分层内存中的虚拟内存辅助缓冲区管理

Yeasir Rayhan 和 Walid G. Aref 普渡大学，西拉法叶，IN，美国
{yrayhan,aref}@purdue.edu

arXiv:2603.03271v1 [cs.DB] 3 Mar 2026

摘要

近年来，分层内存架构在数据库社区中获得了广泛关注。在这些架构中，主机处理器的片上动态随机存取存储器通常称为本地内存，并构成主层。其他可按字节寻址且缓存一致性的内存资源统称为远程内存（简称 RMem），构成一个或多个次级层。RMem 的速度低于本地动态随机存取存储器，但快于磁盘，例如位于远程处理器插槽上的非统一内存访问内存、芯粒连接内存，以及通过高性能互连协议连接的内存，例如远程直接内存访问和 CXL。本文讨论传统的双层（动态随机存取存储器-磁盘）虚拟内存辅助缓冲区管理技术如何推广到 *n*层设置（DRAM-RMem-Disk）。我们提出*vmcache*^{n}，一种 *n*层虚拟内存辅助缓冲池，它利用虚拟内存子系统和操作系统调用在不同内存层级之间迁移页面。在此设置中，页面迁移可能成为瓶颈。为解决这一限制，我们引入 move_pages2 系统调用，为*vmcache*^{n}提供对页面迁移过程的细粒度控制。实验表明，对于 TPC-C工作负载，*vmcache*^{n} 的查询吞吐量最高可比*vmcache* 提升 4× 。

1 引言

大多数数据库管理系统采用两层（动态随机存取存储器-磁盘）缓冲池，将数据库页[1] 缓存到动态随机存取存储器中，以便高效访问。传统上，数据库管理系统使用哈希表，根据页标识符（即 PID）为缓存页建立索引。缓存页的 PID 映射到内存中对应页面的虚拟 内存地址。这种基于哈希表的设计引入了额外的间接寻址，因为每次访问页面都需要进行哈希表查找。即使工作负载完全驻留于动态随机存取存储器中且规模较大，这也会带来显著开销，因为哈希表膨胀、指针追逐、闩锁争用以及缓存局部性降低 [15, 39]会使工作负载受CPU限制。为减少基于哈希表的间接寻址开销，研究者提出了多种替代方案，例如指针调换 [12, 24, 26]，和虚拟内存辅助缓冲区管理 [22, 29], LIPAH [27, 28]。本文关注虚拟内存辅助缓冲池，即*vmcache*。我们探讨这种设计如何推广到传统两层设置之外，并扩展至 *n*层存储架构，其中前 n-1 层基于内存，例如动态随机存取存储器、持久内存和 CXL 内存，而最后一层基于磁盘，例如 NVMe固态硬盘和机械硬盘。

尽管DRAM价格持续下降，直到 2020 [9, 18]，近期市场动态仍导致动态随机存取存储器成本再次飙升，涨幅最高达90% [6]。因此，内存闲置，即各服务器上未充分利用的内存，已成为日益严峻的问题。近期研究，例如 [1, 3, 5, 13, 14, 16, 21, 25, 31, 33, 37]，主张采用内存解耦与多层内存架构，以提高内存利用率，并降低每有效吉字节（GB）内存的成本。尽管早期工作证明了分层内存架构在缓冲池设计中的可行性 [14, 29, 32, 38]，但在通用 *n*层级设置中，虚拟内存辅助缓冲区管理的设计空间在很大程度上仍未得到探索。鉴于其在两层级设计中取得成功，自然的下一步是理解其在 *n*层级设置中的行为与设计权衡。

本文介绍了*vmcache*^{n}，这是一种构建于*vmcache*之上的 *n* 层虚拟内存辅助缓冲池。*vmcache*^{n} 继承了*vmcache*的核心设计原则，并遵循以下不变量：与页关联的虚拟地址在其整个生命周期内保持固定。与*vmcache*类似，*vmcache*^{n} 将 PID 转换交由操作系统页表处理，并通过*libaio*接口 [8]，保留对页面从磁盘提升至内存层级以及页面驱逐的控制。增加额外的内存层级，也给*vmcache*^{n}带来了额外约束，从而扩展了先前的不变量：支撑内存中数据库页的物理页框必须支持随着时间推移在不同的内存驻留层级之间动态映射，同时确保该页对应的虚拟地址在其整个生命周期内保持固定。在本文余下部分，我们将重点讨论三层虚拟内存辅助缓冲池，其中第 0 层、第 1 层和第 2 层分别对应动态随机存取存储器、远程内存和 NVMe 磁盘。本文介绍的技术可以自然地推广至 *n*层架构，其中间层可能由多个远程内存层级构成。

vmcache^n 可在不同内存层级间控制页面提升和页面降级，并利用页面迁移系统调用（如 mbind 和 move_pages）在内存层级间迁移页面。这些系统调用会保留页面的虚拟地址，同时更新为其提供支持的物理页框。系统调用还会在操作系统页表中建立虚拟地址到物理地址的映射，因此无需额外进行哈希表查找，缓冲区管理器也能获知页面的物理位置。在具有快速内存层和 NVMe 固态硬盘的现代硬件上，操作系统内存管理单元（MMU）和页面迁移系统调用的开销可能成为性能瓶颈。因此，我们采用批处理来分摊迁移多个页面的成本。为进一步提升性能，我们基于原生内核系统调用 move_pages 实现了 move_pages2。自定义系统调用 move_pages2 使 *vmcache*^{n} 能够通过以下两个参数，对页面迁移过程进行细粒度控制：migration_mode 和 nr_max_batched_migration。migration_mode 参数让 *vmcache*^{n} 能够选择迁移策略的严格程度。nr_max_batched_migration 参数让 *vmcache*^{n} 能够选择可批量迁移的最大页面数。

