分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题 目：多层内存系统的页面管理综述

学　　号 M202173842

姓　　名 赵子龙

专　　业 电子信息

课程指导教师 施展 童薇

院（系、所） 计算机科学与技术学院

2021 年 1 月 2 日

多层内存系统的页面管理综述

赵子龙 M202173842

摘　要　随着分层内存系统(包括各种类型的内存，如 DRAM 和 SCM)的出现，操作系统对内存管理的支持变得越来越重要。然而，当前操作系统管理页面的方式是在所有内存具有基于 DRAM 的相同功能的假设下设计的。这种过度简化导致分层内存系统中的内存使用不是最佳的。本研究深入分析了当前 Linux 设计中的页面管理方案，将 NUMA 扩展到支持同时配备 DRAM 和 SCM(英特尔的 DCPMM)的系统。在这样的多层内存系统中，我们发现影响性能的关键因素不仅是访问位置，还有内存的访问层。当考虑到这两种特征时，有几种替代页面位置的方法。然而，当前的操作系统只优先考虑访问局部性。本文探讨了页面管理方案的设计空间，称为自动分级，以有效地使用多层内存系统。本文还介绍了 NAP，这是一种黑盒方法，可以将并发持久内存(PM)索引转换为 NUMA-aware 的对应索引。根据观察，真实世界的工作负载总是具有倾斜的访问模式，NAP 在现有并发 PM 索引的顶部引入了 NUMA-aware 层(NAL)，并将对热门项目的访问引导到这一层。NAL 在 PM 中维护 1)每个节点的局部视图，用于服务插入/更新/删除操作的失败原子性;2)在 DRAM 中维护全局视图，用于服务查找操作。NAL 消除了对热项目的远程 PM 访问，而不诱导额外的本地 PM 访问。此外，NAP 采用了快速的 NAL 切换机制来处理动态的工作负载。

关键词　内存系统；动态随机存取存储器 DRAM；Optane DC 持久存储模块 DCPMM；非均匀内存访问 NUMA；页面替代；Linux；

Overview of page management in multi-tiered memory system

Zilong zhao

Abstract　With the arrival of tiered memory systems comprising various types of memory, such as DRAM and SCM, the operating system support for memory management is becoming increasingly important. However, the way that operating systems currently manage pages was designed under the assumption that all the memory has the same capabilities based on DRAM.This oversimplification leads to non-optimal memory usage in tiered memory systems. This study performs an in-depth analysis of page management schemes in the current Linux design extending NUMA to support systems equipped with both DRAM and SCM (Intel’s DCPMM). In such multi-tiered memory systems, we find that the critical factor in performance is not only the access locality but also the access tier of memory. When considering both characteristics, there are several alternatives to page placement. However, current operating systems only prioritize access locality. This paper explores the design space of page management schemes, called AutoTiering, to use multi-tiered memory systems effectively.This paper also presents Nap, a black-box approach that converts concurrent persistent memory (PM) indexes into NUMA-aware counterparts. Based on the observation that real-world workloads always feature skewed access patterns, Nap introduces a NUMA-aware layer (NAL) on the top of existing concurrent PM indexes, and steers accesses to hot items to this layer. The NAL maintains 1) per-node partial views in PM for serving insert/update/delete operations with failure atomicity and 2)a global view in DRAM for serving lookup operations. The NAL eliminates remote PM accesses to hot items without inducing extra local PM accesses. Moreover, to handle dynamic workloads, Nap adopts a fast NAL switch mechanism.

Key words　memory system;Dynamic Random Access Memory DRAM;Optane DC Persistent Memory Module DCPMM;Non Uniform Memory Access NUMA;alternatives to page placement;Linux;

1 引言

随着内存计算的出现，如数据分析、键值存储和图形处理，对高密度 DRAM 的需求近年来稳步增长。然而，由于缩放 DRAM 密度的挑战，一种新的内存类别已受到关注，以弥补 DRAM 和 SSD 之间的性能差距。例如，英特尔最近推出了基于 3D Xpoint 技术的非易失性内存，称为 Optane DC Persistent Memory Module(DCPMM)，它提供了比 DRAM 更大的密度，同时优于基于闪存的 SSD。云厂商如谷歌，Oracle, Microsoft 和百度已经在他们的云服务中采用了这样的存储类内存(SCM)。

数据中心通常使用多芯片 NUMA 架构[7]来提高具有高核数和内存容量的普通服务器的性能。虽然这可以增加每个服务器的 DIMM 插槽的数量，但缩放 DRAM 密度仍然是一个重大的障碍。它在成本有效地构建大型存储系统方面提出了挑战。与此同时，由于 SCM 提供了字节寻址和非易失性的属性，因此它在弥补 DRAM 和 SSD 之间的性能差距方面越来越受欢迎。英特尔最近发布了无需修改就可以安装在 DIMM 上的 3D XPoint 非易失性内存(DCPMM)。许多云厂商，如谷歌、微软、Oracle 和百度，已经在其云服务中采用了英特尔的 DCPMM。最近，三星电子推出了基于 CXL (Compute Express Link)的 DRAM 模块[3]，与该系统相连，形成了分层存储系统。由于这种新型存储器的速度不如 DRAM，因此不能完全替代 DRAM。相反，未来的计算机系统将提供一种具有 DRAM 和 SCM 的分层存储体系结构形式。

在本文中，我们利用 DCPMM 作为 DRAM 和 SSD 之间的新层。Intel DCPMM 提供两种类型的分层内存系统，可分为硬件辅助和软件管理。在硬件辅助模式下，DCPMM 作为主内存暴露给软件，而 DRAM 作为硬件管理的缓存，对软件不可见。内存控制器自动将经常访问的数据放在 DRAM 缓存中，而其余的数据保存在大容量但速度很慢的 DCPMM 上。另一方面，有了操作系统的支持，DRAM 和 DCPMM 都可以作为普通内存公开，并对软件可见，将内存分为快的和慢的[15]。我们称之为软件管理的分层内存系统。在这种环境中，操作系统支持应该有效地使用 DRAM 和 DCPMM，因为完全的控制是交给软件的。本文通过理解硬件是如何组织的来关注分层存储系统的系统软件方面。

虽然之前已经有大量的研究设计高性能的 PM 索引，但非均匀内存访问(NUMA)体系结构对 PM 索引的影响尚未深入探讨。由于单个 CPU 的内存插槽和核心有限，NUMA 架构是提供大量带宽和 PM 容量以及巨大计算能力的必要条件。在 NUMA 机器中，CPU 内核和 DRAM/PM 内存被分组成节点，通过节点间的链路相互连接。

