分　数：	
评卷人：	

华中科技大学

研究生（数据中心技术）课程论文（报告）

题 目：基于持久化存储的动态哈希方案综述

学　　号　M202173482

姓　　名　张婧

专　　业　电子信息

课程指导教师　施展 童薇

院（系、所）　武汉光电国家研究中心

2021 年 1 月 3 日

目录

基于持久化存储的动态哈希方案综述

张婧[1)]

[1)](华中科技大学武汉光电国家研究中心 湖北 武汉 430074)

摘 要 持久化存储越来越多的被用于构建基于哈希的索引结构，并且有着低成本持久化、高性能和及时恢复等特性，特别是随着最近 Intel Optane DC Persistent Memory Modules（DCPMM）的发布，PM 越来越火热。然而，一些专注于特定的指标而忽略了其他重要特性的哈希表已经不适用现在的持久化存储。为解决过度的读或写操作、重量级并发控制对带宽的消耗以及及时恢复性能的缺失，学者们对此展开了相关的研究，并提出了不同的哈希表方案，例如 DASH、CCEH 等。本文针对近年来关于持久性存储的动态哈希优化研究，先是介绍动态哈希表发展与非易失存储器件发展的重要联系，分别从硬件结构、操作模式和性能三个方面介绍了 DCPMM 属性，之后阐述了基于存储器发展而提出的几个主流哈希方案，包含了等级哈希(LHC)、持久缓存线哈希表(PCLHT)和基于链的无锁哈希表(SOFT)，接着列出与此相关的三篇文献，从文章的核心技术和设计方案出发，展示了目前前沿的动态哈希方案 Dash 所完成的四项任务：指纹识别、轻量级并发、新的桶平衡策略和即时恢复功能，列举了 CCEH 贡献的三大创新性技术，尤其是提出了三层哈希表结构、中间层段分割技术等。最后通过评估各类方案的物理实验结果，列举和总结了影响动态哈希表性能的重要因素，为之后的优化研究提供了方向。

关键词 持久化存储：DCPMM；并发控制；动态哈希；非易失存储

A Survey on dynamic hashing schemes based on persistent storage

Zhang Jing1)

[1)](Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei 430074)

Abstract Persistent storage is increasingly being used to build hash based index structures with features such as low-cost persistence, high performance, and just-in-time recovery, especially with the recent release of Intel Optane DC Persistent Memory Modules (DCPMM). However, some hash tables that focus on specific metrics and ignore other important features are no longer suitable for persistent storage. In order to solve the problem of excessive read or write operations, the bandwidth consumption of heavy concurrency control and the loss of timely recovery performance, scholars have carried out relevant researches and proposed different hash table schemes, such as DASH and CCEH. In recent years, the author of this paper about the persistent storage dynamic hash optimization study, first introduces dynamic hash table development and an important link in the nonvolatile memory device development, respectively from the hardware structure, operation mode and performance DCPMM properties are introduced, and three aspects elaborated based on the development of storage after several mainstream hash scheme, It includes grade hashing (LHC), persistent cache line hashing (PCLHT) and chain based unlocked hashing (SOFT). Then, three papers related to this are listed. Starting from the core technology and design scheme of the paper, four tasks accomplished by the current cutting-edge dynamic hashing scheme Dash are presented. Fingerprint recognition, lightweight concurrency, new bucket balancing strategy, and instant recovery are listed as three innovative technologies contributed by CCEH, especially the proposed three-layer hash table structure and intermediate layer segmentation technology. Finally, by evaluating the physical experimental results of various schemes, the important factors affecting the performance of dynamic hash table are listed and summarized, which provides a direction for future optimization research.

Keywords Persistent storage: DCPMM; Concurrency control; Dynamic hash; Nonvolatile storage

1 引言

随着非易失存储器技术的发展，持久性内存(PM)正越来越多地用于基于构建的索引结构，具有廉价的持久性、高性能和即时恢复，特别是最近发布的 Intel Optane DC 持久性内存模块。这些特性为对哈希方案特出了新的要求，同时吸引了该领域学者对于新哈希方案的关注和研究热情。研究证明，盲目地应用之前的基于磁盘或 DRAM 的方法，不会重新获得最大的收益，因此迫切需要提出打破传统的设计。而基于持久内存动态哈希方案的出现为解决许多实际应用程序或系统面临的问题提供了一种新的方法。到目前为止，许多新的哈希表设计已经被提出，但大多数都是基于仿真，并在实际的 PM 上执行次优。它们也是分段和局部解决方案，回避了许多重要的特性，特别是良好的可伸缩性、高负载因数和即时恢复。

在实际的 PM 可用之前，有学者提出了一种基于 DRAM 仿真的专门为 PM 设计的新类型哈希表，例如 B. Debnath, A. Haghdoost 等学者提出的回顾相变存储器的哈希表设计、华中科技大学的研究团队提出的为持久内存写优化的动态哈希等等。他们的主要关注点是减少缓存刷新和 PM 写入，以实现可伸缩的性能。但当它们被部署在真正的 Optane DCPMM 上时，常常存在这样俩个问题：一是可扩展性仍然是一个主要问题；二是理想的性能经常被权衡。经过学者的相关研究，限制动态哈希表性能提升的主要因素是 Optane DCPMM 的有限带宽，它比 DRAM 的带宽低 2-4 倍，即使令服务器被充分填充以提供最大可能的带宽，但过多的 PM 访问仍然很容易使系统饱和，阻止系统扩展。因此，需要不断研究和完善新的哈希方案以适应 PM 器件的发展，挖掘 PM 性能优化的最佳方案。