[1]在本文其余部分中，除非另有说明，page 一词均指数据库页。

表 1：分层内存中缓冲区管理技术的概念性比较。

	mmap	传统方法。[14, 38]	指针置换。[32]	Hyrise [29]	*vmcache*^{n} 第 2 节
PID 转换	页面表	哈希表	侵入式	页面表	页面表
层级跟踪	页面表	哈希表	哈希表	页面表	页面表
控制	OS	DBMS	DBMS	DBMS	DBMS
迁移单元。	操作系统页	缓存行	缓存行	操作系统页	操作系统页
迁移粒度。	*n*	*n*	*n*	1	*n*
数据移动。	AutoNUMA	memcpy	memcpy	mbind	移动_页2*
虚拟地址	same	不同	不同	same	same
页面重复	no	yes	yes	no	no
可变大小	easy	hard	hard	easy	easy
图	yes	yes	no	yes	yes
实现	med	easy	hard	easy	easy

*vmcache^n 还支持 mbind 和移动_页面

2 虚拟内存辅助缓冲区管理在分层内存中

根据设计，在 *vmcache* [22], 中，与页面关联的虚拟地址在其整个生命周期内保持不变。页面可以从动态随机存取存储器中驱逐，之后再以相同的虚拟内存地址重新缓存。这种稳定性使 *vmcache* 能够安全地将 PID 转换委托给操作系统页表。如果页面驻留在动态随机存取存储器中，操作系统页表会将访问解析到相应的物理页框。否则，操作系统会引发缺页异常，而 *vmcache* 则会从磁盘加载该页面。这种设计消除了额外的哈希表查找开销，因为页面访问可通过直接的虚拟内存转换来解析。虽然这一设计在双层级场景中概念上很简单，但在涉及多个内存层级的 *n* 层级场景中，情况会更加复杂。

***vmcache^n*：一种 *n*层虚拟内存辅助缓冲池。** *vmcache^n* 将*vmcache* 的两层设计扩展到 *n*层设置。不同于仅维护单一动态随机存取存储器缓存的*vmcache*，*vmcache^n* 为每个内存层级维护独立的缓存。其关键区别在于，*vmcache^n*的 *n*层设计要求：在不改变虚拟地址的情况下，动态地将支撑页面的物理页框重新映射到多个常驻内存层级。接下来，我们概述*vmcache^n*的设计原则。

1. </style}层虚拟内存辅助缓冲池的基本不变量是稳定的虚拟寻址，以及可在各内存层级之间动态更改的物理页框映射。 **2.** 页面迁移只能更新支撑 PID 的物理页框，同时保留 PID 的虚拟内存地址。**3.** 在任意时间点，一个页面只能位于一个内存层级中。由于每个 PID 使用单一且固定的虚拟地址，页面不会在各层级之间复制。**4.** 由于稳定虚拟寻址这一不变量，只有配置为系统 RAM 模式的内存层级才与虚拟内存辅助缓冲区管理兼容 [3, 33]。

vmcache^n 与其他内存层配置不兼容，例如设备直接访问（DAX）模式 [17]。在 DAX 模式下，内存层以字符设备的形式暴露，从该内存层创建的虚拟内存映射由相应设备的物理页框永久支持。这会阻止将虚拟地址重新映射到不同的物理页框，即 *vmcache^n* 的基本不变量。

***vmcache^n* 与其他缓冲区管理技术的对比。** 表 1 对其他缓冲池设计与 *vmcache^n* 进行了概念性比较。*vmcache^n* 构建于 *vmcache* 之上。它继承了*vmcache* 的核心优势，包括支持可变大小的页面、图工作负载，以及在内存内和内存外工作负载下的稳健性能。在 *n*层设置中，出现了若干新的设计维度，包括跟踪页面的物理位置（tier-trk）、定义迁移单位（迁移单元）、确定迁移粒度（迁移粒度）、指定数据移动接口（数据移动），以及处理页面重复（page dup）。在传统设计 [14, 38] 和指针转换方法 [32], 中，哈希表显式跟踪每个页面在各内存层级中的物理位置。这些设计支持使用 memcpy 以缓存行粒度移动数据，从而导致跨层级的页面重复。这些设计可能在同一逻辑页面的生命周期内为其分配不同的虚拟地址。Hyrise 采用虚拟内存辅助缓冲池设计，依靠 mbind 逐页迁移页面，这在现代硬件上无法扩展。*vmcache^n* 对 *n* 页面执行批量迁移，并利用 move_pages2 实现高效的页面迁移。

2.1. 页表操作。 遵循*vmcache*，在启动时，*vmcache^n* 预留一块与第 *n*层级（即磁盘）大小相等的虚拟内存地址空间。该预留为每个数据库页在其整个生命周期内确立稳定的虚拟地址。要将某一页从磁盘缓存到内存驻留层级，*vmcache^n* 使用pread系统调用。然而，在 *n*层设置中，*vmcache^n* 必须支持比传统两层情形更灵活的页面放置；在传统情形中，页面仅被缓存到动态随机存取存储器。具体而言，*vmcache^n* 必须能够将页面从磁盘缓存到任意内存驻留层级，而不仅限于动态随机存取存储器。此外，*vmcache^n* 必须支持在各内存驻留层级之间透明地迁移页面，同时保留该页的虚拟地址。

将页面从磁盘添加到目标内存层级缓存。 遵循 [29], *vmcache^n* 使用mbind系统调用，将页添加到目标内存层级缓存。例如，我们希望将页P3缓存到层级-1。mbind将页P3的物理内存分配策略设置为MPOL_BIND。MPOL_BIND确保支撑P3的物理页框从tgt_tier位掩码所指定的内存层分配，在本例中即为0b0010（层级-1）。此外，*vmcache^n* 设置MPOL_MF_MOVE标志，以确保若该页已驻留于系统内存，则将其迁移到目标层级位掩码所指示的内存层。当mbind成功时，*vmcache^n* 通过pread从磁盘读取页P3到由虚拟内存地址映射的相应物理页框。随后，操作系统在操作系统页表中为P3安装虚拟地址到物理地址的映射。此后对P3的任何访问均由层级-1提供服务，直至P3被驱逐或被移动到不同的内存层。

> u64 偏移量 = 3* 页面大小；u64 目标_层级 = 0b0010；int 模式 =MPOL_BIND；int 标志 =MPOL_MF_MOVE；mbind（虚拟内存 + 偏移量，页面大小，模式，目标 _ 层级，8* sizeof（目标 _ 层级），标志）；pread（fd，虚拟内存 +偏移量，页面大小，偏移量）；

