迈出飞跃：用户空间中高效可靠的细粒度 NUMA 迁移

Felix Schuhknecht 美因茨
约翰内斯·古腾堡大学，德国
schuhknecht@uni-mainz.de

Nick Rassau 美因茨约
翰内斯·古腾堡大学，德国
rassau@uni-mainz.de

arXiv:2602.05540v1 [cs.DB] 5 Feb 2026

摘要

Mod代的多插槽架构提供单一虚拟地址空间，但会将主存物理划分为多个区域，每个区域连接到一个 CPU 及其核心。虽然这简化了使用方式，但开发人员必须了解非均匀内存访问（NUMA）：在线程运行所在核心的本地 NUMA 区域进行访问，成本明显低于访问核心远端区域。如果查询处理并行分布在多个区域的核心上，那么查询所操作的数据库部分显然也应分布在相同区域，以确保本地访问。由于当前的数据放置方式可能并不合适，可以将页从一个 NUMA 区域迁移到另一个区域，以改善这种情况。为此，有几种不同的方案：一种是依赖 Linux 中集成的自动NUMA均衡，该机制根据观测到的访问模式和频率进行调节。另一种是通过系统调用 迁移_页()主动触发迁移。遗憾的是，这两种方案在功能集和性能方面都有明显缺陷。作为替代方案，我们提出一种名为 页_飞跃() 的新型用户空间迁移方法，它利用虚拟内存子系统的特性，以高性能异步执行页面迁移。该方法（a）由用户主动触发，（b）确保最终迁移所有页，（c）正确处理并发写入，（d）支持池化内存，（e）根据工作负载自适应调整迁移粒度，以及（f）同时支持小页和大页。

1 引言

多插槽服务器架构提供单一的虚拟地址空间，但会将主存物理划分到多个区域中，每个区域都连接到一个 CPU 及其核心。虽然这种单一的虚拟地址空间简化了开发，但程序员仍必须谨慎：如果线程访问的物理内存位于与其当前运行所在核心不同的 NUMA 区域，那么这种远程访问的代价将显著高于本地访问。因此，在追求高性能时，必须考虑这些所谓的非统一内存访问（*NUMA*） [3] 。在数据库管理系统中，如果查询处理跨多个区域的核心进行并行化，那么应将查询所操作的数据库部分分布到相同的区域中，以确保本地访问。遗憾的是，当前的数据放置可能并不适合这种并行化。为改善这一状况，可以在运行时将单个物理内存页从一个 NUMA 区域迁移到另一个区域，使其更接近工作线程，从而加快后续访问。

要执行迁移，有两种不同的选择：（1）依赖操作系统的隐式页面迁移 机制，该机制由内核自动且透明地执行。其核心概念是，操作系统监控访问次数以及已分配内存区域上的访问模式，并按一定频率进行。如果它观察到足够多的远程访问，就可能决定将页迁移到不同的区域。当然，迁移页的代价很高，但未来可能会有所回报，因此该启发式方法旨在平衡这一权衡。Linux将隐式页面迁移这一概念称为自动 *NUMA* 平衡 [4], ，在许多发行版上默认启用（/proc/sys/kernel/numa_balancing设为1），并适用于由小页和透明大页支持的匿名内存分配，例如由mmap()创建的分配。虽然自动NUMA均衡以透明方式工作，但它有两个缺点：（a）用户无法控制页面迁移何时发生，甚至无法控制它是否发生。正如我们的评估将表明的那样，数据库管理系统面对事务型工作负载时常见的频繁并发修改，会使自动 NUMA 平衡常常“等待”负载较低的时机来执行迁移，而这样的时机实际上可能永远不会到来。这会导致持续不断的不必要远程访问。此外，（b）迁移是在新分配的内存中进行的，不允许迁移到已经分配的池化内存中。这会造成代价高昂的缺页故障，如果应用程序本来就会池化内存（数据库管理系统经常如此），就会使迁移变得不必要地昂贵。这些问题将我们引向第二类页面迁移机制，即（2）显式页面迁移 方法。在这种情况下，页的迁移由程序员通过调用相应函数并指定要迁移的页来主动触发。在Linux中，该迁移函数是古老的move_pages() 系统调用，它由numa_move_pages() 方法 [8] （来自libnuma）封装。它允许从用户空间迁移当前已分配页的指定子集。虽然该系统调用提供了对迁移的控制，但不幸的是它也存在问题：（a）无法调整迁移粒度；迁移始终以页大小进行。如果必须迁移大量页，这会严重拖累调用线程。（b）仍然无法保证所有页的页面迁移都会执行。如果某个页当前正忙（即子系统持有对该页的引用），或者该页在迁移期间变脏，就可能干脆不迁移它。（c）与自动 NUMA 平衡一样，它不允许迁移到池化内存中，因此也会遭受缺页故障。

1.1 用户空间重映射：页面_leap()

Ideal可由用户空间控制，能够可靠地迁移所有请求的页，并支持迁移到池化内存中。此外，该方法应具备线程安全性，并能容忍并发修改。接下来，我们提出这样一种方法，称为页面_leap()。其核心概念基于内存重接技术 [16]，通过该技术可以在运行时从用户空间操纵虚拟地址到物理地址的映射