PM 索引上的 NUMA 问题是唯一的。首先，PM 受到 NUMA 的影响比 DRAM 更严重。具体来说，在第一个 PM 产品 Intel Optane DC Persistent Memory(即 Optane DIMM)中，与本地 PM 写入相比，远端 PM 写入的峰值带宽降低到 59%;更糟糕的是，高并发远程 PM 写(即超过 8 个线程)会经历带宽悬崖。其次，为了保证故障的原子性(即，系统可以在系统崩溃时恢复到正确的状态)，PM 索引应该发出 flush 指令，明确地将数据从 CPU 缓存中刷新到 PM。对于驻留在远程节点上的数据，这些 flush 指令暴露关键路径上的远程 PM 写操作，从而降低性能。第三，PM 的带宽有限(写和读的带宽分别是内存的 1/6 和 1/3)，使得基于复制的方法不切实际。现有的 NUMA-aware 的 DRAM 索引总是(部分)在 NUMA 节点上复制索引，并通过紧凑的操作日志同步这些副本。复制有效地减少了远程访问;然而，由于每个更新操作都在每个节点上执行，因此本地访问的数量显著增加。虽然这种放大对 DRAM 来说不是问题，因为它具有极高的本地带宽，但对于低本地带宽的 PM 来说是致命的。

在本文中，我们提出了 AutoTiering 探索多层内存系统的页面管理设计空间的两种方案和 NAP (NUMA-Aware Persistent Memory Indexes)将并发 PM 索引转换为 NUMA-Aware 对应索引，减少远程 PM 访问，而且不消耗额外的本地 PM 带宽。

2 原理和优势

由于现代服务器系统采用非统一内存访问(Non-Uniform Memory Access, NUMA)架构构建，未来的大内存系统将采用传统 NUMA 架构上的分层内存，称为多层内存。图 1 展示了在整个研究过程中使用的真实的多层存储系统。每个计算芯片都有两种类型的内存:DRAM(上层)和英特尔的 DCPMM(下层)。我们将 DRAM 和 DCPMM 配置成作为内存完全暴露于软件中。

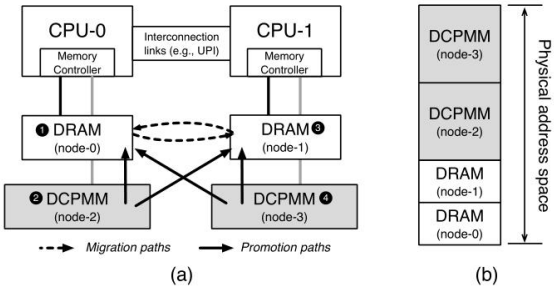


图 1:在 NUMA 架构上扩展的软件管理分层内存系统

当新的内存类型成为主内存的一部分时，影响性能的关键因素不仅是访问位置，还有内存的访问层。然而，目前的页面放置方案已经建立在仅 DRAM 的 NUMA 架构上，只考虑线程和内存之间的局域性。因此，目前的设计远远没有发挥多层存储系统的潜在好处。例如，假设当将页面从底层(DCPMM)提升到上层(DRAM)内存时，本地 DRAM 已经满了。在这种情况下，无论远程 DRAM(上层的)的可用性如何，当前的技术状态都将页面留在较低层内存中。这样的决定对于只有 DRAM 的 NUMA 系统是合理的，因为不同的替代方案之间没有差别。然而，在多层存储系统中，我们不能考虑每一个可能的替代等效于访问层。在放置页面时，应该先考虑内存的访问层，然后再考虑访问位置，因为访问层对性能的影响更大。

这种限制促使我们重新审视普通操作系统的页面管理方案，并探索多层内存系统的页面管理设计空间。本研究介绍了一套新的页面管理方案。第一个方案是 AutoTiering - CPM(Conservative Promotion and Migration)，它在无法找到最佳内存节点(例如，本地 DRAM)时，使用访问层和本地度量来保守地寻找升级或迁移替代方案。

尽管这种保守的方法可以通过考虑替代方案获得更好的性能，但这样的设计并不能充分发挥软件管理的分层内存的潜力。为了有效地利用有限的上层内存容量，我们设计了一种针对多层内存系统的页面回收方案。我们的第二种技术是机会性的页面提升或迁移，称为 AutoTiering-OPM(Opportunistic Promotion and Migration)，它明智地将页面从上层内存降级。为了恢复效率，我们通过估计页面的访问频率来预言被访问次数最少的页面为上层内存中的目标页面。在决定提升哪个页面时，我们的 OPM 会将该页面与目标页面进行比较，以确定哪个页面的访问量相对较高。使用 OPM，我们可以在减少对底层内存的内存访问的同时，实现上层内存的更好的效率。

除非上层内存中有空闲空间，否则提升操作将等待直到降级操作完成。为了隐藏从关键路径降级页面的延迟，我们在上层内存中保留了一组空闲页面，以便立即为提升请求服务。当保留的页面数量超过阈值时，kdemoted 唤醒并将访问次数最少的页面回收到后台的空闲页面池中。Kdemoted 与传统回收不同，因为它只负责将页面降级到较低级别的内存，而不是存储。

本文还提出了 NAP (NUMA-Aware Persistent Memory Indexes)，这是一种黑盒方法，可以将并发 PM 索引转换为 NUMA-Aware 对应索引。NAP 基于一个常见的观察结果:真实的工作负载总是以倾斜的访问模式为特征，其中一小部分热门条目接收到非常频繁的访问。NAP 的关键思想是制造热访问 NUMA-aware。NAP 引入了一个通用的 NUMA-aware 层(NAL)，它可以放在任何现有并发 PM 索引的顶部。NAL 吸收对热项目的访问，而底层的 PM 索引处理对其他项目的访问。具体来说，NAL 在 PM 中维护每个节点的局部和崩溃一致视图(partial and crash-consistent views,PC-view)，这些视图从本地线程提供插入/更新/删除操作，这些操作具有失败原子性。NAL 不同步 PC -view 之间的状态，以避免远程 PM 访问并且不诱导额外的本地 PM 访问。这种无同步的方法带来了两个挑战:1)为热门项提供查找操作;2)从多个 PC -view 恢复中识别最新的值。对于 1)，NAL 在 DRAM 中维持一个额外的全球热点项目视图。对于 2)，NAL 采用了一种基于版本的机制，将插入/更新/删除操作排序到相同的项上，并采用了低开销的失败原子性方法。