本次主要调研了三篇近三年 CCF/A 类文献。第一篇《Dash：scalable Hashing on Persistent Memory》是 VLDB’2020 的一篇文献。Baotong Lu 等**错误!未找到引用源。**人提出了指纹识别技术、轻量级锁、新的负载平衡策略等方法用于减少 PM 写来带的性能损失以及实现 PM 的重要特性，并在此基础上实现了一个动态且可拓展的整体方法。第二篇《Write-Optimized Dynamic Hashing for Persistent Memory》是 FAST’2019 的一篇文章。Moohyeon Nam 等**错误!未找到引用源。**提出了一种 cacheline - conscious 可扩展哈希(CCEH)算法，它在保证哈希表查找时间不变的情况下，减少了动态内存块管理的开销。第三篇《Persistent Memory Hash Indexes: An Experimental Evaluation》是 VLDB’2019 的一篇文章。Daokun Hu 等学者采用统一的测评框架、选择有代表性的工作负载，对持久化哈希表进行了详细的测评，在真实的 PM 硬件上测试了 6 种最新的哈希表技术，包括 Level hashing, CCEH, Dash, PCLHT, Clevel, SOFT。

2 理论

2.1 Optane DC Persistent Memory

DCPMM 是第一款商用 PM 产品，通过在易失性 DRAM 和基于块的存储之间创建一个新的非易失性层，改变了传统的计算机内存层次结构。

首先，硬件体系结构。Optane DCPMM 是与 Cascade Lake 架构[4]一起引入的，该架构支持多个插槽(2/4/8)，每个插槽由一个或两个处理器模块组成，这些处理器模块组成独立的 NUMA 节点[5]。Cascade Lake 上的集成内存控制器(iMC)能够与 ddr4 和 Optane DCPMM 内存接口。为了保持数据持久性，iMC 位于异步 DRAM 刷新(ADR)域中，可以确保到达这里的 CPU 存储在断电[6]后仍能存活。iMC 为每个 Optane DCPMM 维护读和写挂起队列(RPQs 和 WPQs)，但只有 WPQs 在 ADR 域中。因此，一旦数据到达 WPQs，在系统崩溃时，iMC 将把数据刷新到 PM。由于 3d - Xpoint 物理介质访问粒度为 256 字节，on-DIMM 控制器将较小的请求转换为较大的 256 字节请求，导致写放大，因为较小的存储变成读-修改-写操作。on-DIMM 控制器有一个小的写合并缓冲区(称为 XPBuffer[5])来合并相邻的数据写道。与 WPQs 类似，XPBuffer 位于 ADR 域中。因此，当所有更新到达 XPBuffer 时，它们已经是持久化的了。

其次是操作模式。Optane DCPMM 可以配置为运行在内存模式或应用直接模式。在每种模式下，内存可以跨通道和内存交叉使用。这两种模式都允许直接访问 PM。在内存模式下，DRAM 充当最频繁访问的数据的缓存，而 DCPMM 提供大容量的内存，但没有持久性。在应用直接模式中，应用程序和操作系统明确地知道有两种类型的直接访问内存使用加载/存储指令。值得注意的是，虽然 pm 是持久的，但 CPU 缓存和寄存器仍然是易失的。当 cacheline 刷新指令被执行或其他隐含导致 cacheline

刷新的事件发生[37]时，数据将被持久化到 PM 中。为了确保应用程序能够从系统崩溃中正确地恢复，我们必须使用内存栅栏来避免不需要的指令重新排序。

最后是性能。尽管 Optane DCPMM 设计用于直接和可字节寻址的负载/存储访问，但与 DRAM 相比，它的带宽较低，读/写延迟较高。Optane DCPMM 的读时延为~300ns，是 DRAM (~ 75ns)的 4 倍。此外，它表现出非对称的读/写延迟。根据最近的一项研究[5]，应用程序看到的端到端写延迟通常比读延迟要低，因为一旦数据到达内存控制器的 ADR 主控，写操作就会返回。但是，如果数据没有在 RPQs 中缓存，那么读操作将有很高的概率访问物理 DCPMM 介质。dcpmm 的最大顺序读写带宽约为 40gb /s 和 13gb /s，分别比 DDR4 DRAM 低 3 倍和 11 倍左右。非对称带宽在随机读写时更加突出。

2.2 哈希方案

由于后续会详细介绍 Dash 和 CCEH，因此在此节中不包含对这两种哈希方案的相关介绍。

2.2.1 Level Hashing and Clevel

Level Hashing 是针对 PM 的写优化、可扩展的哈希表，支持低成本的一致性保障和调整。使用基于共享的 2 层结构，来实现常数级别的时间复杂度。图 1 中，只有顶层桶可寻址，底层的桶用于存储从顶层收回的内容。类似于 PCM 友好的哈希表（PFHT），Level Hashing 在每个桶中有多个槽，在往已经满了的桶中插入 1 个记录时允许回收至多 1 个记录。每个键可以从 2 个哈希计算得到的目标位置中选择，与 Cuckoo 哈希类似，从而提高负载因子。

在 Level hash 中，在调整大小的过程中，会创建一个 4 倍大的哈希表，底层桶中的键值对首先被刷希到新表中，然后旧底层桶的键值对被删除。此时，新表成为顶层，而之前的顶层成为底层。为了实现低开销的一致性，在删除、插入、调整大小等操作中，采用了无日志的方式，通过每个桶头部的 bitmap 的进行操作。Level hash 依赖于一个槽粒度的锁来进行并发控制，并且在其数据结构中不维护指针。虽然 Level hash 比之前的工作有改进,但它有两个缺点。首先，它是一个静态的方案，它的重哈希开销仍然很高。其次，它没有在真正的 PM 上进行评估。Clevel 是一个基于 Level hash 的无锁并发哈希表，它提出了一个动态多级结构来支持并发。调整大小是由后台线程执行的，不会阻塞并发查询。

