH（hybrid memory extendible hashing）。他们在CCEH的基础上，考虑到可扩展哈希的目录层可通过非易失性内存上额外构建的一颗基数树来重建，因此将目录层放到DRAM中，以减少非易失性内存上的读写。另外他们还提出了一种利用8 byte原子写实现的键值交错策略，使HMEH无须任何日志也无须显式地刷写缓存行就能够保证一般操作的崩溃一致性。对于段分裂操作，基于之前设置的基数树，HMEH实现了一种无须任何日志和写时复制的分裂方法。

哈希表在非易失性内存上的实现与优缺点见**表2**，可以发现，其发展趋势与有序索引数据结构是一致的，在充分利用

硬件特性进行优化的同时，无须日志的崩溃一致性保证和无锁算法成为在非易失性内存上高效实现索引数据结构的关键所在。

5 结束语

本文分别对B+树和哈希表在非易失性内存上的设计和实现进行了深入的分析。根据两者的发展发现，在非易失性内存上实现持久索引数据结构所需要解决的3个问题目前已有一致的趋势。为了减小索引数据结构的软件开销，现有研究会放松对数据有序性的维护，减少内存屏障与缓存行的刷写，感知内部存储介质的读写粒度以及尽可能减少额外写，而对于崩溃一致性与适应多核架构，则分别趋近于无须日志和无须加锁的方法。

最后，对在非易失性内存下高效实现索引数据结构的过程中存在的一些挑战进行总结。

一是如何在保留崩溃一致性的同时利用DRAM进一步优化索引数据结构的性能。目前的非易失性内存尽管性能接近DRAM，但是其时延仍然比DRAM高好几倍，且带宽更加受限。由于非易失性内存和DRAM之间的性能差距仍然不可忽略，现有的研究工作已经在尝试使用DRAM来优化索引数据结构的性能，但也有研究表明，在一些场景中，DRAM和非易失性内存混用可能会由于额外的数据迁移造成性能损失，因此如何高效地利用DRAM来优化索引数据结构仍然是一个需要深入研究的问题。

二是如何更进一步地利用非易失性内存的硬件特性对索引数据结构进行优化。非易失性内存与DRAM类似，但又有其独特的性质，之前的研究大多在使用DRAM模拟的非易失性内存上完成，近两年的研究工作大多基于英特尔傲腾持久内存，指出过去的研究中存在的问题，并对傲腾内存的硬件特性进行了一些适配。但一方面英特尔并没有公布傲腾持久内存的内部原理和架构，研究人员只能通过猜测其硬件特性进行优化，另一方面还有更多的新型非易失性内存存储介质未面市，这些存储介质可能具备不一样的特性，在这样的背景下，持久索引数据结构需要具有更灵活的设计才能适应存储介质的发展。

三是如何实现高效的非易失性内存空间分配器。在索引数据结构的设计和实现中，非易失性内存空间的动态分配是一个不可或缺的操作，其作为软件开销的一部

表 2 哈希表在非易失性内存上的实现与优缺点

对比项	研究工作	优点	缺点
静态哈希	Path-Hashing[27]	独特的哈希表结构大大减少了额外写	不能保证崩溃一致性，没有实现并发
	Level-Hashing[28]	哈希表扩容的复杂度为$O(1/3N)$，对一般的静态哈希扩容实现了常数级别的优化	在更新键值对和进行扩容时需要写日志，有额外的开销
	CLevel-Hashing[29]	完全使用无锁算法实现，同时无须任何日志	存在脏读现象，无法支持更高级别的隔离性
动态哈希	CCEH[30]	实现了无须日志即可保证崩溃一致性的可扩展哈希	没有针对非易失内存做更多特定的优化
	Dash[31]	使用更轻量的原子变量和版本号实现优化的并发，并使用指纹减少额外的读取	段分裂时仍然需要通过日志保证崩溃一致性
	HMEH[32]	将目录层放到DRAM，优化读写性能，同时无须日志即可保证崩溃一致性	仍然需要锁来实现并发

分，对索引数据结构的性能有较大影响。同时一个高效的非易失性内存分配器要求避免永久性的内存泄露，并支持高效的并发操作。然而现有的研究中大多仅关注数据结构本身，忽略了这部分重要的软件开销，另外在DRAM中的内存分配器也需要更进一步的改进才能充分利用非易失性内存的特性。

参考文献:

[1] RAOUX S, BURR G, BREITWISCH M J, et al. Phase-change random access memory: a scalable technology[J]. IBM Journal of Research and Development, 2008, 52(4): 465-479.