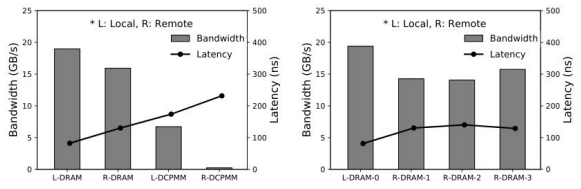
当工作负载发生变化时，NAP 可以识别新的热项集，然后快速切换到新的 NAL。热集识别是通过结合精确高效的流算法(如 count-min sketch)来实现的。为了缓解 NAL 开关过程中前台索引操作的阻塞，NAP 引入了三相开关。该机制通过基于宽限期（grace-period-based)的轻量级方法检测访问线程的状态。通过利用这些状态，NAP 将交换机划分为三个阶段，并小心地将任务(例如，初始化新的 NAL、刷新和回收旧的 NAL)划分为不同的阶段。因此，在一个很小的时间间隔内，只有一小部分索引操作被阻塞。

实验结果表明，本文的 AutoTiering（自动分级技术）可以显著提高各种应用程序的性能。与基线 Linux 内核[15]相比，GraphMat 和 graph500 的性能分别提高了 2.3 倍和 6.9 倍。大多数 SPECAccel 工作负载都有 2 倍的加速。与 Intel 最近的方法相比，我们的性能提高了 3.5 倍。NAP 有几个好处。首先，它具有普遍性和有效性;我们使用 NAP 转换了五个最先进的并发 PM 索引，通过 NAP 转换的对应索引显著提高了一台四节点机器的吞吐量。其次，由于热门项的集合总是很小，NAP 所导致的额外内存消耗和恢复时间是有限的。我们对一台运行 72 个线程的四节点机器的评估表明，当在 NAL 中维护 100K 热项目时，NAP 使用不到 70MB 的额外 DRAM/PM 空间，恢复时间不到 1 秒。

3 研究进展

3.1 性能特性

本文描述了与 Linux 操作系统一起运行的软件管理的分层内存系统的显著性能特征。图 1 是本研究中使用的系统组织。系统中有两个 CPU 插座。每个 CPU 插槽连接 1 个 DRAM 节点和 1 个 DCPMM 节点。整个物理地址空间由 DRAM 和 DCPMM 节点组成。



(a) Multi-tiered memory (b) DRAM-only memory

图 2:多层和 DRAM-only 的内存系统的内存访问延迟和带宽

在多层内存系统中，影响性能的关键因素不仅是访问位置，还有内存的访问层。图 2a 显示了从 MLC[8]测量的四个内存节点的读访问延迟和带宽。访问本地 DRAM 的性能优于其他三个内存节点，这些节点在传统的 NUMA 架构中建立得很好。另一方面，我们观察到，由于设备特性，本地 DCPMM (L-DCPMM)比远程 DRAM (R-DRAM)慢。这与认为本地内存总是比远程内存快的传统观点形成了鲜明的对比。注意，我们在带宽测量中也观察到类似的模式。

类似地，图 2b 显示了在四个 CPU (Intel Xeon Gold 6242)插座上使用只使用 DRAM-only 系统的相同类型的评估。在访问任何远程 DRAM 节点的延迟和带宽方面没有显著差异。因此，在仅使用 DRAM 系统的远程 DRAM 节点上放置页面是一项相对简单的任务。

这些鲜明的特点促使我们探索操作系统中页面管理的设计空间。操作系统需要能够通过理解多层内存系统的性能特征来高效、动态地(重新)定位内存。与仅使用 DRAM 的系统不同，由于访问层的关系，并不是所有的远程内存节点都可以被认为是相等的。其次，Linux 不支持将页面从上层内存降级(或回收)到底层内存。在本研究中，我们通过考虑跨访问层和访问局部性的性能特征来重新讨论页面放置策略。

在 Linux 中，默认的页面分配策略尝试尽可能多地使用本地内存，以最小化访问远程内存所带来的性能损失。只有在本地内存中没有空闲空间时，内存分配器才会在称为 a fallback path（回退路径）的远程内存节点上寻找空闲空间。

因此，默认的(local-first)分配策略在多层内存系统中被认为是有害的。图 1a 中的数字表示线程在 CPU-0 上运行时的默认回退路径所使用的顺序，仅考虑物理距离。如果本地 DRAM 节点没有足够的空闲空间，内存分配器将检查回退路径，以确定应该发送分配请求的内存节点。我们预期分配器应该请求远程 DRAM 节点获得一个空闲页面，因为这个节点提供了比本地 DCPMM 节点更好的性能。然而，令人惊讶的是，最先进的 Linux 内核中的回退路径指示（indicate)本地 DCPMM(较低层)。它没有考虑到在多层存储系统中，内存类型对性能比访问位置更敏感的显著特征。

Linux 中包含了 AutoNUMA 功能，它可以自动将页面迁移到更接近在 runtime 中运行的线程的内存节点。操作系统检查访问位置，以确定被访问的页面是放在本地内存还是远程内存上。如果这是在远程内存上，则将该页迁移到本地内存，以避免对后续请求进行远程访问。这种方法提高了在 DRAM-only 的 NUMA 系统上运行的应用程序的性能。

然而，我们发现目前的设计并没有利用多层内存层次结构的优势。图 3 显示了 128GB 数据集的 graph500 内存节点的内存使用情况。首先，我们注意到上层内存没有得到有效使用，因为更频繁访问的页面(暗红色)主要驻留在下层内存(node-2)。相反，访问频率较低的页面则放在上层内存 1 中。造成这种情况的主要原因是，当没有空闲空间时，当前内存管理不允许将页面提升或迁移到上层内存。虽然这样的设计决策对于只使用 DRAM 的系统是合理的，但是对于多层内存系统，我们需要重新考虑这个假设。

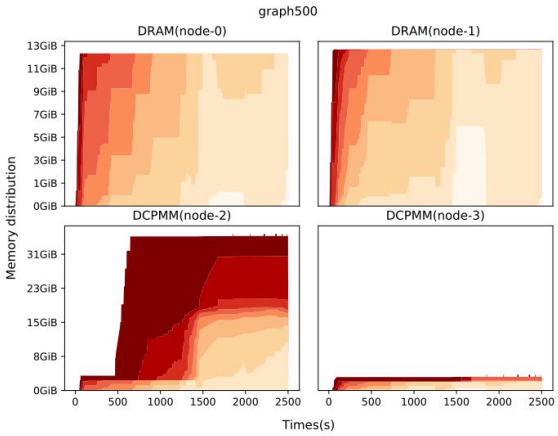


图 3:graph500 在内存节点上的页面分布和访问强度(颜色较深的表示访问相对频繁的页面)