2.2.2 PCLHT

持久缓存线哈希表(PCLHT)是一个基于链的并

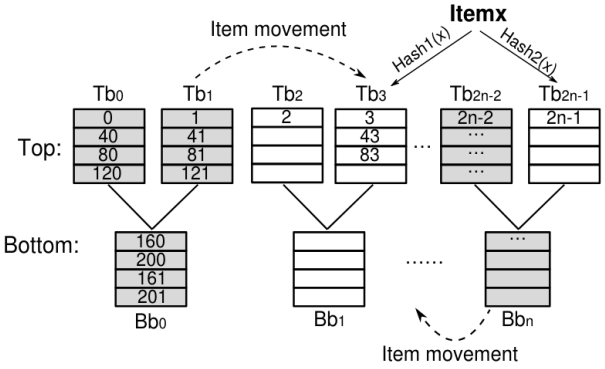

图 1 每带有 4 个槽位的 Level hash 桶结构图

发崩溃一致哈希表，在 RECIPE[7]中讨论过。它是基于 dram 的哈希表 CLHT 的变体。CLHT 是缓存友好的，因为它的桶大小是 64 字节(通常的 cacheline 大小)。每个桶包含一个 8 字节的字(用于并发控制)、一个下一个指针和最多 3 个 16 字节的键值对。这种设计的目的是通过确保在普通情况下每次更新哈希表只需要一次 cacheline 访问来解决缓存一致性问题。为了确保一个非阻塞的读能够找到正确的结果，CLHT 使用桶的原子快照来进行写操作。插入和删除操作都使用一个原子提交点，该提交点按内存栅栏排序:在更新 8 字节的键之前先写入正确的值(用于插入)，或者在键上写入 0(用于删除)。如果由于表已满而导致插入失败，那么 CLHT 将使用写时复制方法调整表的大小。

像插入、删除和调整大小这样的操作，在 PCLHT 中是通过单个原子存储来实现的，所以它们可以通过简单地在相应的存储指令后插入 cacheline flush 和 memory fence 来转换为持久化的对等操作。

2.2.3 SOFT

SOFT[8]是一个基于链的无锁哈希表，被提出以避免在数据结构中持久化任何指针。SOFT 有两个领域组件，即持久节点(PNode)和易变节点(VNode)。持久节点包含一个键值对和三个有效位。易变节点由一个键-值对和两个指针组成:一个指向具有相同键-值对的持久节点，另一个指向下一个易变节点。只有持久节点持久化在 pm 中，所有易变节点都存储在 DRAM 中。在 SOFT 中没有实现调整大小操作。当插入一个密钥时，一个新的持久节点和一个新的易变节点被分配，并且易变节点将被

原子地(通过 CAS)链接到一个现有的易变节点。除了节点被删除之外，删除操作与插入操作具有相似的过程。SOFT 可以通过扫描 PM 中的持久节点和重建 DRAM 中的结构来从系统崩溃中恢复。

3 相关研究

3.1 持久内存上的可伸缩哈希

3.1.1 介绍

为了解决先前哈希方案中过度的点读、重量级的并发控制以及部分功能的缺失。作者提出了 dash，一个动态和可扩展的整体方法，在真实的 PM 上，而不以理想的属性为牺牲。Dash 结合了新的和现有的技术，精心设计以实现这一目标。首先，作者在 PM 树结构中使用指纹识别，以避免在记录探测期间不需要的 PM 读取。这个想法是生成键的指纹(单字节哈希)，并将它们紧凑地放置，以总结键可能存在的情况。这允许线程通过扫描比实际键小得多的指纹来判断一个键是否可能存在。其次，Dash 没有使用传统的桶级锁，而是使用了一种乐观的、轻量级的锁，它依赖于验证来检测冲突，而不是(昂贵的)共享锁。这允许 Dash 避免为搜索操作编写 PM。通过指纹识别和乐观的并发性，Dash 避免了不必要的读写，节省了 PM 带宽，并允许 Dash 很好地扩展。最后，Dash 保留了理想的属性。文章提出了一种新的负载平衡策略，以提高空间利用率来推迟分段。为了支持即时恢复，我们将恢复时要做的工作量限制在很小的范围内(读取和可能写入一个字节计数器)，并将恢复工作分摊到运行时。与以前以特殊方式处理 PM 编程问题的工作相比，Dash 使用 PM 编程模型(PMDK，最流行的 PM 库之一)系统地处理崩溃一致性、PM 分配并实现即时恢复。

虽然结合了当下的许多相关技术，但 Dash 是第一个将它们集成在一起的系统方案，构建了可扩展的哈希表并且不以牺牲实际 PM 特性为低价。Dash 中的技术可以应用于各种静态和动态哈希方案。在本文中重点介绍了 Dash 的两种经典应用:可扩展哈希和线性哈希。它们都广泛应用于数据库和存储系统。

3.1.2 动态哈希

（1）可拓展的散列

可扩展哈希的关键是它使用一个目录来索引桶，以便它们可以在运行时动态地添加和删除。当一个桶满了，它被分成两个新桶，并重新分配键。如果没有足够的空间来存储指向新桶的指针，目录可能会被扩展(加倍)。图 4 左图显示了一个包含四个桶的示例，每个桶由一个目录条目指向;一个桶最多可以存储 4 条记录(键值对)。在图中，目录条目的索引用二进制表示。哈希值的两个最低有效位(LSBs)用于选择桶;Dash 把这里使用的后缀位的数目称为全局深度。哈希表最多可以有 2 的全局深度次方个目录条目。一个搜索操作跟随对应目录项中的指针来探测桶。每个桶也有局部深度。在图 4 左图中，每个桶的局部深度为 2，与全局深度相同。假设我们想插入键 30，它被散列到桶 012，而桶 012 是满的，需要分割以容纳新键。分割桶将需要更多的目录条目。在可扩展哈希中，目录总是以当前大小的两倍增长。结果如图 2 右图所示。在这里，图 2 左图中的桶 012 被分成两个新桶(0012 和 1012)，一个占用原始目录条目，另一个占用新添加的目录一半中的第二个条目。其他新条目仍然指向它们相应的原始桶。搜索操作将使用三位来确定目录条目索引(全局深度现在是三位)。