在内存层级缓存之间添加和移除页面。 在 *n* 层级结构中，页面可以在两个内存层级之间迁移。例如，当工作集超出动态随机存取存储器层的容量时，页面可以从动态随机存取存储器中驱逐，并卸载到

RMem 层，而不是磁盘。同样，页面也可以从 RMem 层预取回动态随机存取存储器层。中间 RMem 层之间也可以进行迁移，以降低页面的访问延迟。*vmcache*^{n} 在迁移期间维持以下不变量，即尽管支撑页面的物理页框可能发生变化，但其虚拟内存地址（即 *PID*）必须保持稳定。这一不变量排除了基于软件复制的页面迁移机制（例如 memcpy），因为这类机制会改变页面的虚拟地址。

vmcache^n 支持两个系统调用，即 mbind 和 move_pages，用于在内存层级之间迁移页面。这两个系统调用都维护 *vmcache*^{n} 的不变量。迁移完成后，系统调用会更新页表中相应的页表项，以反映更新后的物理页框。虽然 mbind 和 move_pages 都可以迁移页面，但二者的迁移粒度不同。move_pages 可以迁移多个页面，而 mbind 一次只能迁移一个页面。例如，假设我们希望将页面 P3 和页面 P6 迁移到第1层内存。*vmcache*^{n} 需要调用两次 mbind（即分别调用），才能迁移这些页面。

```
u64 offset1 = 3* pageSize; u64 offset2 = 6* pageSize; int
mode  = MPOL_BIND; int flags  = MPOL_MF_MOVE;
u64 tgt_tier  = 0b0010; for offsets in  [offset1, offset2]:
mbind（virtMem +offset, pageSize, mode, tgt_tier, 8*
sizeof（tgt_tier）, flags）;
```

相比之下，*vmcache*^{n} 仅调用一次 move_pages 系统调用，即可迁移多个页面（见下例）。第一个参数（=0）指调用进程，即 *vmcache*^{n}。第二个参数（=2）设置要迁移的页面数量，并定义后续三个数组参数的大小。页面数组包含待迁移页面的虚拟地址，即 {P3, P6}。tgt_tier 数组指定每个页面的目标内存层。状态数组在迁移完成后报告每个页面的迁移状态。最后，MPOL_MF_MOVE 标志请求将指定页面的物理页框迁移到目标内存层。

```
u64[]页面  ={ virtMem +3* pageSize , virtMem +6* pageSize };
整数 flag  = MPOL_MF_MOVE ;整数 []tgt_层级 ={1 , 1}; 整数 []
status ={-1, -1}; move_pages (0, 2, offset , tgt_层级 , status , flag);
```

2.2. 页面状态与同步。继承*vmcache*后，*vmcache*^{n} 使用 64 位表示一个页面的状态。最高的 8 位表示*vmcache*中的 4 种页面状态，即未锁定（0）、共享锁定（1-252）、已锁定（253）、已标记（254）和已驱逐（255）。为了适应 *n*层配置中的其他内存层级，*vmcache*^{n} 使用 $\lceil \log_2(n-1) \rceil$ 位编码支撑该页面的物理页框所在的位置，即内存层。以三层配置（DRAM-RMem-Disk）为例，使用 1 位编码内存层，因此总共需要 9 位来编码页面状态。由此产生层特定页面状态，例如 UnlockedDRAM、Unlocke-dRMem 等。与*vmcache*相比，*vmcache*^{n} 保留更少的位，即 $56-\log_2(n-1)$ 位，用于编码版本计数器。

状态转换。图1展示了三层配置下*vmcache*^{n} 页的状态转换图。与*vmcache*不同，*vmcache*^{n} 维护层特定的页面状态，以便在不同内存层级之间灵活迁移页面。接下来，[38], *vmcache*^{n} 维护四个迁移标志，以便在内存层级之间按概率迁移页面，即 Dr、Dw、Rr、Rw，后文将对此进行讨论。假设*vmcache*^{n} 需要访问处于已驱逐状态的页面P1。*vmcache*^{n} 调用 fix(P1)。在三层配置中，fix(P1)有两个可能的目标层级：动态随机存取存储器或 RMem，由 Rr 按概率选择。假设*vmcache*^{n} 选择动态随机存取存储器作为目标层级。P1 通过比较并交换操作（简称 CAS）转换到锁定状态。*vmcache*^{n} 从磁盘读取该页面，并使用 mbind 系统调用将其复制到动态随机存取存储器。mbind 会隐式更新操作系统页表。访问完成后，*vmcache*^{n} 调用 unfix(P1)，P1 随即转换到未锁定状态。当动态随机存取存储器缓存利用率达到预定义阈值（例如 95%）时，*vmcache*^{n} 会使用时钟置换算法主动标记未锁定页面，例如 P1。动态随机存取存储器中的已标记页面有两个目标层级，即 RMem 或磁盘，由 Rw 控制。如果 Rw 选择 RMem，*vmcache*^{n} 会使用 move_pages 系统调用，将 P1 从动态随机存取存储器降级迁移到 RMem。该系统调用会隐式更新操作系统页表中 P1 的驻留内存层级。页面P1在 RMem 中以未锁定状态重新开始。否则，P1 将被驱逐到磁盘。再考虑另一种情况：假设*vmcache*^{n} 需要访问驻留在 RMem 中的页面P2。*vmcache*^{n} 调用 fix(P2)。由于 RMem 具有字节可寻址性，*vmcache*^{n} 可以直接在原处访问 P2，也可以先将其迁移到动态随机存取存储器，再进行访问。该决策由

Rr. 假设 Rr 选择将 P2 提升到动态随机存取存储器。在这种情况下，*vmcache*^{n} 会从远程层预取一组未锁定页面，并将它们与 P2 捆绑在一起，通过批量迁移将所有选中的页面一并提升。

2.3. 页面置换。*vmcache*^{n} 为每个内存层级维护一个独立的缓存。当所有缓存都已满时，*vmcache*^{n} 需要从中驱逐页面。与*vmcache*一致，*vmcache*^{n} 为此使用时钟置换算法。与*vmcache*不同，在三层配置中，存在两 条不同的驱逐路径，即内存层级之间的驱逐，以及从内存层驱逐到磁盘。此外，引入额外的内存层级也增加了内存层级之间的页面提升路径。*vmcache*^{n} 使用一种简单的启发式方法，在三级层次结构中向上移动页面。