。为执行页面迁移，我们分两步进行：一步在物理层，另一步在虚拟层。在物理层，页面内容以非原子方式从源 NUMA 区域复制到目标 NUMA 区域，也可能复制到已池化的内存中。在虚拟层，随后将虚拟页原子地重新映射到目标区域的物理页，使其变为“活动”状态。为检测并处理可能导致正在进行的页面迁移失效的并发写入，我们使用段错误处理程序，使该页的迁移失效并触发新一轮迁移尝试，直至最终所有页都完成迁移。我们的方法还支持不同的迁移粒度，并会根据观察到的工作负载自适应地调整。表 1 总结了页面_leap() 与自动 NUMA 平衡和move_pages()相比所具备的特性。

方法	池化	受控	可靠	自适应
自动 NUMA 平衡	×	×	×	×
move_pages()	×	✓	×	×
page_leap()	✓	✓	✓	✓

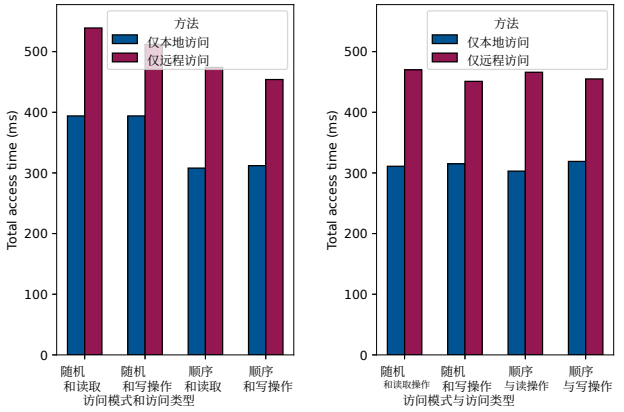
表 1：自动 NUMA 平衡、move_pages()与我们提出的方法page_leap()的特性比较。

1.2 实验环境与设置

在即将进行的实验评估中，我们使用一台配备两颗 Intel Xeon Gold 6326 处理器的双路服务器，运行 Alma Linux 及原版内核 5.14。每个 NUMA 区域拥有 128GB 随机存取存储器，总计 256GB。自动 NUMA 平衡通过将 /proc/sys/kernel/numa_balancing 设置为 1 来启用）代码使用 g++11.4 以 O3 优化级别编译。所有报告的运行时间均为三次运行的平均值。在后续所有实验中，我们使用 taskset -c 1 命令 [9]，将执行全部被测内存访问的进程主线程固定到 NUMA 区域 1 的第一个核心上。这意味着在下文中，NUMA 区域 0 为远程区域，而 NUMA 区域 1 为本地区域。要为 page_leap() 使用大页，必须挂载 hugetlbfs 文件系统，并在每个 NUMA 区域上预先分配足够数量的大页。对于我们已相应扩展的move_pages()，情况同样如此。自动 NUMA 平衡依赖透明大页，因为它不支持迁移hugetlbfs 大页。

2 本地访问与远程访问

我们首先分析在当前机器上，远程访问实际上比本地访问贵多少。为此，
我们分别使用 mbind() 方法，在 NUMA 区域 0（远程）和 NUMA 区域 1（本地）上分配 4GB 内存区域。该方法会将内存永久绑定到相应区域。用随机整数初始化内存后，我们在运行于 NUMA 区域 1 的主线程中，对该区域进行一系列访问。我们测试了四种不同的访问模式（顺序读写和随机读写），以观察访问模式是否会影响开销。对于顺序读写，我们只需从区域起始位置到末尾逐字节访问，总计约进行 42.亿次顺序访问。对于随机读写，我们访问 1000 万个随机选取的字节。图1展示了小页和大页两种情况下的结果。我们可以看到，对于所有访问模式，远程访问的开销确实显著高于本地访问



(a) 小页。 **(b) 大页。**

图1：不同访问模式下小页和大页的本地访问与远程访问。

，这说明本地访问十分必要，也因此需要高效的 NUMA 迁移。有趣的是，顺序访问时远程访问与本地访问之间的差距，与随机访问时一样明显，这表明在顺序访问模式下，预取无法掩盖远程访问的开销。

3 使用 move_pages() 进行页面迁移w

下一移。为了评估 move_pages() 的开销，我们首先将其与通过估算 move_pages() 的最佳情况进行比较。传统 memcpy() 将所有页的内容从一个非统一内存访问区域复制到另一个区域进行比较。由于每种迁移方法至少都必须物理复制所有页的内容，因此这一基线是任何迁移方法所能达到的最佳水平。在这里，我们区分复制到新分配的内存和复制到池化内存的情况：前者意味着复制是对内存的首次访问，会触发页错误；后者则是复制到已经发生过页错误的内存中。请注意，memcpy() 并不能恰当地替代 move_pages() 或其他迁移方法，原因有两个限制：(1) 它会使数据位于新的虚拟位置。因此，所有使用该内存的组件都必须更新其引用。相比之下，使用迁移方法时，迁移过程完全透明。(2) 在 memcpy() 期间，其他线程对“旧”页的并发写入会破坏正确性。