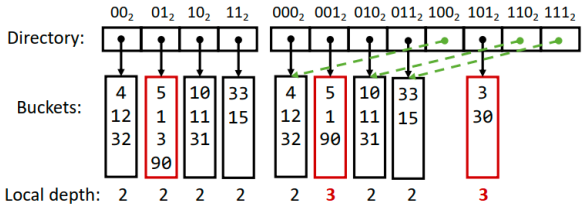

图 2 关于可拓展哈希的举例

在一个桶被分割后，Dash 将它的局部深度增加 1，并将新桶的局部深度更新为相同(在我们的例子中是 3)。其他未拆分桶的本地深度为 2。这允许我们确定是否需要目录加倍:如果一个局部深度等于全局深度的桶被分割(例如，桶 0012 或 1012)，那么这个目录需要加倍以适应新的桶。否则（当地深度小于全球深度)，目录上应该有 2 个条目指向该桶，并且该条目的本地深度等于全局深度。

（2）线性散列

内存中线性散列采用类似的方法来组织存储桶，使用一个目录，其中的条目指向单个存储桶[29]。与可扩展哈希的主要区别是，在线性哈希中，要分割的桶是“线性”选择的。也就是说，它保留一个指针(页面 ID 或地址)，指向下一个要拆分的桶，并且在每轮中只拆分那个桶，并在当前桶的拆分完成后将指针向前移动到下一个桶。因此，被分裂的桶不一定与被插入的桶相同，最终溢出的桶将被分裂并

重新分配其键。如果一个桶已经满了，请求插入它，更多的溢出桶将被创建，并与原来的满桶链接在一起。为了正确的寻址和查找，线性哈希使用一组哈希函数$h_1 \ldots h_n$，其范围为h_{n-1}的两倍。对于已经分裂的桶，使用h_n，以便我们可以在新的哈希表容量范围内处理桶，而对于其他未分裂的桶，我们使用h_{n-1}来找到所需的桶。在所有的桶都被分割之后(拆分完成)，哈希表的容量将翻倍，指向下一个拆分桶的指针将被重置为第一个桶，以便进行下一轮拆分。

3.1.3 Dash的核心任务

（1）指纹技术

桶探测(即在一个桶中搜索)是哈希表支持的所有操作(搜索、插入和删除)所需要的基本操作，以检查键是否存在。搜索一个桶通常需要对槽进行线性扫描。这可能会导致大量缓存丢失，并且是 PM 读取的主要来源，特别是对于存储为指针的长键。这是 PM 上哈希表表现出低性能的主要原因。此外，这种对负搜索操作的扫描(当目标键不存在时)完全不必要。Dash 使用指纹系统来减少不必要的扫描。

Dash 在哈希表中采用指纹来减少缓存和加速探测。指纹是一个字节的键散列，用于预测一个键是否可能存在。是我们使用键哈希值的最小有效字节。要探测一个 key， 探测线程首先检查是否有指纹匹配搜索键的指纹。然后，它只访问具有匹配指纹的 slots，跳过所有其他 slots。如果没有匹配，key 肯定不在桶中。使用 SIMD 指令[9]可以进一步加速这个过程。

指纹特别有利于负搜索(在不存在搜索键的情况下)和插入的唯一性检查。它还允许 Dash 使用更大的桶来容忍更多的碰撞，并提高负载系数，而不会导致许多缓存遗漏:大多数不必要的探测都可以通过指纹来避免。这一设计与先前的设计形成对比，这些设计通过使用 1-2 个 cachelines 的小桶来交换负载因子以提高性能。如图 3 所示，每个桶包含 14 个槽，但 18 个指纹(位 64-208);14 表示桶中的槽，4 表示放在存储桶中的键，但最初被散列到当前桶中。它们可以尽早避免访问存储桶，节省 PM 带宽。

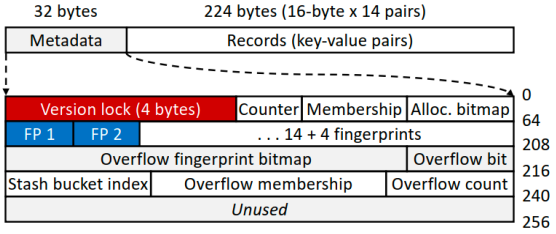

图 3 桶的层次结构图

（2）桶负载平衡策略

策略一平衡的插入，为了插入一个其键被哈希到桶 b 的记录，Dash 探测桶 b+1 并将该记录插入到较少满的桶中(图 4 步骤 1)。其背后的基本原理是，通过摊销热桶的负载，同时限制 pm 访问(最多两个桶)，来提高效率。与平衡插入相比，线性探测允许将一条记录插入到桶 b+n（n>1）。如果 buckets b…b + n−1 都满了，那么探测多个桶可能会导致更多的 PM 读取和缓存丢失，从而降低性能。而调整要探测的桶的数量也相当困难。