内存层级之间的批量驱逐。与*vmcache*不同，*vmcache*^{n} 为每个缓存维护独立的驻留集，即一个开放寻址哈希表，用于索引相应内存层级中的驻留页 PID。例如，在三层配置中，当动态随机存取存储器缓存大小达到阈值且目标层级为 RMem 时，*vmcache*^{n} 会启动以下驱逐流程：(1) 扫描动态随机存取存储器驻留集，收集选定驱逐的已标记页面，并锁定这些页面。(2) 使用 move_pages 更新操作系统页表中的页面位置。由于页面仍驻留在内存中，因此无需对脏页进行额外处理。(3) 从动态随机存取存储器驻留集中移除已锁定页面，并将其插入 RMem 驻留集。(4) 更新每个页面的物理层级位置元数据，并释放锁。

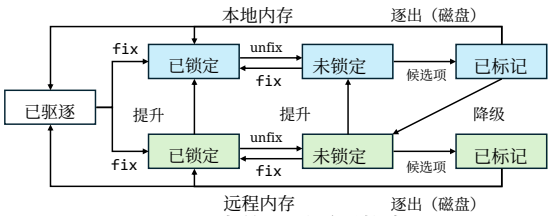


图1：*vmcache*^{n}中的三层页面状态。

内存层级之间的批量提升。在*vmcache*^{n}中，页面可能驻留在内存中，但位于 RMem 层。当页面查找需要访问 RMem 层中的页面时，*vmcache*^{n} 会以概率方式决定是否应提升该页面。如果触发提升，*vmcache*^{n} 会锁定目标页面，扫描相应的驻留集，并收集选定提升的未锁定页面。*vmcache*^{n} 会锁定这些页面，并执行批量驱逐流程的步骤 2–4。

3 move_pages2：跨内存层级的高效页面迁移

vmcache^n 依赖虚拟内存子系统和操作系统驱动的系统调用（例如 mbind [20] 和 move_pages [19],），以便在保留页的虚拟地址的同时动态改变其物理位置。在任何运行于 *n*层级设置中的虚拟内存辅助缓冲池中，页面迁移 都处于查询执行的关键路径上。因此，这些系统调用的性能对于实现高效的虚拟内存辅助缓冲池至关重要。尽管原生页面迁移系统调用通常已足够，但在某些工作负载下可能会遇到性能和可扩展性问题。为解决这一局限，我们提出了 move_pages2。

3.1. 动机。Linux 通过两个系统调用处理页面迁移：mbind [20] 和 move_pages [19]。我们重点关注 move_pages，因为它支持批量页面迁移。相比之下，mbind 逐页迁移，且无法扩展。

1. TLB 刷新。 move_pages 接受页列表作为输入，并分多个轮次迁移这些页面。在每个轮次中，内核会将最多 NR_MAX_BATCHED_MIGRATION 个、目标为同一内存层且连续的页面分组，进行批处理，然后将其加入页面列表队列。批量大小会直接影响性能。较大的批量可以减少 TLB 失效开销，并减少更新逐页状态的 store_status 辅助函数的调用次数。TLB 刷新开销较高，因为需要通过处理器间中断（IPI）使各 CPU 上的 TLB 条目失效。默认情况下，在 move_pages 中，NR_MAX_BATCHED_MIGRATION 设置为 512。默认的迁移模式为 MIGRATE_SYNC，每个轮次最多可能触发 NR_MAX_BATCHED_MIGRATION 次 TLB 失效，从而显著降低迁移性能。

2. 失败即中止策略。 在迁移过程中，move_pages 会调用内核函数 do_pages_move，其错误处理策略会显著影响性能。当内核遇到任意页面的用户空间访问失败、无效目标节点或权限错误时，会立即中止操作。在中止之前，它只会迁移当前批处理中已累积的页面，跳过输入页列表中剩余的所有页面，即使这些页面符合迁移条件也是如此。这种“失败即中止策略”在多线程环境中尤其不利，因为并发活动会增加页面已锁定、处于回写状态或暂时不可用的可能性。

3.2. 设计原则与实现细节。
移动_页面2。 move_pages2 不再采用原 move_pages 接口中的 pid 和标志参数，而是引入两个新参数 migrate_mode 和 nr_max_批处理_迁移，以便为 *vmcache*^{n} 提供对页面迁移的精细控制。默认情况下，move_pages2 将 pid = 0和标志 = MPOL_MF_MOVE 设为参数，从而赋予 *vmcache*^{n} 对迁移过程的独占控制权。所提出的自定义 move_pages2 实现的函数签名如下。

long move_页面2 (unsigned long count, void *页面[.count]，const int nodes[.count]，int status[.count],enum migrate_mode mode, int nr_max_批处理_迁移);

- count：要处理的页数。• pages：需要处理的页指针数组。• nodes：目标非均匀内存访问节点 ID 数组。• status：用于存储每个页面迁移状态的数组。• mode：迁移模式。
- nr_max_batched_migration：调用 TLB 刷新前可累积的最大页面数。

1. 减少 TLB 刷新。 move_pages2 引入参数 nr_max_batched_migration，用于调节单个批处理中迁移的页面数。此参数通过将 TLB 刷新的开销分摊至最多 nr_max_batched_migration 个页面，直接影响迁移性能。此外，move_pages2 通过 migration_mode 参数提供三种页面迁移模式。

- MIGRATE_异步会执行异步迁移，也就是说，即使当前页面迁移失败，也会继续处理下一个页面，因此采用的是非阻塞方式。• MIGRATE_同步会执行同步迁移，也就是说，内核会一直阻塞，直到当前页面迁移成功。
- MIGRATE_同步_LIGHT 不会因页面回写而阻塞，以缩短停顿时间。

2. 乐观故障处理。 move_pages2 采用乐观的错误处理策略，允许进行部分迁移。调用系统调用会产生跨越操作系统与 *vmcache*^{n} 边界的开销。move_pages2 会在每次调用中尽可能多地迁移页面，以摊销这部分开销，而不是因错误而过早终止。