当前的页面回收也专为基于存储的交换设备支持的 DRAM-only 系统而设计，而不是为分层内存系统。传统上，当内存节点耗尽时，kswapd 将内存中的非活动页面直接回收到存储中，而不考虑内存层。将底层内存中的页面回收给存储设备是有意义的。然而，当较低层有足够的空间时，这对于上层内存页不是理想的解决方案。

拥有众多 CPU 内核和 OptaneDIMM 的 NUMA 机器应该是快速和大容量存储的理想架构;然而，这是不正确的，因为远程 PM 访问速度很慢(例如，在远程 NUMA 节点上访问 PM)。

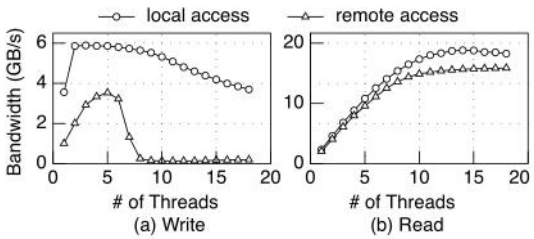


图 4:带有不同线程的三个 128GB Optane DIMM 的带宽。**本**

地访问:线程访问本地的 Optane DIMM;**远程访问:**线程访问安装在其他 NUMA 节点上的 Optane DIMM。我们使用 ntstore 指令来写 PM。

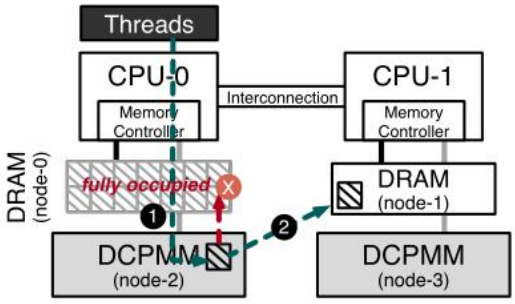
图 4 报告了 Optane DIMM 的本地/远程带宽(每个 NUMA 节点有 3 个 Optane DIMM 和 18 个 CPU 核)。每个线程对 2GB PM 空间执行顺序访问。我们使用 32 字节的非时态存储(ntstore)来进行 PM 写入。远程访问的写峰值带宽(3.5GB/s)仅为本地访问(5.9GB/s)的 59%。更糟糕的是，在超过 8 个并发线程的情况下，远程写的带宽会崩溃(< 250MB/s)。对于读操作来说，尽管 Optane DIMM 本地带宽和远程带宽之间的差距相对较小(16.9%)，但节点间链路(即 UPI)导致的额外访问延迟相当大(~ 100ns)，加剧了本已很高的 PM 读延迟(~ 300ns)。根据这些观察，我们得出结论，高性能的 PM 系统应该避免访问远程 PM，特别是对于写操作。

我们将远程 PM 写入的低性能归因于两个原因。首先，nstore 指令的行为可能类似于 cache line 的 read-modify-write 指令，这会减少可用的 PM 带宽。其次，由于 read-modify-write 行为，远程写可能会触发多套接字缓存一致性流量，从而导致额外的 PM 写。

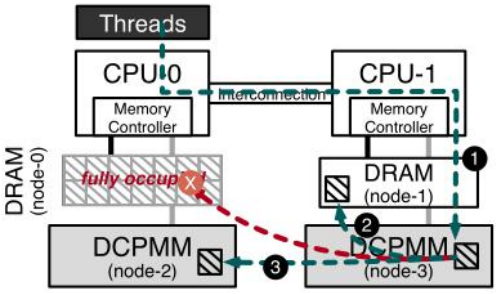
3.2 关键思想

3.2.1 AutoTiering - CPM

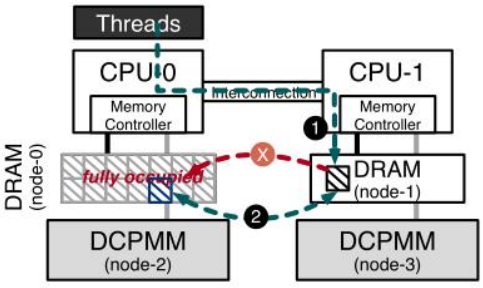
当前的 AutoNUMA 只在上层(本地 DRAM)内存有空闲空间时才处理提升和迁移请求。否则，请求将被丢弃，出错的页面将保留在原始内存中。当本地 DRAM 被完全占用时，我们建议的设计允许将页面提升或迁移到多层内存层次结构中的下一个最佳内存节点。这种方法可以利用多层内存层次结构的优势，提供比普通 Linux 内核更高的性能。



(a) Page promotion
(LDCPMM → RDRAM)



(b) Page promotion or migration
(RDCPMM → RDRAM or LDCPMM)



(c) Page exchange between DRAM nodes

图 5:我们的保守设计:利用多层内存层次结构(L:本地，R:远程)

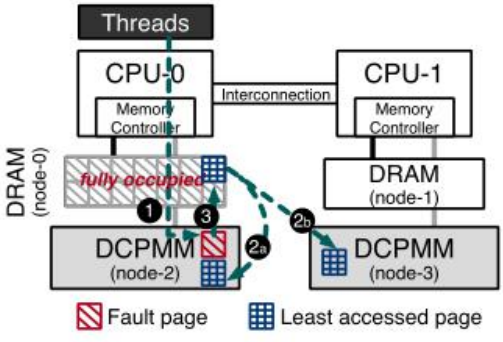
图 5 描述了当本地 DRAM 满时我们的反应。在多层内存系统中，页面迁移或提升到本地 DRAM 的需求有三个来源。多层结构为设计内存布局提供了新的机会。首先,(5a)faulted 页面驻留在本地 DCPMM(1),我们页面作为第二好的位置提升到远程 DRAM(2)。因为远程 DRAM 提供了比本地 DCPMM 更低的延迟和更高的带宽,我们可以提高内存最初分配在较低层的应用程序的性能。(5b)如果故障页面位于远程 DCPMM(1)中，那么我们有两种方法来利用多层内存层次结构的优势。我们尝试将页面提升到远程 DRAM(2)。如果远程 DRAM 也没有空闲空间，我们尝试将页面迁移到本地 DCPMM(3)。与普通 Linux 内核不同，我们修改后的内核支持将页面迁移到无 CPU 节点(本地 DCPMM)。我们称之为自动分级保守提升和迁移(AutoTiering Conservative Promotion and Migration，CPM)。最后，(5c)故障页面位于远程 DRAM(1)中。这意味着该页面已经位于第二好的位置。现有的 AutoNUMA 无法完成上层内存节点之间的页面迁移操作。结果，它导致了次优性能。在这种情况下，我们考虑页面交换选项来满足内存亲和性的需求。