图 2 展示了小页和大页的结果。可以看出，对于小页，move_pages() 相较于复制到新分配内存的 memcpy()，开销仅为 18%；而相较于复制到已发生页错误的内存中的 memcpy()，开销则高达 82%。对于大页，令我们惊讶的是，复制到新分配内存中的 memcpy() 实际上比 move_pages()稍慢，我们认为这是因为 move_pages() 在一个单独的线程中运行，且该线程被固定在目标区域。当复制到池化内存时，我们也观察到 move_pages()的开销为 46%。总的来说，这表明对于小页和大页，使用池化内存时，相比 move_pages() 显然还有改进空间。

4 页_跃迁()架构Th

核心概念
n page_leap() 利用虚拟内存与物理内存分离的特性。如果虚拟页 vpage0 当前映射到物理页 ppage0

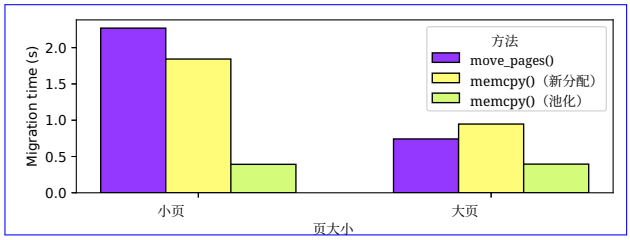


图 2：对于小页和大页，将页从一个 NUMA 区域迁移到另一个区域时，`move_pages()` 与`memcpy()` 的耗时。

该页分配在 NUMA 区域 0 上，我们只需执行两个简单步骤即可完成迁移：首先，从该区域的内存池中分配一个位于 NUMA 区域 1 上的物理页ppage1，并使用 `memcpy()`将源物理页的内容复制到这个目标物理页。其次，复制完成后，使用 `mmap()`自动将虚拟页vpage0重新映射到物理页 ppage1，这样后续访问就会命中 NUMA 区域 1 上的 ppage1。

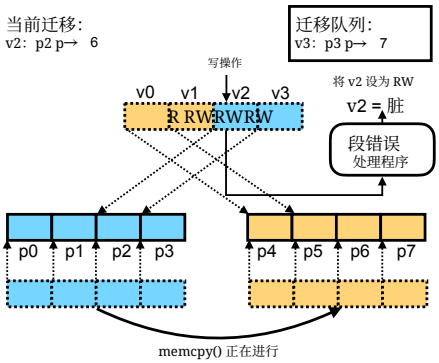
为了能够进行这种重新映射，我们首先采用一种称为内存重连的技术 [14, 16, 17]，它通过所谓的主存文件，在用户空间中提供对物理内存的访问句柄。借助 `mmap()`，可以将虚拟页映射（或重新映射）到主存文件中的偏移量，从而能够在运行时自由调整虚拟内存与物理内存之间的映射。这种技术还有一个不错的副作用：重新映射可以按多个相邻页的粒度进行，从而减少对大小为页大小整数倍的区域执行的 `mmap()` 调用次数。

4.1 正确处理并发写操作

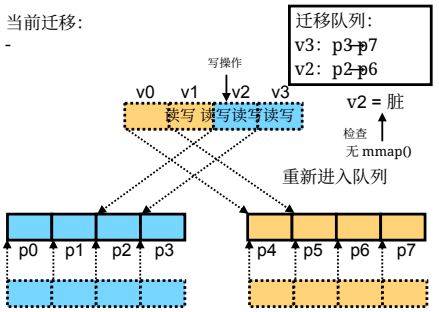
So f[ar, i] f 一次写操作是针对 formed 某个部分 f th源物理页 h

中已复制的部分时，在重新映射到目标物理页之后，该写操作就会丢失。为处理这种情况，我们必须 (a) 检测当前正在复制的区域是否发生了写操作。如果检测到写操作，(b) 就必须使已复制区域失效，并重试迁移。为实现 (a)，同时避免要求用户主动通过特殊接口转发所有写操作，我们使用信号处理来完成这项工作。具体来说，如图3a 中针对页面 v2 所示，我们使用 `mprotect()` 系统调用，将待迁移区域的保护属性设为只读。这样，对该区域的写尝试会触发段错误，我们随后可以通过自定义段错误处理程序 [13, 18] 捕获该错误。这个处理程序会预先注册，但除此之外就是一个普通的用户空间函数；发生段错误时，操作系统会调用它，并向其传递发生写尝试的虚拟内存地址。在此处理程序中，我们随后会在辅助结构中将该区域标记为脏，将其保护属性重置为可写，然后返回以重试写尝试，此时写操作便会成功。为处理 (b)，当某个区域的复制已执行完毕，即将通过重新映射生效；此时我们首先检查它是否变脏了[1],，如图3b所示。只有在它没有变脏的情况下，我们才执行重新映射。否则，我们会将该区域放入队列，以便重试迁移；正如下一节所述，我们可以自适应地拆分该区域，以提高迁移成功的几率。该过程会持续进行，直到