3.3. 实现细节。 我们在以下四个内核函数中总计修改了约 150 行代码：do_pages_move、move_pages_and_store_status、do_move_pages_to_node，以及 migrate_pages，以实现 move_pages2。我们基于 Ubuntu 24.04LTS 上的 Linux 内核 6.8.0 实现了 move_pages2。内核磁盘映像可在 Cloud-Lab [10] 获取，并可通过此处[2]访问。

1. 迁移模式与批量处理。我们自定义实现的 migrate_pages（参见清单 1）使 *vmcache*^{n} 能够覆盖 Linux 默认的迁移模式，即 MIGRATE_SYNC。对于 MIGRATE_SYNC，内核遵循默认的阻塞路径（第 13 行）；否则，内核遵循非阻塞路径（第 15 行）。move_pages2 还使 *vmcache*^{n} 能够在迁移期间动态选择最大批量大小（第 6、9 行）。

```
1 int migrate_页面2 (. . ., enum migrate_mode mode , . . ., int nr_max_批处理_迁移 ){ 2 again: 3 nr_页面 = 0; // 遍历 "from" 中的页面 4 list_for_each_entry_safe (folio , folio2 , from , 最近最少使用链表) { 5 nr_页面 += folio_nr_页面(folio); 6 if (nr_页面 >= nr_max_批处理_迁移 ) break;
```

[2]urn:publicid:IDN+utah.cloudlab.us+镜像+pmoss-PG0:LINUX6.8.12_迁移

```
7 } 8 // 将“from”中的页面批处理为 folio 并进行迁移 9 if (nr_pages >= nr_max_batched_migration ) 10 list_cut_before (&folios , from , &folio2 -> lru); 11 else 12 list_splice_init (from , &folios); 13 if (migration_mode == MIGRATE_SYNC) 14 rc = migrate_pages_sync (. . ., MIGRATE_SYNC , . . .); 15 else 16 rc = migrate_pages_batch (. . ., mode , . . ., 17 nr_max_batched_migration ); 18 // 将未迁移的页面存入“ret_folios”，以便重试 19 list_splice_tail_init (&folios , &ret_folios); 20 if (rc < 0) { 21 rc_gather = rc; list_splice_tail (& split_folios , & ret_folios); goto out; 22 } 23 rc_gather += rc; 24 if (! list_empty(from)) goto again; // 准备下一批 folio 25 out: 26 // 将当前迁移轮次中未迁移的页面移回 27 // “folios”，以便在下一迁移轮次中 28 // 视情况重试 29 list_splice (& ret_folios , from); 30 if (list_empty(from)) rc_gather = 0; 31 return rc_gather; 32 }
```

清单 1：migrate_pages2 的伪代码。

2. 乐观错误处理。 我们对内核函数 do_pages_move（参见清单 2）的自定义实现可确保：如果内核遇到用户空间访问失败、无效的目标节点，或特定页面的权限问题，不会立即中止。相反，它会在状态数组中记录错误，并调用自定义内核函数 move_pages_store_status2，迁移当前轮次中已累积的页面。完成批处理后，内核会启动新一轮迁移，并继续处理页列表中剩余的页面（第 26–31 行）。move_pages2 在迁移过程中遇到错误时，也会采用相同的处理流程（第 19–25 行）。

```
1 static int do_pages_move2 (. . .,enum migrate_mode mode ,int nr_max_batched_migration ){ 2 for (i=start =0; i<nr_pages; i++) { 3 if (复制页面[ i ]或节点[i ]时出错) 4 goto handle_error; 5 if (处理目标层级时出错) 6 goto handle_error; 7 if (current_node == NUMA_NO_NODE) { // 开始迁移。 8 current_node = node; start =i; 9 } else if (node != current_node) { 10 // 结束当前迁移轮次并迁移 11 // 当前轮次收集的页面 12 err = move_pages_and_store_status2(. . ., mode , nr_max_batched_migration ); 13 if (err) goto migrate_error; 14 start =i; current_node = node; 15 } 16 // 将页 p 加入当前迁移轮次的队列 17 err = add_page_for_migration ();if (err > 0) continue; 18 err = store_status (); 19 migrate_error: 20 if (i + 1 == nr_pages || err < 0) { 21 err1 = move_pages_and_store_status2 (); // 迁移 22 if (err >= 0) err = err1; 23 current_node = NUMA_NO_NODE; // 新的迁移轮次 24 } 25 continue; // 继续迁移 26 handle_error: 27 err1 = store_status(status , i, err , 1); 28 if (err1) err = err1; 29 err1 = move_pages_and_store_status2 (); // 迁移 30 if (err >= 0) err = err1; 31 current_node = NUMA_NO_NODE; // 新一轮@
```

```
32 } 33 out_flush: 34 if (current_node != NUMA_NO_NODE) { 35 err1 = move_pages_and_store_status2(. . ., mode , nr_max_batched_migration ); 36 if (err >= 0) 37 err = err1; 38 } 39 out: 40 return err; 41 }
```