[2] KAWAHARA T. Scalable spin-transfer torque RAM technology for normally-off computing[J]. IEEE Design & Test of Computers, 2011, 28(1): 52-63.

[3] DÉVELOPPEMENT Y. Intel and Micron produce breakthrough memory technology[R]. 2021.

[4] 陈游旻, 李飞, 舒继武. 大数据环境下的存储系统构建：挑战、方法和趋势[J]. 大数据, 2019, 5(4): 27-40.
CHEN Y M, LI F, SHU J W. Building storage systems in big data era: challenges, methods and trends[J]. Big Data Research, 2019, 5(4): 27-40.

[5] LIU H K, CHEN D, JIN H, et al. A survey of non-volatile main memory technologies: state-of-the-arts, practices, and future directions[J]. Journal of Computer Science And Technology, 2021, 36(1): 4-32.

[6] LERSCH L, HAO X, OUKID I, et al. Evaluating persistent memory range indexes[J]. Proceedings of the VLDB Endowment, 2019, 13(4): 574-587.

[7] HU D K, CHEN Z W, WU J B, et al. Persistent memory Hash indexes: an experimental evaluation[J]. Proceedings of the VLDB Endowment, 2021, 14(5): 785-798.

[8] 邓镇龙, 陈志广. 面向非易失内存的MPI-IO接口优化[J]. 大数据, 2021, 7(2): 172-181.
DENG Z L, CHEN Z G. An optimization of MPI-IO interface for non-volatile memory[J]. Big Data Research, 2021, 7(2): 172-181.

[9] 杨青霖, 吴桂勇, 张广艳. 分布式存储系统中的数据高效缓存方法[J]. 大数据, 2021, 7(2): 147-157.
YANG Q L, WU G Y, ZHANG G Y. An approach to buffering data efficiently in distributed storage systems[J]. Big Data Research, 2021, 7(2): 147-157.

[10] YANG J, KIM J, HOSEINZADEH M, et al. An empirical guide to the behavior and use of scalable persistent memory[J]. arXiv preprint, 2020, arXiv:1908.03583v1.

[11] KADEKODI R, LEE S K, KASHYAP S, et al. SplitFS: reducing software overhead in file systems for persistent memory[C]// Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York: ACM Press, 2019: 494-508.

[12] HARIA S, HILL M D, SWIFT M M. MOD: minimally ordered durable datastructures for persistent memory[C]//Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2020: 775-788.

[13] VENKATARAMAN S, TOLIA N, RANGANATHAN P, et al. Consistent and durable data structures for non-volatile byte-addressable memory[C]//Proceedings of the 9th USENIX Conference on File and Storage Technologies. Carlsbad: USENIX Association, 2011.

[14] YANG J, WEI Q S, CHEN C, et al. NV-Tree: reducing consistency cost for NVM-based single level systems[C]// Proceedings of the 13th USENIX Conference on File and Storage Technologies. Carlsbad: USENIX Association, 2015: 167-181.

[15] CHEN S M, JIN Q. Persistent B+-trees in non-volatile main memory[J]. Proceedings of the VLDB Endowment, 2015, 8(7): 786-797.

[16] OUKID I, LASPERAS J, NICA A, et al. FPTree: a hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory[C]//Proceedings of the 2016 International Conference on Management of Data. New York: ACM Press, 2016: 371-386.

[17] ARULRAJ J, LEVANDOSKI J, MINHAS U F, et al. BzTree: a high-performance latch-free range index for non-volatile memory[J]. Proceedings of the VLDB Endowment, 2018, 11(5): 553-565.