[1]在我们的协议中，即使某个区域已完全复制完成，它仍有可能变脏；也就是说，在相应的重新映射发生之前，如果有写操作发生，就会出现这种情况。这种罕见情况会导致不必要地重试该区域，但可以保留这次写操作。



(a) 正在对区域 v2 执行写操作，而该区域此时正在被复制。这会触发段错误，并将该区域标记为脏。



(b) 执行重新映射之前，会检查该区域是否在此期间变脏。如果是，则不会执行重新映射，而是将该区域重新加入队列，以便稍后迁移。

图3：正确处理四个页的并发写入，其中第三个页当前正在迁移。

直到所有页都已迁移，或达到指定的超时。请注意，这是相对于 `move_pages()` 的一项优势，后者并不保证所有页都会被迁移。

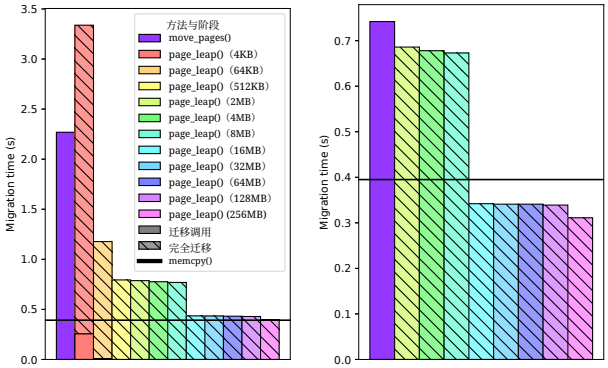
4.2 自适应区域大小

由于 `page_leap()` 能够以页大小的整数倍任意选择区域大小进行迁移，问题在于如何选择区域大小：区域过小可能会因需要重复的系统调用而产生不必要的开销，而区域过大则会增加区域变脏并导致重试的可能性。为了在无需了解工作负载及其在修改方面的压力的情况下处理该问题，`page_leap()` 实现了一种自适应方案：用户只需选择一个初始区域大小作为起点——在实验评估中，我们将确定这样一个合适的初始大小。随后，在迁移过程中，如果某个区域变脏且必须重试，`page_leap()` 会根据缩减因子将该区域拆分为多个更小的区域。由此，`page_leap()` 会自动根据工作负载的压力调整其区域大小，进而调整迁移粒度。此外，对数据集子集的偏斜修改只会减小处于压力之下的那些区域的区域大小。

5 无并发写入的迁移

在图4中，我们首先评估在没有并发访问的情况下，`page_leap()`与`move_pages()` 分别迁移小页和大页所需的时间，这相当于两种方法的最佳情况。

对于这两种方法，我们报告迁移完全完成所需的时间。对于 page_leap()，我们分别测试了十一种（小页）和八种（大页）不同的初始区域大小。由于本实验中没有并发访问，因此不会发生重试，也就不会进行区域大小的自适应调整。此处的比较不包含自动 NUMA 平衡，因为没有访问会触发迁移；我们将在第 6 节中纳入并讨论它。作为基准，我们将直接使用 memcpy() 的迁移时间作为理论最优值。



(a) 小页（池化）。 **(b) 大页（池化）。**

图4：move_pages()与page_leap() 在无并发访问时，不同粒度和页大小下的性能对比。memcpy() 的性能接近理论最优值。

我们可以看到，对于两种页面大小，page_leap() 在几乎所有配置下都优于move_pages() 。只有在区域大小为4KB时，系统调用开销才会变得过于显著。对于小页和大页，区域大小分别为64KB和16MB时，page_leap() 的性能已达到move_pages() 的两倍或更高，表明在无并发访问的迁移场景中，page_leap() 具有优越性。我们还可以观察到，每张图中都存在平台期，性能在512KB和16MB处出现跃升。这表明，在采用自适应拆分时，这些区域大小非常适合作为初始区域大小。此外，我们可以看到，对于两种页面大小，当区域大小为16MB时，page_leap() 已达到memcpy()的最优值，这表明page_leap() 的性能瓶颈转为复制阶段。

6 并发写入下的迁移

接下来，我们检查并发写入下的性能。对于 页_leap()，信号处理和迁移重试可能带来的额外开销也会产生影响。对于每种方法，我们都会异步执行迁移，同时以指定的次写入/秒吞吐量，持续对内存执行均匀随机选取的随机写入突发。作为低、高和极高压力情况的代表，我们分别测试 10K、100K 和 10M（小页）以及 100M（大页）次写入/秒。此外，我们还纳入一种偏斜工作负载：在 100K 次写入/秒中，75% 的写入集中于仅占内存 128MB（3.125%）的区域。触发写入突发有两个目的：首先，我们观察并发突发写入会如何对各方法的迁移产生负面影响，例如减慢迁移速度、增加重试次数，甚至导致部分页未能迁移。其次，我们观察迁移是否也会对写入突发造成负面影响，以及从何时起无法再维持请求的吞吐量。以及请求的吞吐量从何时起无法再维持。请注意，一旦迁移完成，或 10 秒后发生超时，写入突发也会立即终止，实验随之结束。随后，我们还会报告所有页的当前位置或错误状态。对于 页_leap()，为完整起见，我们再次展示此前测试过的所有区域大小，但重点关注此前确定的最有前景的 512KB 和 16MB 大小；这里所述的区域大小现在是初始区域大小——结合所用的缩减因子 2，某个区域重试时会自动将其一分为二，并重试对这两个半区的迁移。对于自动 NUMA 平衡，由于它不会以任何方式发出开始或完成的信号，我们每 100ms 检查一次所有页是否都已迁移；若是，则停止测量。否则，我们将持续测量，直到达到超时。