由于这种设计不需要对现有的 Linux 操作系统进行重大更改，因此很容易将其集成到 AutoNUMA 设施之上。我们期望我们的保守设计能够成为这种软件管理的分层存储

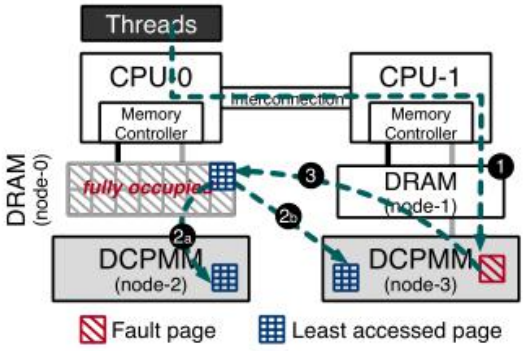
系统的实用解决方案。

3.2.2 AutoTiering - OPM

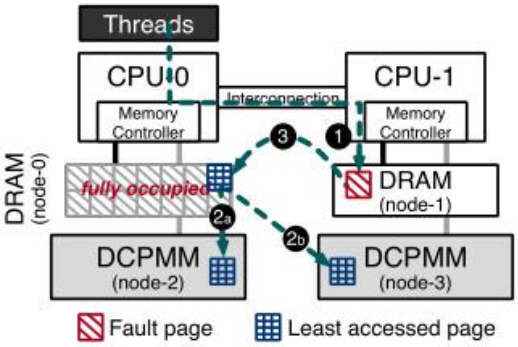
在保守设计中，经常使用的页面可以驻留在较低层(DCPMM)内存中，而上层(DRAM)内存保存不经常访问的数据。为了缓解这种不受欢迎的内存放置，我们探索了一种渐进的策略，机会地从上层内存降级一个页面来创建空闲空间。这是保守设计和进步设计的主要区别。通过降级页面，页面提升的请求可以成功。为了有效地进行页面降级，我们需要能够选择在短时间内不太可能重用的页面。否则，错误的选择可能会对性能产生负面影响。



(a) Eviction & Promotion
(LDCPMM → LDRAM)



(b) Eviction & Promotion
(RDCPMM → LDRAM)



(c) Eviction & Migration
(RDRAM → LDRAM)

图 6 描述了我们的渐进式方法如何处理页面降级。

当发生 NUMA 页面故障(1)时，我们从上层(本地 DRAM)内存中找到访问次数最少的页面，并将访问次数最少的页面与出现故障的页面的访问频率进行比较。如果所选页面的访问频率相对低于出错页面，那么我们将所选页面降级，将出错页面放在更高层的内存节点上。否则，我们将阻止页面提升或迁移请求，以保留更频繁访问的页面的上层内存。

为了从本地 DCPMM (6a)或远程 DCPMM (6b)提升页面，我们将选择的访问最少的页面降级到较低层内存(2a 或 2b)。降级页面的目的地取决于之前访问页面的位置，以保留位置。之后，(3)我们最终可以将页面提升到本地 DRAM 节点。此外，图 6c 显示了如何从远程 DRAM 发出页面迁移请求。我们称之为自动分层机会提升和迁移(AutoTiering Opportunistic Promotion and Migration ， OPM)。

我们进一步优化了渐进式设计，将降级和升级（或迁移）融合到一个 exchange 操作中，称为 AutoTiering OPM with Exchange (OPMX)。当降级的目的地等于升级或迁移的源时，我们利用 exchange 操作，而不是单个升级和迁移。例如（6a），如果我们需要将所选页面降级到本地 DCPMM 并将该页面升级到本地 DRAM，则会融合两个单独的操作。交换操作消除了不必要的页面分配和释放操作。

3.2.3 隐藏页面降级延迟

我们保留了一个包含几个保留页面的页面池。我们根据经验确定 4KB 和 2MB 的预留页面分别为 16 和 4。保留的页面允许我们立即服务于提升请求，而不需要降级过程，即使上层内存已满。这种方法比页面交换方案更具成本效益，因为它隐藏了关键路径中页面降级的延迟。将页面降级到较低级别内存所花费的时间要比将页面提升到较高级别内存所花费的时间长，因为用于较低级别内存的存储类内存提供的读性能优于写性能。为了在后台有效地降级页面，我们维

护一个名为 kdemoted 的新内核线程，以批处理方式降级访问次数最少的页面。一旦保留的页面数量低于某个阈值，我们就唤醒内核线程以启动降级进程。通过敏感性研究阈值应设置为 4。

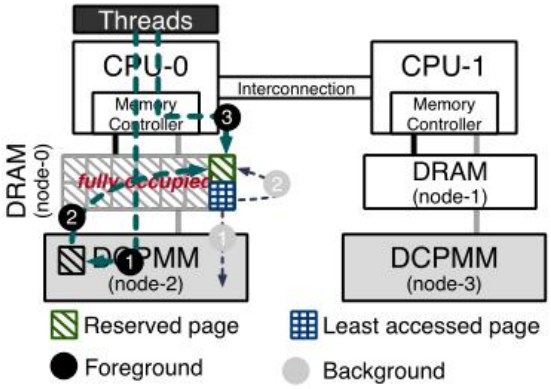


图 8:用我们的 kdemoted 来隐藏页面降级的延迟

图 8 描述了简单的优化如何隐藏页面降级的延迟。对于每个提升请求(1)，即使上层内存已满(2)，页面也可以被提升。完成提升后，返回 NUMA 错误处理程序，而不进行降级进程，并且未来访问页面(3)将发生在上层内存上。与此同时，kdemoted 会根据需要在批处理(1)中降级访问最少的页面，以回收内存池(2)。我们称之为 OPM-BD(Background demotion)。

3.2.4 NAP 概述

NAP (NUMA-Aware Persistent Memory Indexes)，一种将并发 PM 索引转换为 NUMA-aware 索引的方法。图 9 展示了 NAP 的体系结构和交互。NAP 由两个主要组件组成:原始 PM 索引和 NUMA-aware 层。