策略二取代，如果两个目标桶带探测桶 b+1 都满了，那么 Dash-EH 尝试从桶 b+1 置换(移动)一条记录，为新记录腾出空间。对于平衡插入，桶 n+1 中的记录可以被移到 n +2，前提是(1)它可以被插入到任意一个桶中(即，n+2 是正在移动的记录的探测桶)， (2)bucketn+2 有一个空闲槽。因此，对于一个 hash(key) =b 的记录，如果桶 b 和桶 b+1 都是满的，Dash 首先尝试在 b+1 中找到一个 hash(key) =b+1 的记录，并将其移动到 b+2。如果这样的记录不存在，我们对桶 b 重复使用 hash(key) =b−1(目标桶)移动记录。本质上，置换遵循与平衡插入相似的策略，但是针对现有记录。

策略三储藏桶，如果在平衡插入和取代之后，记录不能被插入到 bucketborb+1 中，那么在段分割发生之前，存储将是最后的手段。在图 6 中，每个段中的正常桶后面跟着可调数量的存储桶。如果一个记录不能被插入到它的目标桶或探测桶中，Dash 将该记录插入到存储桶中。

（3）乐观的并发

Dash 使用乐观锁定，这是桶级锁定的一种乐观风格。插入操作将遵循传统的桶级锁定来给受影响的桶上锁。搜索操作被允许在不持有任何锁的情况下继续进行(从而避免写 PM)，但需要验证读记录。要做到这一点，在 Dash 中，锁由(1)一个充当“锁”角色的位和(2)一个用于检测冲突的版本号组成。这是通过尝试比较和交换(CAS)指令[9]来自动设置每个桶中的锁位，直到成功。然后线程进入临界区并继续其操作。在插入完成后，线程通过(1)重置锁位和(2)用原子写一步递增版本号来释放锁。

3.1.4 实验评估

为了评估 Dash 提出的核心技术的性能，作者从 5 各方面进行了对比实验。

（1）从单线程性能开始，选择固定长度键的只读工作，展示不停方案底层设计的缓存效率和并发

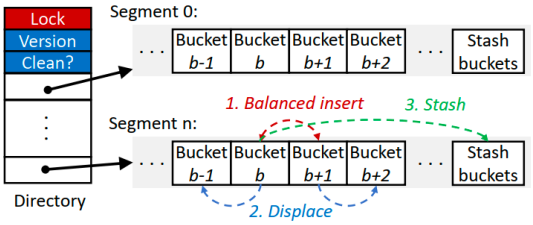

图 4 Dash-EH 的完整结构图

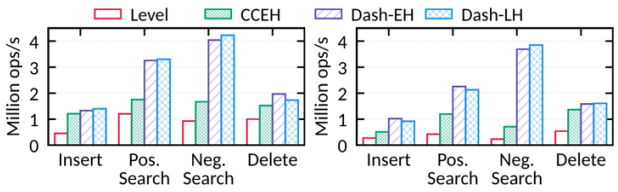

图 5 定长键和变长键下的单线程性能

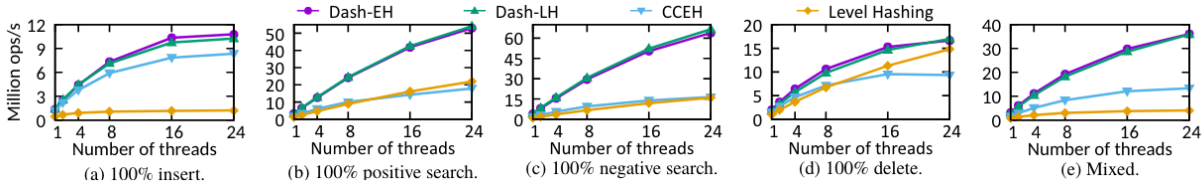

图 6 不同工作负载下的线程数量对吞吐量的影响

控制的开销。

（2）分别测试单一工作负载和混合工作负载的性能，对比 Dash 方案与其他方案的性能差异。

（3）检测指纹技术对性能优化的影响

（4）检测元数据对性能优化的影响

（5）研究线性探测、平衡插入、位移和叠加对负载因子的影响。

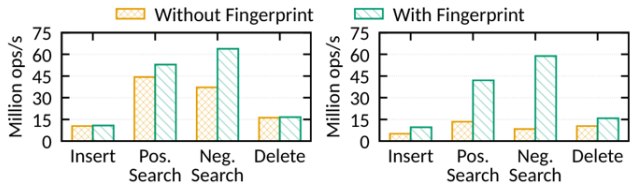

图 7 指纹技术在不同负载下对性能的影响

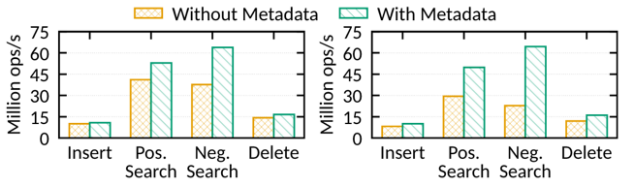

图 8 元数据在不同负载下对性能的影响

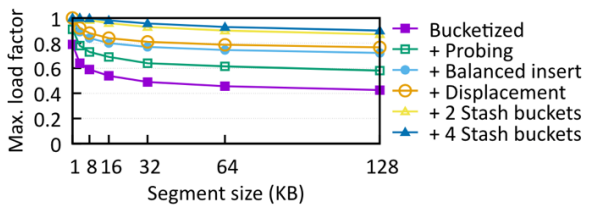

图 9 各项技术对负载因子的影响

3.2 持久内存的写优化动态哈希

3.2.1 介绍

低延迟存储媒体，如可字节寻址持久内存(PM)，需要从优化的角度重新考虑各种数据结构。在 PM 上实现基于哈希的索引结构的主要挑战之一是如何通过有效地使用 cacheline 来实现效率，同时保证故障原子性和动态哈希扩展和收缩。在文章中提出了一种可扩展哈希(CCEH)算法，它在保证哈希表查找时间不变的情况下，减少了动态内存块管理的开销。CCEH 保证了故障原子性，而没有使用显式日志记录。我们的实验表明，在细粒度的失败原子性约束下，CCEH 能够有效地随着需求的增加而适应其大小，其最大查询延迟比最先进的哈希技术低一个数量级。