我们首先来看图5中小页的结果，其中报告了迁移时间、实际达到的吞吐量占请求吞吐量的百分比，以及页面状态。先比较迁移时间可以发现，对于所有工作负载，在许多初始区域大小下，page_leap() 的表现都明显优于move_pages() 和自动 NUMA 平衡。先前确定的 16MB 在每秒 10K 和 100K 次写入的场景中显然表现最佳，而在请求每秒 10M 次写入的极端场景中，初始区域大小为 512KB 时表现最佳。在倾斜场景中，两种区域大小的表现不相上下。因此，对于小页，我们建议通常以 16MB 的初始区域大小运行page_leap() ，以达到最佳平衡点——在正常压力下，我们可以发挥较大复制和较少重新映射的优势；而在极端压力下，该方法仍能足够快速地适应较小的区域大小，从而继续胜过其他方案。查看写入的实际吞吐量可以发现，所有方法都能妥善应对低压和高压场景，吞吐量接近 100%。只有在极端压力下，吞吐量才无法维持。就页面状态而言，page_leap() （采用合理的初始区域大小时）和move_pages() 始终能在超时发生前迁移所有页。自动 NUMA 平衡在所有工作负载下表现都很差，因为它只迁移了少部分页。对于图7中的大页，可以看到，在低压、高压和倾斜场景中，初始区域大小为 16MB 的page_leap() 仍比move_pages() 快至多 2 倍，但总体优势较小。由于底层页大小更大，现在所有方法的运行表现都更接近理论上的复制最优值。必须承认，move_pages() 在极端场景下表现更稳定——但其吞吐量也更低。此外，尽管自动 NUMA 平衡发生了超时，最终仍迁移了所有页。这表明，对于使用页数较少的大页，自动 NUMA 平衡会利用写入突发结束后的“暂停”时间来实际迁移页。不过，这也再次体现出它在迁移实际发生的时间以及是否会发生方面难以预测。

接下来，在表2中，我们量化在 100K 次写入/秒情况下，并发写入期间页面_leap() 的实际开销。为此，我们首先测量不同区域大小下，memory页面_leap() 实际需要复制的总量，包括重试期间额外复制的内存。然后，我们测量通过memcpy() 将相同数量的内存从一个区域复制到另一个区域所需的时间。这里，我们还确保逐个复制各个区域。