清单 2：do_pages_move2 的伪代码。

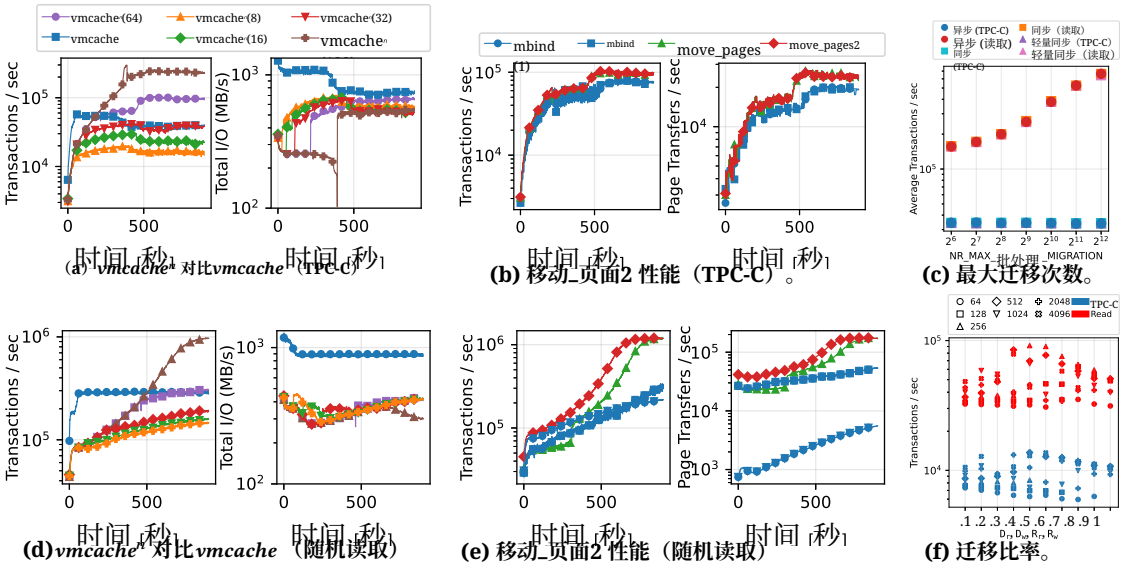
4 性能评估

所有实验均在 CloudLab [10]sm220u 节点上运行，该节点配备英特尔® 至强® 银牌 4314 CPU 和 2 个非均匀内存访问插槽，每个插槽有 32 个逻辑线程。我们将非均匀内存访问插槽 0 的动态随机存取存储器视为本地内存（层级 1），将非均匀内存访问插槽 1 的动态随机存取存储器视为远程内存（层级 2），并将一块 960 GB 三星 PCIe4 NVMe 设备视为磁盘（层级 3）。所有实验均使用非均匀内存访问插槽 0 的 32 个线程。按照 [22]，我们使用相同的 TPC-C 工作负载和由随机点查找组成的键值工作负载。键值工作负载使用 8 字节的均匀分布键和 120 字节的值。TPC-C 和随机读工作负载的数据规模分别约为 190 GB 和 130 GB。

***vmcache^n* 与 *vmcache*。** 图 2a 和 2d 比较了 TPC-C 和随机读工作负载下 *vmcache^n* 与 *vmcache* 随时间变化的性能。本地内存容量固定为 32 GB，而远程内存容量从 8 GB 变化到 128 GB。最初，在时间 T_0，远程内存层为空。随着事务随时间执行，远程内存逐渐被填充。对于 TPC-C，当远程内存容量分别为本地内存大小的 2× 和 4× 时，*vmcache^n* 的吞吐量最高比 *vmcache* 高 1.67× 和 3.82× 。尽管 *vmcache^n* 减少了磁盘 I/O，但远程内存较小时，内存层级之间的流量会成为性能瓶颈。对于没有写入操作的随机读工作负载，性能提升并不显著。例如，当远程内存为本地内存大小的 4× 时，*vmcache^n* 的事务吞吐量仅提高 1.36× 。在这种情况下，由于工作集呈现随机访问模式，内存层级之间的页面传输主导了执行成本。

时间都花在哪里了？ 图 3 展示了使用不同页面迁移系统调用时，*vmcache^n* 在 TPC-C 工作负载（图 3a）和随机读工作负载（图 3b）下的执行时间分解。磁盘 I/O 在 TPC-C 工作负载中占主导地位，而内存层级之间的页面传输则在随机读工作负载中占主导地位。对于随机读工作负载，使用 mbind 会产生大量开销，占总执行时间的 64.8%，因为 mbind 不支持批量页面移动。对于随机读工作负载，即使是 move_pages 和 move_pages2，也分别产生了 48.5% 和 48.4% 的显著开销。这些结果凸显了在 *n*层内存层级中，为虚拟内存辅助缓冲池提供高效页面移动接口至关重要。在这些实验中，本地内存和远程内存容量分别设为 32 GB 和 64 GB。内存层级之间的迁移比例和批量驱逐大小分别设为 1 和 512。

move_页面2的性能评估。图2b和图2e展示了在*vmcache^n* 中，move_pages2与基线页面迁移系统调用（即 mbind 和 move_pages）的性能对比。mbind(1) 将内存层级间的批量驱逐大小设为 1。其他情况下，



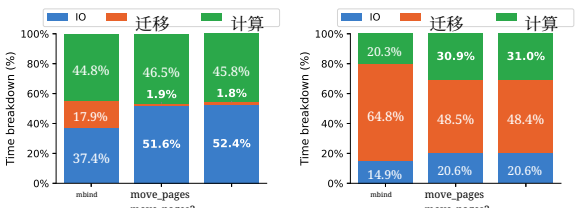
(a) *vmcache*^{n} 对比*vmcache*（TPC-C）。 (b) 移动_页面2 性能（TPC-C）。 (c) 最大迁移次数。

(d) *vmcache*^{n} 对比*vmcache*（随机读取） (e) 移动_页面2 性能（随机读取） (f) 迁移比率。

图 2：*vmcache*^{n} 与*vmcache*。

批量驱逐大小设为 512。对于 TPC-C 工作负载，move_pages2 的性能与 move_pages 相近（参见图2b）。然而，对于随机读工作负载，与 move_pages 相比，move_pages2 的查询吞吐量和页面迁移吞吐量分别提升了 1.42× 倍和 1.32× 倍（参见图2e）。这是因为，与基线方案相比，move_pages2 在单轮中能够批量处理的内存页数量更多，从而摊销了 TLB 失效的开销。

对 move_pages2的消融分析。图 2c 展示了在不同的 |NR_MAX_BATCHED_MIGRATION| 取值和迁移模式下，move_pages2 的性能。|NR_MAX_BATCHED_MIGRATION| 决定单轮迁移的内存页最大数量。在 *vmcache*^{n} 中，该值设为内存层级之间的批量驱逐大小。对于随机读工作负载，提高此值可改善性能，因为较大的批次能够分摊迁移开销。对于以写操作为主的 TPC-C，增大批量大小无法带来类似的收益。无论采用哪种迁移模式，异步、同步和同步_NO_COPY 在 *vmcache*^{n} 中带来的性能收益都相近，因为页面在迁移前都会被锁定。因此，允许部分页面迁移并不能显著改善 *vmcache*^{n} 的性能。图 2f 展示了 *vmcache*^{n} 在不同迁移比率下的性能。页面迁移不频繁时，较大的 |NR_MAX_BATCHED_MIGRATION| 可以改善性能，但随着迁移频率增加，它无法带来同等程度的收益。