3.2.2 CCEH的核心任务

（1）三级结构

为了在目录大小和查找性能之间取得平衡，文章提出在目录和存储桶之间使用一个中间层，称之为段。也就是说，段是由目录指向的一组桶。CCEH 的结构如图 10 所示。为了在三层结构中定位一个桶，我们使用 G 个比特位 (代表全局深度)作为一个段索引，并附加 B 个比特位(决定一个段中 cacheline 的数量)作为一个桶索引来定位一个段中的桶。

在图 10 所示的例子中，我们假设每个桶可以存储两条记录(用图中分段内的实线分隔)。如果我们使用 B 位比特作为桶索引，与目录直接访问每个桶相比，我们可以减少 1/2B(在本例中为 1/256)的目录大小。注意，尽管三层结构减少了目录大小，但它允许访问一个特殊的桶(cacheline)而不访问段中不相关的缓存线。

（2）故障原子段分割

拆分执行大量内存操作。因此，在 CCEH 中，段分割不能由单个原子指令执行。与需要对哈希表指针进行单个失败原子更新的全表哈希不同，可扩展哈希设计用于重用大多数片段和目录项。因此，

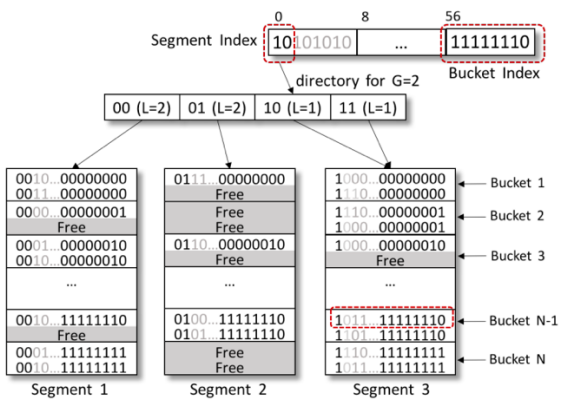

图 10 CCEH 的结构

可扩展哈希的段分割算法支持在目录和写复制时执行几个就地更新。

文章使用图 11 例子来详细介绍故障原子段分割算法的工作原理。假设我们插入键 1010…11111110(2)。段 3 的左首位为 1，但段中的第 255 个(11111111(2)th)桶没有空闲空间，即发生哈希冲突。为了解决哈希冲突，CCEH 分配了一个新的段，并根据它们的哈希键复制键值对，不仅在该段的碰撞桶中，而且也在同一段的其他桶中。在本例中，该算法分配了一个新的段 4，并将键前缀以 11 开头的记录从段 3 复制到段 4。

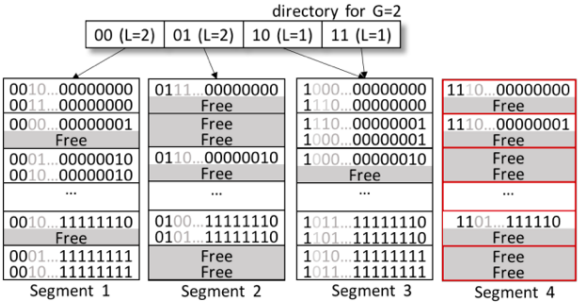

图 11 哈希冲突引发的段分割

下一步是更新段 4 的目录条目，如图 11 所示。首先，新桶的指针和本地深度被更新。然后，更新分割片段的局部深度（片段 3）。也就是说，从右到左更新目录条目。这些更新的顺序必须通过在每个指令之间插入一个 fence 指令来强制执行。此外，当它跨越缓存线的边界时，我们必须调用 flush，就像在 FAST 和 FAIR B-tree[11]中所做的那样。执行这些更新的顺序对于保证恢复特别重要。也就是说，如果在段分割期间系统崩溃，目录可能会发现自己处于部分更新的不一致状态。例如，更新后的指向新段的指针被刷新到 PM，但是两个本地深度在 PM 中没有更新。而这种不一致可以很容易地通过恢复过程检测和修复，而无需显式的日志记录。

然而，段分割会导致低负载因素和更多的 PM 访问，因为段中的其他桶可能仍然有空闲的槽位。因此，CCEH 使用线性探测(在分裂之前)试图提高空间利用率。

（3）懒惰删除

在遗留的可扩展哈希中，拆分后通过页写自动清理桶，这样桶就不会有迁移的记录。对于失败原子性，基于磁盘的 ex- tenable 哈希更新本地深度，并通过单页写删除迁移的记录。

与以前的可扩展哈希不同，CCEH 不会从分割段删除已迁移的记录。如图 12 所示，即使段 3 和段 4 有重复的记录。一旦目录条目更新，搜索迁移记录的查询将访问新段，搜索非迁移记录的查询将访问旧段，保证了搜索操作的正确性，只是带有一些不必要的重复。

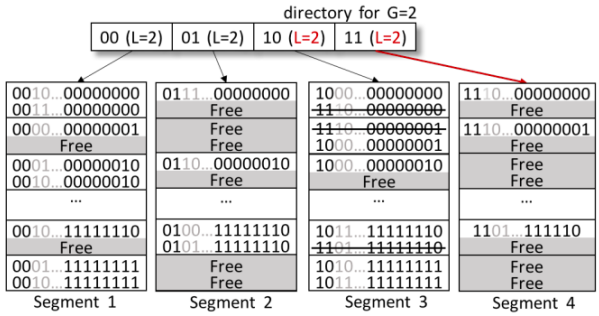

图 12 段分割和懒惰删除

文章提出了懒惰删除，这有助于避免昂贵的“写时复制”并减少分割开销。一旦我们在目录条目中增加分割段的本地深度，迁移的键(图 12 中被划掉的键)将被后续事务认为是无效的。因此，它们会被读事务忽略，并且它们会被随后的插入事务以一种惰性的方式覆盖。