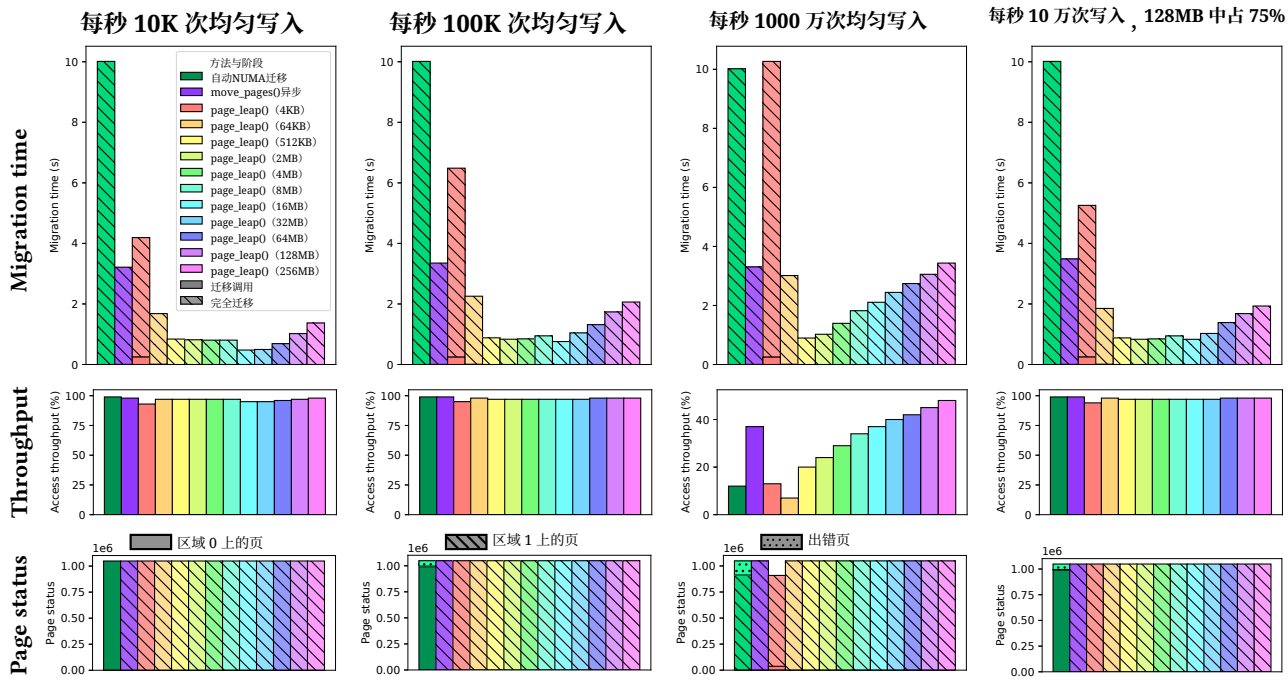


图5：并发写入下的迁移小页以及缩减因子 2页_跳转()。

随后，我们报告两者的时间差，这基本上反映了页面_leap() 在原始复制之外所产生的一切开销。表2展示了两种页面大小下的内存开销和时间开销，其中包括不切实际的极端值 4KB 和 256MB，以及建议的初始区域大小 512KB 和 16MB。对于 4KB 的情况，可以清楚地看到系统调用开销是瓶颈，因为完全没有发生重试。对于 256MB 的情况，额外复制量约为 420%，导致时间开销分别达到 27.5% 和 18.3%。然而，建议的大小能够将内存开销和时间开销都控制在memcpy()之上约 50% 以内，这表明在合理选择初始区域大小时，页面_leap() 具有较高的效率。

	小页		大页	
Area	**内存**	**Time**	**内存**	**Time**
4KB	0MB (0.0%)	5635ms (659%)	-	-
512KB	**5.5MB（0.13%）**	**210ms（31.3%）**	-	-
2MB	42MB（1.02%）	155ms（22.9%）	66MB（1.61%）	34毫秒（5.1%）
16MB	**2.1GB（52.5%）**	**245毫秒（47.5%）**	**1.8GB（45%）**	**135毫秒（28.4%）**
256MB	17.1GB（427.5%）	437ms（27.5%）	16.80GB（420%）	274ms（18.3%）

表2：相较于 memcpy() 的时间开销和内存开销。

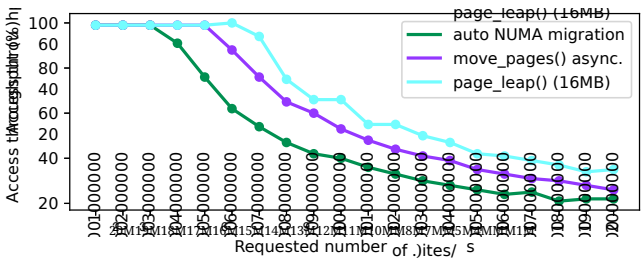


图 6：运行 10s 的写入吞吐量。

到目前为止，我们只分析了迁移期间的访问吞吐量：迁移结束（或发生超时）后，便立即停止触发写入突发。为了进一步衡量快速迁移带来的积极影响，即能够更早执行本地访问，我们调整了实验，让写入突发持续整整 10 秒，不受迁移何时完成的影响。图 6 展示了小页在请求吞吐量变化时的结果。可以看到，随着请求吞吐量提高，页_leap() 能够比两个基线方案更长时间地维持该吞吐量，并且始终表现最佳。例如，在请求吞吐量为每秒 600 万次写入时，页_leap() 仍能完全维持该吞吐量，而自动 NUMA 平衡仅能达到其 65%。这表明，尽可能快速地迁移确实大有裨益，因为在整个持续时间内发生的远程访问会更少。

7 处理数据库工作负载

最后，让我们评估一下 页_leap() 如何处理实际的数据库工作负载。我们评估这样一种场景：它可能出现在基于 morsel 的数据库引擎中 [1]，该引擎会在语义上将数据库划分到两个 NUMA 区域中；其中一个区域的线程当前处于空闲状态，而所有仍待处理的 morsel 都位于另一个区域。在这种情况下，该线程可以将这些 morsel 迁移到区域 1，以加快查询处理。

为分析该场景，我们在 NUMA 区域 0 上分配一组总大小为 1GB 的 TPC-H lineitem 表 morsel。随后，在查询处理之前，我们分别通过move_pages() 与 page_leap() 将这些 morsel 迁移到池化内存中。此外，我们还纳入自动 NUMA 平衡。作为查询，我们评估 TPC-H Q1 和 Q6 的手写变体，因为它们仅作用于 lineitem，并研究其在有无立即发起的并发写入时的性能；这些并发写入由一个单独的线程执行，该线程向均匀选取对象的 L_ORDERKEY 字段执行 $10M$ 次写操作