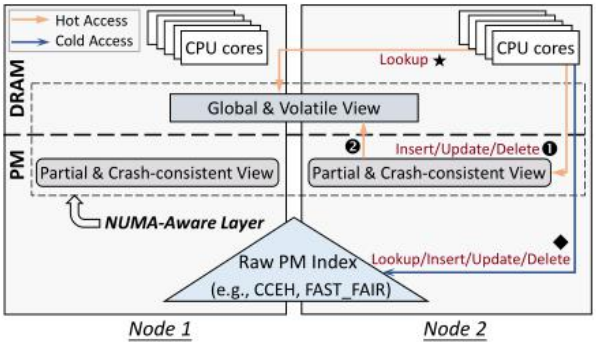


图 9:NAP 的架构和交互。

•原始 PM 指数。原始 PM 索引可以是任意现有的并发 PM 索引(例如，CCEH[13]， FAST_FAIR[11])，而不考虑其并发控制机制(基于锁或无锁)和结构(基于树、基于散列表或混合)。原始 PM 索引跨越多个 NUMA 节点;它管理冷项(图 9 中的♦)，它在总数据集中占非常大的比例。

•NUMA-aware 层(NAL)。NAP 引导访问 NAL 的热门项目，NAL 包含两部分:全局和不稳定视图(global & volatile 即 GV-view)和每个节点的局部和崩溃一致视图(partial & crash-consistent 即 PC-view)。GV-view 驻留在 DRAM 中，并维护热点项的最新值来服务查找请求(图 4 中的★)。每个节点的 PC-view 驻留在 PM 中。当线程向一个热项发出插入/更新/删除操作时，同一 NUMA 节点中的 PC-view 吸收该操作，并以崩溃一致的方式保持该操作的效果(①)。然后，更新 GV-view 中相应的值(②)，以确保 GV-view 始终拥有最新的热门项值。为了消除远程 PM 访问和避免额外的本地 PM 访问，我们不同步不同 PC-view 之间的状态，因此每个 PC-view 只有热点项的部分最新值。当热点发生转移时，NAP 可以及时识别新的一组热点项目，并切换到 NAL 的新版本;同时，旧 NAL 中的热门项目被刷新到基础的原始 PM 索引。

3.2.5 全局和易变视图（GV-view）

设计目标。除了为热点项提供查找服务，DRAM-resident GV-view 还负责 1)控制对 NAL 的并发访问，2)检查一个项目是否属于热点集（为了简化表述，我们将热门项的集合称为热集。这里假设热集的内容是事先已知的）。因此，GV-view 的设计必须是轻量级和高效的。

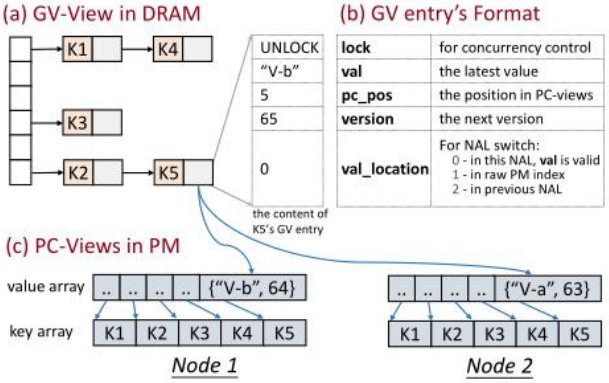


图 10:GV-view 和 PC-view 的结构。

设计细节。NAP 将 GV-view 组织为一个 DRAM-resident 索引，该索引为每个热点项目维护从键到 GV 条目的映射。图 10(a)显示了 GV-view 的结构。GV-view 默认使用哈希表;但如果原始 PM 索引支持范围查询，它使用基于树的数据结构。由于热集是固定的，除非 NAL 被切换(例如，热集在图 10 中是{K1, K2, K3, K4, K5})， GV-view 的索引完全是在 NAL 的初始化期间构建的，此后不会对其结构做任何更改。因此，任何线程不安全的高性能索引都是适用的(例如，c++ unordered_map)。

对于每个热门项目，关联的 GV 条目维护其运行时信息。图 10(b)显示了 GV 条目的格式，它由 5 个字段组成:1)一个读写锁来控制对热点条目的并发访问;2)条目的最新值;3)一个指针，指示在 PC-view 中持久化项目的位置。4)本项目的版本，用于 PC-view 的可恢复性;5)一个辅助 NAL 开关的枚举值。

查找操作。在没有 NAL 开关的情况下，执行如下查找操作:访问线程检查 GV-view 中目标项目;如果目标项不存在，则将查找重定向到原始 PM 索引;否则，线程在相应的 GV 条目中获取读锁，复制该值，最后释放锁。

范围查询操作。NAP 使范围查询复杂化，因为目标范围的项可能同时存在于 GV-view 和原始 PM 索引中。一个访问线程执行一个范围查询如下:它搜索 GV-view，得到目标范围内的项目(S1);然后，它通过调用原始 PM 索引的范围查询接口来获得 S2;最后，它合并 S1 和 S2(如果一个项在 S1 和 S2 中都存在，我们就留下 S1 中的项)，返回结果。就像 FAST_FAIR[10]和 P-Masstree[12]一样，NAP 中的范围查询操作不是并发插入/更新/删除操作;如果 NAP 之上的系统(如数据库)需要更高的隔离级别(如可重复读取)，则需要实现 next-key 锁定或版本机制。

3.2.6 部分和崩溃一致性视图（PC-view）

设计目标。每个节点的 PM-resident PC-views 吸收更新/插入/删除操作，并确保这些操作的效果能够在断电时保持不变。PC-views 有两个设计目标:1)可恢复性。PC-view 之间的状态不一致，因此 NAP 必须能够在恢复时识别最新的值。2)低开销故障原子性。为了保证故障原子性，我们必须显式地使用 flush 指令(例如 clflush、clwb 和 clflushopt)持久化数据，并避免使用 fence 指令(例如 sfence)进行存储重排序。减少这些昂贵指令的使用是实现高性能的关键。

设计细节。NAP 将每个节点 PC-view 组织成两个 PM-resident 数组:一个只读键数组和一个可写值数组(图 10(c))。键数组存储热集的所有键。值数组为每个热项保留一个 PC 条目来记录值。一个热项的 PC 表项通过对应的 GV 表项的 PC_pos 字段指定;例如，在图 10 中，每个 PC-view 中的第 5 个 PC 条目属于 K5。请注意，每个 PC 条目都包含一个指向键数组中相关键的指针，以使 NAL 可恢复。

因为两个线程可能更新相同的热点项，但操作不同的 PC-view，所以热点项的值在 PC-view 之间是不一致的。为了在恢复时识别最新的值，我们采用了一种简单的基于版本的机制。每个热点项都有一个单调递增的 64 位版本，记录在 GV-view 中(图 10 中的版本字段)。一个版本的最重要位是删除标记。