(a) TPC-C 工作负载。 (b) 随机读工作负载。

图 3：*vmcache*^{n} 的执行时间细分。

5 结论

我们介绍了 *vmcache*^{n}，一种 *n* 层虚拟内存辅助缓冲池，其 TPC-C 吞吐量最高可比 *vmcache* 高 4× 。主要发现总结如下。

1. 对于虚拟内存辅助缓冲池，当远程内存容量介于 1× 和 2× 倍 DRAM 容量之间时，远程内存投资存在明确的成本盈亏平衡点。在此阈值以下，与两层缓冲池相比，每美元 QPS 的净收益为负，因为内存层级之间的页面迁移开销超过了任何容量收益。超过盈亏平衡点后，每美元 QPS 的净收益将显著转为正值，并随规模扩大而增加；当远程内存容量分别为 2×和 4× 倍 DRAM 容量时，可达到每美元 +41 QPS 和 +87 QPS。

2. 在*n*层虚拟内存辅助缓冲池中，内存层级之间的页面迁移是瓶颈，尤其是在工作集大部分都能容纳于这些内存层级中时。move_pages2 通过在每个迁移轮次中进行大规模批量传输，摊销迁移成本，从而缓解了这一开销。然而，页面迁移开销依然存在。在我们的实验中，执行时间中归因于内核态与用户态转换以及迁移准备开销的部分不到 0.005% [35]。因此，围绕微内核重新构建系统，以便在用户空间管理页面和页表，不太可能解决主要瓶颈 [23]。相反，更高效的页面迁移实现可能带来更大收益，无论是在用户空间 [30]还是内核空间 [2, 4, 7, 11, 34, 36]。

3. 由于稳定虚拟地址与交替的物理内存层级这一不变量，*n*层虚拟内存辅助缓冲池只有在远程内存层级以 System-DRAM 模式暴露给主机服务器时，才能纳入这些远程内存层级。通过 DAX 或 App Direct 模式分配的内存不能用于 *n*层虚拟内存辅助缓冲池，这限制了虚拟内存辅助缓冲池在该场景中的适用性。

[3]假设一条 Micron DDR4 3200MHz ECC RDIMM 内存的价格为 175 美元

参考文献

[1] Minseon Ahn、Andrew Chang、Donghun Lee、Jongmin Gim、Jungmin Kim、Jaemin Jung、Oliver Rebholz、Vincent Pham、Krishna T. Malladi与Yang-Seok Ki。2022。为内存数据库管理系统实现CXL内存扩展。载于DaMoN。ACM，8:1–8:5。[2] Nadav Amit。2017。利用页面访问跟踪优化TLB击落算法。载于USENIX。27–39。[3] Ramesh Aravind与Groves John。2024。基于FamFS的CXL内存共享及其用例研究。载于HiPC Workshops。IEEE，73–77。doi:10.1109/ HIPCW63042.2024.00022[4] Amro Awad、Arkaprava Basu、Sergey Blagodurov、Yan Solihin与Gabriel H. Loh。2017。通过自失效TLB表项避免TLB 刷新。载于PACT。273–287。[5] Yannis Chronis、Anastasia Ailamaki、Lawrence Benson、Helena Caminal、Jana Giceva、Dave Patterson、Eric Sedlar与Lisa Wu Wills。2025。以内存为中心的计算时代的数据库。载于CIDR。[6] Counterpoint Research。2026 4 2025 90。内存价格较Q上涨高达%。https://counterpointresearch.com/en/insights/Memory-Prices-Surge-Up-to-90-From-Q4-2025。访问于2026年2月11日。[7] Guilherme Cox与Abhishek Bhattacharjee。2017。面向多种页面大小架构的高效地址转换。载于ASPLOS。435–448。[8] Daniel Ehrenberg。2026。异步输入/输出（AIO）接口。https: https://github.com/littledan/linux-aio。访问于2026年2月11日。[9] Daniel Ehrenberg。2026。动态随机存取存储器每吉字节价格趋势。https://wiki.aiimpacts.org/ai_timelines/hardware_and_ai_timelines/trends_in_ dram_price_per_gigabyte。访问于2026年2月11日。[10] Dmitry Duplyakin、Robert Ricci、Aleksander Maricq、Gary Wong、Jonathon Duerig、Eric Eide、Leigh Stoller、Mike Hibler、David Johnson、Kirk Webb、Aditya Akella、Kuang-Ching Wang、Glenn Ricart、Larry Landweber、Chip Elliott、Michael Zink、Emmanuel Cecchet、Snigdhaswin Kar与Prabodh Mishra。2019。CloudLab的设计与运行。载于USENIX。USENIX协会，1–14。[11] Jayneel Gandhi、Vasileios Karakostas、Furkan Ayar、Adrián Cristal、Mark D. Hill、Kathryn S. McKinley、Mario Nemirovsky、Michael M. Swift与Osman S. Unsal。2016。面向高速虚拟内存的范围地址转换。《IEEE Micro》36，3（2016），118–126。[12] Goetz Graefe、Haris Volos、Hideaki Kimura、Harumi A. Kuno、Joseph A. Tucek、Mark Lillibridge与Alistair C. Veitch。2014。面向大数据的内存级性能。Proc. VLDB Endow. 8, 1 2014 37 48 10 14778 2735461 2735465 ( ), – . doi: . / . [13] Yunyan Guo与Guoliang Li。.由CXL驱动的数据库系统：机遇与挑战。载于ICDE。IEEE，5593–5604。[14] Xiangpeng Hao、Xinjing Zhou、Xiangyao Yu与Michael Stonebraker。2024。面向分层主存的缓冲区管理。SIGMOD 2, 1 (2024), 31:1–31:26。[15] Stavros Harizopoulos、Daniel J. Abadi、Samuel Madden与Michael Stonebraker。2008。镜中的联机事务处理，以及我们在其中的发现。载于SIGMOD。981–992。doi:10.1145/1376616.1376713[16] Yibo Huang、Newton Ni、Vijay Chidambaram、Emmett Witchel与Dixin Tang。2025。Pasha：面向CXL Pod的高效、可扩展数据库架构。载于CIDR。[17] 英特尔公司。2024。CXL* 3型内存设备软件指南。修订版1.1。https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://cdrdv2-public.intel.com/643805/643805_CXL_Memory_Device_SW_ Guide_Rev 1_1.pdf。访问于2026年2月9日。[18] John C. McCallum。2026。内存价格 1957+。https://docs.google.com/spreadsheets/d/1qF5vJ2-jkRygMQQIk3fj6G-WEW0ScDiKjyGseovFGqc/。访问于2026年2月11日。[19] Michael Kerrisk。2025。move_pages(2) — Linux手册页。https://man7.org/ linux/man-pages/man2/move_pages.2.html。访问日期：2026年2月11日。[20] Andi Kleen。2026。mbind — Linux手册页。https://man7.org/linux/man-pages/man2/mbind.2.html。访问日期：2026年2月11日。[21] Sangjin Lee、Alberto Lerner、Philippe Bonnet与Philippe Cudré-Mauroux。2024。数据库内核：通过CXL实现数据库系统与高速存储的无缝集成。载于CIDR。[22] Viktor Leis、Adnan Alhomssi、Tobias Ziegler、Yannick Loeck与Christian Dietrich。2023。虚拟内存辅助缓冲区管理。Proc. ACM Manag. Data 1, 1 2023 7 1 7 25 10 1145 3588687 ( ), : – : . doi: . / . [23] Viktor Leis与Christian Dietrich。.云原生数据库系统与单内核：为现代硬件重新构想操作系统抽象。VLDB 17, 8 (2024 2115 2122 10 14778 3659437 3659462), – . doi: . / . [24] Viktor Leis、Michael Haubenschild、Alfons Kemper与Thomas Neumann。2018。LeanStore：超越主存的内存数据管理。载于ICDE。185– 196。doi:10.1109/ICDE.2018.00026[25] Jinshu Liu、Hamid Hadian、Yuyue Wang、Daniel S. Berger、Marie Nguyen、Xun Jian、Sam H. Noh与Huaicheng Li。2025。大规模系统性CXL内存特征分析与性能分析。载于ASPLOS。1203–1217。[26] Thomas Neumann与Michael J. Freitag。2020年。Umbra：具备内存级性能的基于磁盘的系统。载于CIDR。https://vldb.org/cidrdb/2020/umbra-a-disk-based-system-with-in-memory-performance.html