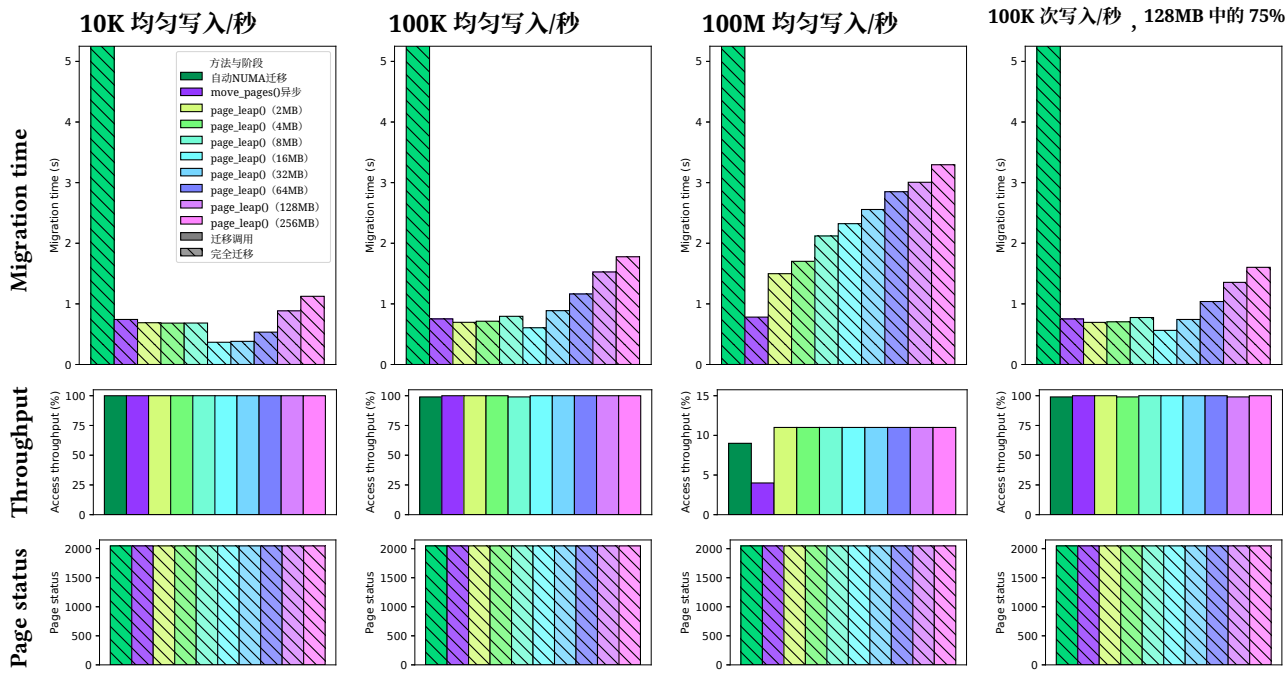


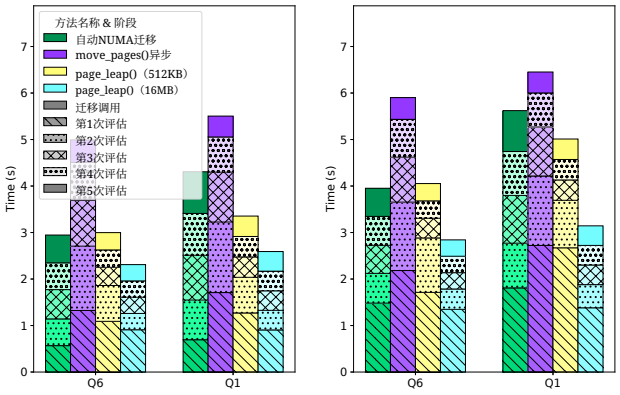
Figure 7: 大页在并发写入下的迁移，且缩减因子为 2，用于page_leap( ).

行。由于 L_ORDERKEY 既不被 Q1 也不被 Q6 使用，因此这不会影响查询结果，但仍会拖累迁移过程。图8展示了异步迁移调用（针对page_leap() 和 move_pages()）的耗时，随后是相应查询的连续五次执行。我们可以观察到，page_leap()再次明显优于 move_pages()；对于两种推荐的区域大小，page_leap() 从第二次查询起就已相对 move_pages()显现出收益。这表明 page_leap() 在数据库工作负载场景中同样能够有效运行。

8 相关工作

在结束之前，我们将讨论该领域的相关工作。

首先，就非统一内存访问迁移的通用技术而言，还有一些内置方法值得一提：其中一个典型代表是numactl，这是一款命令行工具，也可以用来触发页面迁移。不过，由于它不像其他方法那样便于在进程内使用，因此我们没有将其作为此处的基线。此外，还有来自libnuma的系统调用migrate_pages()。然而，与move_pages()不同，它无法只迁移进程中的部分页，而是始终迁移所有页，这通常不够灵活。因此，它无法实现细粒度的页面迁移。与move_pages() 相关的还有近期的一项工作 [12],，该工作提出了系统调用move_pages2()的一个更快版本。不过，由于这需要修改内核，而page_leap() 无需修改即可在原版内核上运行，因此我们没有将其作为此处的基线。在数据库领域，非统一内存访问感知算法和数据结构多年来一直备受关注。这些工作从整体角度关注非统一内存访问系统上的高效数据放置 [11] 与调度 [10]、非统一内存访问感知连接 [5], 以及非统一内存访问感知执行引擎 [6]。在内存重连方面，也有许多其他利用该技术的工作，涉及的范围包括数据结构，如链表 [16], 紧凑内存数组 [7],可伸缩哈希表方案 [15], 和基数树 [2]；也包括非原地划分 [16] 和快照 [13, 18]等算法。