[18] WANG T, LEVANDOSKI J, LARSON P. Easy lock-free indexing in non-volatile memory[C]//Proceedings of the 2018 IEEE 34th International Conference on Data Engineering. Piscataway: IEEE Press, 2018: 461-472.

[19] HWANG D, KIM W H, WON Y, et al. Endurable transient inconsistency in byte-addressable persistent B+-tree[C]//Proceedings of the 16th USENIX Conference on File and Storage Technologies. Carlsbad: USENIX Association, 2018: 187-200.

[20] CHEN Y M, LU Y Y, FANG K D, et al. uTree: a persistent B+-tree with low tail latency[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 2634-2648.

[21] LIU J H, CHEN S M, WANG L J. LB+Trees: optimizing persistent index performance on 3DXPoint memory[J]. Proceedings of the VLDB Endowment, 2020, 13(7): 1078-1090.

[22] LEE S K, LIM K H, SONG H, et al. WORT: write optimal radix tree for persistent memory storage systems[C]//Proceedings of the 15th USENIX Conference on File and Storage Technologies. Carlsbad: USENIX Association, 2017: 257-270.

[23] LEE S K, MOHAN J, KASHYAP S, et al. Recipe: converting concurrent DRAM indexes to persistent-memory indexes[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York: ACM Press, 2019: 462-477.

[24] PAN W, XIE T, SONG X J. HART: a concurrent hash-assisted radix tree for DRAM-PM hybrid memory systems[C]//Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium. Piscataway: IEEE Press, 2019: 921-931.

[25] ZHOU X J, SHOU L D, CHEN K, et al. DPTree: differential indexing for persistent memory[J]. Proceedings of the VLDB Endowment, 2019, 13(4): 421-434.

[26] MA S N, CHEN K, CHEN S M, et al. ROART: range-query optimized persistent ART[C]//Proceedings of the 19th USENIX Conference on File and Storage Technologies. Carlsbad: USENIX Association, 2021: 1-16.

[27] ZUO P F, HUA Y. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(5): 985-998.

[28] ZUO P F, HUA Y, WU J. Write-optimized and high-performance hashing index scheme for persistent memory[C]//Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018: 461-476.

[29] CHEN Z Y, HUA Y, DING B, et al. Lock-free concurrent level hashing for persistent memory[C]//Proceedings of the 2020 International Conference on Advanced Technologies for Communications. [S.l.:s.n.], 2020: 799-812.

[30] NAM M, CHA H, CHOI Y R, et al. Write-optimized dynamic hashing for persistent memory[C]//Proceedings of the 17th USENIX Conference on File and Storage Technologies. Boston: USENIX Association, 2019: 31-44.

[31] LU B T, HAO X P, WANG T Z, et al. Dash: scalable hashing on persistent memory[J]. Proceedings of the VLDB Endowment, 2020, 13(8): 1147-1161.

2021062-10

[32] ZOU X M, WANG F, FENG D, et al. HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory[C]// Proceedings of the 36th International Conference on Massive Storage Systems and Technology. [S.l.:s.n.], 2020: 11.

作者简介

王永锋（1998- ），男，中山大学计算机学院博士生，主要研究方向为存储系统、分布式系统。

陈志广（1984- ），男，博士，中山大学计算机学院副教授，主要研究方向为大数据存储与处理、并行与分布式计算、高性能计算与超级计算机。在并行文件系统、大规模并行I/O优化、大数据分析处理方面取得了关键技术突破。

收稿日期： 2021-04-15

基金项目： 国家重点研发计划资助项目（No.2018YFB0203904）；国家自然科学基金资助项目（No.61872392，No.61832020，No.U1811461）；广州市珠江科技新星资助项目（No.201906010008）；广东省自然科学基金资助项目（No.2018B030312002）

Foundation Items: The National Key Research and Development Program of China (No.2018YFB0203904), The National Natural Science Foundation of China (No.61872392, No.61832020, No.U1811461), Pearl River S & T Nova Program of Guangzhou (No.N201906010008), Guangdong Natural Science Foundation (No.2018B030312002)