[27] Riki Otaki、Charles Benello、Aaron J. Elmore 和 Goetz Graefe。2025。采用分页内存管理的资源自适应查询执行。载于 *CIDR*。https://vldb.org/cidrdb/2025/resource-adaptive-query-execution-with-paged-memory-management.html[28] Riki Otaki、Jun Hyuk Chang、Aaron J. Elmore 和 Goetz Graefe。2025。通过间接寻址跳过增强事务处理。*VLDB*学报18，11（2025 4104 4116 10 14778 3749646 3749680 ），–。doi: ./。[29] Niklas Riekenbrauck、Marcel Weisgut、Daniel Lindner 和 Tilmann Rabl。2024。面向数据库系统、集成CXL设备内存的三层缓冲管理器。载于 *HardBD & Active*。395–401。[30] Felix Schuhknecht 和 Nick Rassau。2026。迈出关键一步：用户空间中的高效可靠细粒度NUMA迁移。*CoRR* abs/2602.05540（2026）。arXiv:2602.05540 http://arxiv.org/abs/2602.05540[31] Yan Sun、Yifan Yuan、Zeduo Yu、Reese Kuper、Chihun Song、Jinghan Huang、Houxiang Ji、Siddharth Agarwal、Jiaqi Lou、Ipoom Jeong、Ren Wang、Jung Ho Ahn、Tianyin Xu 和 Nam Sung Kim。2023。借助真正的CXL就绪系统和设备揭开CXL内存的奥秘。载于 *MICRO*。105–121。[32] Alexander van Renen、Viktor Leis、Alfons Kemper、Thomas Neumann、Takushi Hashida、Kazuichi Oe、Yoshiyasu Doi、Lilian Harada 和 Mitsuru Sato。2018。在数据库系统中管理非易失性内存。载于*SIGMOD*。1541–1555。doi:10.1145/3183713.3196897[33] Marcel Weisgut、Daniel Ritter、Florian Schmeller、Pınar Tözün 和 Tilmann Rabl。2025。CXL-Bench：共享CXL内存访问基准测试。载于 *ADMS*。https://pure.itu.dk/en/publications/cxl-bench-benchmarking-shared-cxl-memory-access/[34] Lingfeng Xiang、Zhen Lin、Weishu Deng、Hui Lu、Jia Rao、Yifan Yuan 和 Ren Wang。2024。Nomad：通过事务性页面迁移实现非独占内存分层。载于 *USENIX*。19–35。[35] Tong Xing 和 Antonio Barbalace。2025。利用CXL共享内存池重新思考应用程序地址空间。载于 *HCDS*。52 59 10 1145 3723851 3723858 –。doi: ./。[36] Zi Yan、Daniel Lustig、David W. Nellans 和 Abhishek Bhattacharjee。2019。敏捷页面管理，面向分层内存系统。载于 *ASPLOS*。331–345。[37] Xinjun Yang、Yingqiang Zhang、Hao Chen、Feifei Li、Gerry Fan、Yang Kong、Bo Wang、Jing Fang、Yuhui Wang、Tao Huang、Wenpu Hu、Jim Kao 和 Jianping Jiang。2025。释放CXL在云原生数据库解聚内存中的潜力。载于 *SIGMOD*。689–702。[38] Xinjing Zhou、Joy Arulraj、Andrew Pavlo 和 David E. Cohen。2021。Spitfire：面向易失性和非易失性内存的三层缓冲管理器。载于 *SIGMOD*。2195–2207。doi:10.1145/3448016.3452819[39] Xinjing Zhou、Viktor Leis、Xiangyao Yu 和 Michael Stonebraker。2025。透过棱镜看OLTP：16年后的回顾——通信成为新的瓶颈。载于 *CIDR*。https://vldb.org/cidrdb/2025/oltp-through-the-looking-glass-16-years-later-communication-is-the-new-bottleneck.html