插入/更新操作。如果没有 NAL 交换机，插入/更新操作如下:

1)访问线程在 GV-view 中搜索目标条目;如果目标项不属于该热集，则操作被重定向到原始 PM 索引。

2)线程在 GV-view 中获得目标对象的写锁，然后获得一个新的版本。

3)线程将带有新值(即(value, version) pair)的版本自动保存到本地 NUMA 节点的目标 PC 条目中。

4)线程更新 GV-view 中的易变值(用于以后的查找操作)，最后释放锁。

删除操作。删除操作的过程与插入/更新操作相同，除了上面的步骤 3):访问线程设置所获得版本的删除标记，并将其持久化到本地 PC-view 中。

使用基于版本的机制，我们可以准确地从多个 PC -view 中识别一个热点项的最新值:具有最大版本的值(没有删除标记)是最新的;如果设置了最大版本的删除标记，则对应的热项已被删除。例如，在图 10(c)中，在最大版本下，节点 1 PC-view 中的“V-b”是 K5 的最新值。

现在我们描述如何以低开销保证更新(value、version)pair 的失败原子性。NAP 采用了两种不同的机制，分别有效地支持可变长度值和固定的 8 字节值。

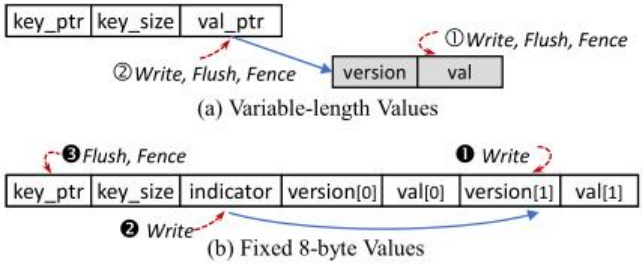


图 11:两种类型的 PC 条目结构。Key_ptr 指向键数组中对应的键，key_size 表示该键的大小。(a)对于变长值，我们使用写时复制来处理失败原子性。每个 PC 条目是 24 字节。[version, val]的灰空间是从 PM 中分配的。(b)对于固定的 8 字节值，我们采用轻量级的双化身切换机制。每个 PC 条目是 49 字节(指示器是 1 字节，其他字段是 8 字节)，cache-line 对齐，包含（value，version）pair 的两个化身。

对于变长值，我们利用写时拷贝(CoW)来更新(value, version) pair;图 11(a)显示了相应的 PC 条目。访问线程首先分配空闲的 PM 空间并拷贝(value, version) pair 给它;然后，线程通过 clflushopt 指令和 sfence (①)来刷新 pair;最后，线程自动更新 8 字节的指针到（value, version）pair 的地址，通过 clwb 刷新指针，并发布 sfence 以确保持久性已完成(②)。对于（value, version）pair，我们使用 clflushopt(使缓存行失效)而不是 clwb(不执行缓存失效)，这样可以为其他操作节省 CPU 缓存空间;这是因为 PC-view 中的值只在恢复

期间读取。

NAP 为固定的 8 字节值设计了一种双化身切换机制，这在 PM 索引中非常常见(8 字节值通常是指示实际数据位置的指针)。图 11(b)显示了相应 PC 条目的结构，它是 49 字节和缓存行对齐的。有两种 8 字节值和 8 字节版本的化身，以及指向有效化身的指示器。当写入一个新的（value version） pair 时，访问线程首先将该 pair 复制到无效的化身中(①)，这可以根据指示器计算(例如，如果指示器指向第一个转化，那么第二个转化就是无效的)。然后，线程切换指示器(②)，让它指向更新后的化身。最后，该线程向 PC 条目发出一个 clwb，后面跟着一个 sfence(②)。与 CoW 相比，双化身切换机制节省了 flush 指令和 fence 指令，提高了效率。在切换指示器之前，我们不需要一个 fence，因为在英特尔 cpu 的 TSO(总存储顺序)架构下，对同一缓存线的写操作按程序顺序到达 PM。值得注意的是，虽然每个 PC 条目占用 64 字节的 PM 空间来强制缓存行对齐，但是 PM 的消耗是有限的;这是因为热集很小。

3.2.7 热集标识

设计目标。在实际工作负载中，热集会随着时间不断变化[21];因此，NAP 需要实时识别热集。识别热集的设计目标是:1)尽量减少对前台索引操作的干扰，2)在面对无限的索引操作流时减少内存占用。

设计细节。NAP 使用专用的开关线程进行热集标识，将该进程从索引操作的关键路径中分离出来。图 12 显示了交换线程如何与访问线程交互，并标识热集。

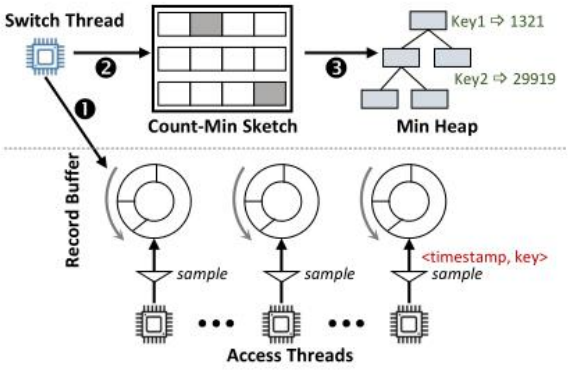


图 12:热集标识。访问线程通过采样将其访问模式发布到记录缓冲区中。开关线程使用一个 count-min 示意图来估计键的频率，并使用一个最小堆来维护当前的热集。

每个访问线程维护一个循环记录缓冲区，以发布其访问模式。为了减少热集识别造成的干扰，访问线程使用采样，并使写入记录缓冲区无协调。具体来说，每进行几次操作(例如，32)，一个访问线程就会向记录缓冲区写入一个（timestamp, key）对，其中 timestamp 是一个通过 rdtsc 指令生成的 64 位数字，key 是当前索引操作的 key。访问线程盲目地将（timestamp、key）对追加到循环缓冲区，而不管被覆盖的数据是否已经被切换线程消耗(即，没有与切换线程协调)。

在 count-min 缩略图和 min 堆的帮助下，switch 线程在以下重复的三个步骤中提取记录缓冲区。

1)开关线程以轮询方式选择一个记录缓冲区，并从中取一批(例如，8 个)新的（timestamp, key）对;这种批量取操作减少了缓存线的移动。两种类型的（timestamp, key）对被认为是无效的:i)时间戳小于从相应的记录缓冲区中读取的最大时间戳，表明我们接近记录缓冲区的尾部;因此，取操作停止。Ii)(当前时间-时间戳)大于阈值(例如，100ms)，说明该对太陈旧;因此，这对被跳过。注意，虽然 rdstc 生成的时间戳在 CPU 核之间并没有严格同步，但它并没有对 NAP 造成任何明显的影响。