3.2.3 实验评估

为了评估 CCEH 展示的核心技术性能，作者从以下几个方面进行了实验。

（1）量化每个 CCEH 设计的性能效果，检测段大小分别对插入吞吐量、cacheline 刷新数以及搜索吞吐量的影响。从图 13 实验结果可以看出，当段大小增加到 16KB 时，插入吞吐量会增加，因为段分割发生的频率降低了，而读取 cacheline 的数量不受段大小的影响。但是，高于 16KB 以后，段分割带来大量的 cacheline 刷新，带来性能的降低。

（2）固定单字节大小的桶索引，选择更优的 MSB 方案，在不同读写延迟环境中比较 CCEH 与

其他静态哈希表的性能表现，包含读取和搜索探测时长、重哈希时长以及记录转移时长等方面的性能

（3）检测 CCEH 的并发性能，比较展示 CCEH 性能的优越性，包括插入延迟时长与 CDF 的关系，线程数量对插入/搜索吞吐量的影响。

3.3 持久内存哈希索引：一个实验评估

3.3.1 介绍

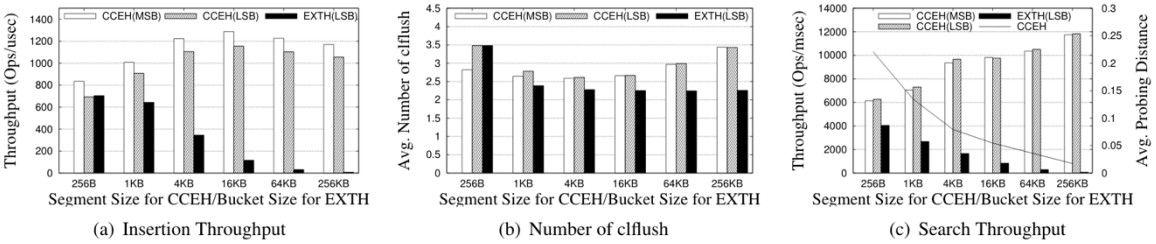

图 13 段大小对吞吐性能的影响，其中(a)插入操作吞吐量，(b)clflush 的数量，(c)搜索操作吞吐量

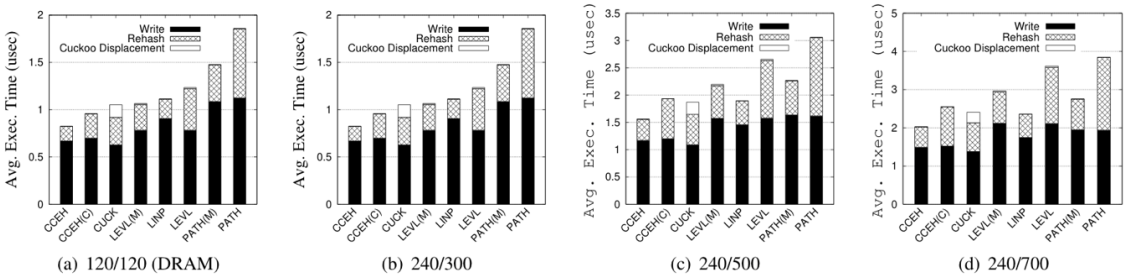

图 14 并发执行的性能:延迟 CDF 和插入/搜索吞吐量,其中(a)读写延迟为 120/120 (b)读写延迟为 240/300(c) 读写延迟为 240/500(d) 读写延迟为 240/700

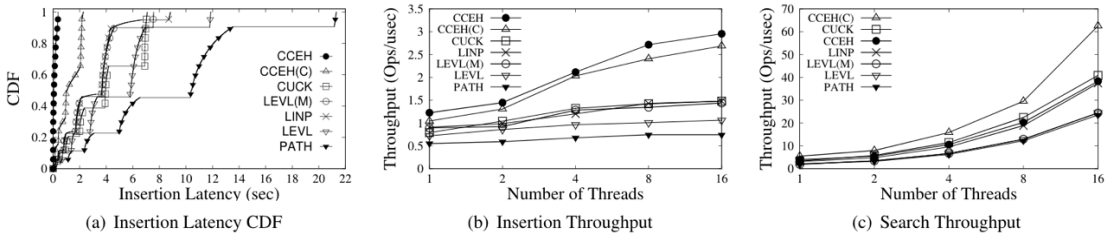

图 15 并发执行的性能:延迟 CDF 和插入/搜索吞吐量，其中(a)插入延迟分布函数(b)插入操作吞吐量(c)搜索操作吞吐量

持久性存储(PM)正在越来越多地被用于构建基于哈希的索引结构，具有廉价的持久性、高性能和即时恢复，特别是最近发布的 Intel Optane DC 持久性内存模块。然而，它们中的大多数都是在基于 dram 的模拟器上进行评估，并带有不真实的假设，或者专注于评估特定的指标，而忽略了重要的属性。因此，如果考虑到更大范围的性能指标，就必须了解提议的哈希索引在实际 PM 上的表现如何。

为此，文章提供了一个持久哈希表的综合评价。特别是，文章将重点放在使用实际 PM 硬件评估 6 个最先进的哈希表，评估使用统一的基准框架和具有代表性的工作负载。除了描述常见的性能属性外，还探讨了硬件配置(如 PM 带宽、CPU 指令和 NUMA)如何影响基于 PM 的哈希表的性能。通过文章的深入分析，确定了设计的权衡和现有技术中的好范例，并为基于 PM 的哈希表的未来发展提出了理想的优化和方向。