2)对于从记录缓冲区中获取的每一个键，switch 线程利用一个 count-min 缩略图来更新和估计它的访问频率。count-min 结构是内存高效的，因为它只使用了几个小数组。访问线程使用的采样会过滤掉最不常用的键，避免缩略图溢出。

3)最小堆以（key, frequency）对的形式维护当前的热集，按 frequency 域排序。堆的大小有一个上限(例如，10,000)，这是可以配置的。对于从记录缓冲区中获取的键(我们称它为 K，并称其估计的频率为 F)，如果它已经在堆中，开关线程将相应的频率字段更新为 F;否则，开关线程将(K, F)对插入堆中。如果堆已满，并且 F 大于堆根的频率，则线程用(K, F)替换堆根中的对。每次修改堆时，我们都需要调整其结构以加强其排序属性。

开关线程周期性地(例如，每 1 秒)比较堆和当前 NAL 使用的热集。如果两者之间存在较大的差异，即不同键的比例超过 25%，则 switch thread 用新的 hot set(即堆中的键)触发 NAL 切换。所有的统计数据，包括 count-min 缩略图和最小堆，都会被定期清除。

处理均匀工作负载。NAP 将在均匀工作负载下由 NAL 引起的开销降至最低。具体来说，开关线程检测均匀的工作负载，在此负载下，它初始化一个空的 NAL(带有 0 大小的 GV-view)。对于索引操作，访问线程在搜索 GV-view 之前检查它的大小，这只会引起少于 5 个 CPU 周期。开关线程可以使用两个信号来标识均匀的工作负载:①堆中的条目接收不到所有访问的 10%;②堆中最热的项接收到与最冷的项相当的访问(即在 3 倍以内)。

4 总结与展望

这项工作探索了一组新的称为 AutoTiering 的页面管理方案，该方案受益于多层内存系统。我们发现 Linux 操作系统关注的是 NUMA 造成的访问局部性，而不是内存层。然而，在多层内存系统中，访问内存的成本并不仅仅与位置成比例。通过考虑两个因素:访问层和位置，我们全面解决了利用多层内存系统的不同方面。我们用一个真实世界的分层存储系统构建了一个概念证明。我们的评估显示，在各种基准测试中，与 Linux 内核的普通版本 5.3 和英特尔的 Tiering v0.6 先前的方法相比，性能有了显著提高。本文设计、实现和评估了 NAP，这是一种黑盒方法，可以将并发 PM 索引转换为 NUMA-aware 的对应索引。NAP 使用 NUMA-aware 层来吸收对热项目的访问，这消除了远程 PM 访问，而不会导致额外的本地 PM 访问。NAP 可以显著提高多节点机器上 PM 索引的性能。

未来的分层存储系统预计将更加多样化和异构。为了使我们的方法更一般化，我们可以维护一个表来描述位置的可能替代方案。在操作系统中初始化内存时，我们可以跨内存节点度量实际性能。有了这样一个新表，AutoTiering 可以自适应地调整页面需要提升、降级或迁移的位置，正如解释的那样，而不需要静态决策。

参考文献

[1] Jonghyeon Kim,Wonkyo Choe,and Jeongseob Ahn,Ajou University.Exploring the Design Space of Page Management for Multi-Tiered Memory Systems.In Proceeding of the 2021 USENIX Annual Technical Conference(ATC),July 2021.https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon.

[2] Qing Wang,Youyou Lu,Junru Li,and JiwuShu,Tsinghua University.Nap:A Black-Box Approach to NUMA-Aware Persistent Memory Indexes.In Proceeding of the 15th USENIX Symposium on Operating Systems Design and Implementation(OSDI).July 2021.https://www.usenix.org/conference/osdi21/presentation/wang-qing.

[3] Samsung unveils industry-first memory module incorporating new cxl interconnect standard, 2021. https://bit.ly/3uBo27J.

[4] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis:Transparently self-replicating page-tables for large-memory machines. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASP-LOS), 2020.

[5] Keith Busch. Page demotion for memory reclaim, Mar 2019. https://lwn.net/Articles/783672/.

[6] Wonkyo Choe, Jonghyeon Kim, and Jeongseob Ahn. A study of memory placement on hardware-assisted tiered memory systems. IEEE Computer Architecture Letters (CAL), 19(2), 2020.

[7] Linux Kernel Documentation. What is numa?, June 2019. Available at https://www.kernel.org/doc/Documentation/vm/numa.

[8] Intel. Intel memory latency checker v3.9,2020. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[9] Faisal Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey,Dhruva R. Chakrabarti, and M. Scott. Dalí: A Periodically Persistent Hash Map. In DISC, 2017.

[10] Lily Looi and Jianping Jane Xu. Intel optane data center persistent memory. In HotChips : A Symposium on High-Performance Chips (HotChips), 2019.

[11] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In 16th USENIX Conference on File and Storage Technologies (FAST 18),pages 187–200, Oakland, CA, February 2018. USENIX Association.

[12] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap,Taesoo Kim, and Vijay Chidambaram. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP'19, page 462–477,New York, NY, USA, 2019. Association for Computing Machinery.

[13] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H.Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In 17th USENIX Conference on File and Storage Technologies (FAST 19),pages 31–44, Boston, MA, February 2019. USENIX Association.

[14] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. DPTree: Differential Indexing for Persistent Memory. Proc. VLDB Endow., 13(4):421–434, December 2019.

[15] Dave Hansen. Allow persistent memory to be used like normal ram, Jan. 2019. https://lwn.net/Articles/776921/.

[16] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees:Optimizing Persistent Index Performance on 3DXPoint Memory. Proc. VLDB Endow., 13(7):1078–1090, March 2020.

[17] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang,and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. Proc. VLDB Endow., 13(12):2634–2648,July 2020.

[18] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu.ROART: Range-query Optimized Persistent ART. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 1–16. USENIX Association,February 2021.

[19] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable

Persistent Memory.In 18th USENIX Conference on File and Storage Technologies (FAST 20), pages 169–182, Santa Clara, CA,February 2020. USENIX Association.

[20] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'17, page 207–221, New York,NY, USA, 2017. Association for Computing Machinery.

[21] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu,Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In 18th USENIX Conference on File and Storage Technologies(FAST 20), pages 239–252, Santa Clara, CA, February 2020. USENIX Association.