3.3.2 实验结果

1. 随机小记录写入是从根本上限制 PM 哈希表性能的主要因素。随机小记录写入是哈希表固有的访问模式。因此，即使通过特定的优化来解决这个问题，比如减少随机写操作的数量，现有 pm 哈希表的性能从根本上还是受到限制的。文章认为，新哈希表设计的目的是为了减轻随机小写入的不利影响，实现更高的性能。

2. 缓存刷新和内存栅栏指令对性能的影响很小。与在使用软件事务内存的 PM 数据结构设计中所做的常见观察相反，PM 哈希表的性能不受缓存刷新和内存栅栏指令的限制，因为这些指令在单个操作中的数量有限。因此，优化应该集中在设计 PM 哈希表的其他方面。

3.指纹可以显著地加速负面查询。缓存驻留的指纹可以大大加速负面搜索，因为对 PM 的不必要访问可以被过滤。这对于不对称读写的 PM 来说尤其有意义。

4. 调整大小不应阻碍正常操作。全表重哈希对并发性有害，会导致低吞吐量和意外延迟。动态可扩展方案(Dash 和 CCEH)和无锁调整大小(Clevel)是很好的范例，可以减轻调整大小的开销。

5．注意由内存分配器和同步原语引起的写放大的重要性。除了由小的随机写引起的写放大(这是哈希表固有的)，内存分配器和同步原语也会引起严重的写放大，这可能远远超出了作者在本文中量化的预期。写入放大不仅降低了 DCPMM 的性能，而且损害了 DCPMM 的耐久性。在未来，我们计划探索为 PM 系统和 PM 友好锁设计穿戴式内存分配器。

6.对混合内存设计的关注。在 DRAM 中维护结构元数据和在 PM 中维护键值对可以大大减少 PM 读/写和访问延迟。然而，未来应该考虑新的技术来降低混合存储器设计的回收成本。

7. PM 带宽是一种稀缺资源，但是哈希表的性能并不一定会随着 DCPMM 的增加而增加。对于某些散列表，当系统中安装更多 DCPMM 时，它们的性能几乎是线性扩展的。而对于其他情况来说，吞吐量的增加很小，甚至几乎不引人注意。需要更多的努力来解决 PM 哈希表的可伸缩性问题。此外，希望使用现有哈希表的系统设计人员将意识到在 DCPMM 和 DRAM 资源之间实现平衡的重要性，以实现更好的性能

8. PM 问题的微观结构特征。我们已经说明了哈希表的设计者应该意识到 XPBuffer 的存在和重要性。此外，文章希望看到更多的特征研究来揭示性能关键的微架构组件。

3.3.3 小结

本文对基于最近发布的 Intel Optane DC 为 PM 设计的哈希表进行了全面的评估。作者为 pibench 框架扩展了更多特定于哈希表的功能，并详细评估了六个状态索引，不仅考虑了常见的指标，还考虑了硬件相关的属性，如 PM 带宽、flush/fence 指令和 NUMA 架构。文章确定了设计基于 PM 的哈希表的关键见解。希望实验结果对研究人员和实践者开发更有效和可扩展的基于 PM 的哈希表有价值。

4 总结

由于持久化存储器件制作工艺的发展，为满足持久性内存的性能需求和发挥其重要功能和属性，对持久哈希表提出了新的要求。通过不断的研究，目前已经发现了影响 PM 性能优化的重要原因，以及提出了几种主流的解决方案。但是对于持久哈希表性能优化问题还有待进一步优化。本文提出了三篇相关的研究文献。首先是目前比较前沿的 Dash 方案，建立在 EECH 的技术基础上并解决了 EECH 的问题，保留了 PM 的三大重要特性。其次是 EECH 方案，作者贡献了三层哈希表结构、段分割技术以及懒惰删除等技术，给动态哈希表更新提出新的见解。最后是关于哈希方案的实际物理评估，作者通过控制变量实验验证了各类哈希方案的有效性，提取了对于哈希表性能影响的重要因素，对后续 PM 哈希表的研究提供有力支持。

5 参考文献

[1] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash:ScalableHashing on Persistent Memory. Proceedings of the VLDB Endowment13, 8 (2020),1147–1161. https://doi.org/10.14778/3389133.3389134

[2] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong.1979. Extendible Hashing-a Fast Access Method for Dynamic Files. ACM Trans.Database Syst.4, 3 (1979),315–344. https://doi.org/10.1145/320083.320092

[3] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen.Persistent Memory Hash Indexes: An Experimental Evaluation. PVLDB,14(5): 785 - 798, 2021.

[4] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi,Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedara-man, et al.2019. Cascade lake: Next generation intel xeon scalable processor.IEEE Micro39, 2 (2019), 29–36

[5] Jianhua Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and StevenSwanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persis-tent Memory. InProceedings of the 18th USENIX Conference on File and StorageTechnologies (FAST'20)

[6] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, AmirsamanMemaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, JishenZhao, and Steven Swanson. 2019. Basic Performance Measurements of theIntel Optane DC Persistent Memory Module.CoRRabs/1903.05714 (2019).arXiv:1903.05714

[7] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chi-dambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. InProceedings of the 27th ACM Symposium on Operating Sys-tems Principles (SOSP'19). ACM, Huntsville, Ontario, Canada, 462–477. https://doi.org/10.1145/3341301.3359635

[8] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank.2019. Efficient Lock-Free Durable Sets.Proceedings of the ACM on ProgrammingLanguages3, OOPSLA, Article 128 (2019), 26 pages. https://doi.org/10.1145/3360554798

[9] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual. 2015.