（a）无并发写入。 **（b）有并发写入。**

图8：TPC-H 工作负载。

9 结论

本文分析了内置隐式与显式页面迁移机制、自动 NUMA 平衡以及 移动_页() 的不足，并提出了一种可在原版 Linux 内核的用户空间中运行的实用替代方案。我们通过实验证明，该替代方案在确保并发写入下的正确性并提供最终完成迁移保证的同时，显著降低了迁移开销。此外，我们的方法支持向池化内存进行迁移，相较基线方法具有显著优势。

制品

本项目的全部代码（包括构建说明和所有产物）可在 https://gitlab.rlp.net/fschuhkn/efficient_numa_ page_ migration 获取。

参考文献

[1] Kayhan Dursun、Carsten Binnig、Ugur Çetintemel、Garret Swart 和 Weiwei Gong。2019。面向异构多核的 Morsel 驱动查询执行引擎。VLDB Endowment会议论文集 12, 12 2019 2218 2229 10 14778 3352063 3352137 ( ), –。doi: . / .

[2] Philipp Fent 、 Michael Jungmair 、 Andreas Kipf 和 Thomas Neumann。2020. START - 自调优自适应基数树。发表于第 36 届 IEEE 国际数据工程会议研讨会（ICDE Workshops 2020），美国德克萨斯州达拉斯，2020 年 4 月 *20 24, 2020* 147 153 10 1109 49219 2020 00015 -。IEEE，–。doi: . /I

[3] CDEW . . Mrunal Gawade 和 Martin L. Kersten。2015. 通过内存映射实现 NUMA无关性。发表于第 11 届新硬件数据管理国际研讨会论文集（DaMoN 2015），澳大利亚维多利亚州墨尔本，2015 年 5 月 31 日至 6 月 4 日，Ippokratis Pandis 和 Martin L. Kersten（编）。ACM，4:1–4:7。https://doi.org/ 10 1145 2771937 2771948 . / . 红帽。. 自动NUMA均衡。https://docs.redhat.com/en/ documentation/red_hat_enterprise_ linux/7/html/virtualization_tuning_and_ optimization_guide/sect-virtualization_ tuning_optimization_guide-numa- auto_numa_balancing [5] Harald Lang、Viktor Leis、Martina-Cezara Albutiu、Thomas Neumann 和 Alfons Kemper。2013。大规模并行 NUMA感知哈希连接。发表于《内存数据管理与分析——第一届和第二届国际研讨会论文集》，IMDM 2013，意大利加尔达湖畔里瓦，2013 年 8 月 26 日；中国洪州，2014 年 9 月，修订精选论文（《计算机科学讲义》，第 卷），Arun Jagatheesan、Justin J. Levandoski、Thomas Neumann 和 Andrew Pavlo（编）。Springer，

[4] 2025. 3 14 10 1007 978 3 319 13960 9_1 –。doi: . / - - - - Viktor Leis、Peter A. Boncz、Alfons Kemper 和 Thomas Neumann。2014。Morsel 驱动并行：面向多核时代的 NUMA感知查询执行框架。发表于数据管理国际会议（SIGMOD 2014），美国犹他州斯诺伯德，2014 年 6 月 22—27 日，Curtis E. Dyreson、Feifei Li 和 M. Tamer Özsu（编）。ACM，

[6] 743 754 10 1145 2588555 2610507 –。doi: . / . Dean De Leo 和 Peter A. Boncz。2019. . 紧凑内存数组 —— 重新布线。发表于第 35 届 IEEE 国际数据工程会议（ICDE 2019），中国澳门，2019 年 4 月 *8 11, 2019* 830 841 10 1109 2019 00079 -。IEEE，–。doi: . /ICDE. .

[7] 2025. 7 Linux 手册页。. move pages()。https://man .org/linux/man-pages/ man2/move_ pages.2.html [9] Linux 手册页。2025。taskset。 https://www.man7.org/linux/man-pages/ man1/taskset.1.html[10] Iraklis Psaroudakis、Tobias Scheuer、Norman May、Abdelkader Sellami 和 Anastasia Ailamaki。2015。扩展并发主存列存储扫描：迈向自适应NUMA感知数据与任务放置。VLDB Endowment会议论文集 8, 12 2015 1442 1453 10 14778 2824032 2824043 ( ), –。doi: . / . Iraklis

[11] Psaroudakis、Tobias Scheuer、Norman May、Abdelkader Sellami 和 Anastasia Ailamaki。2016。主存列存储中面向分析型工作负载的自适应NUMA感知数据放置与任务调度。VLDB Endowment会议论文集 10, 2（2016）37 48 10 14778 3015274 ( ), –。doi: . / .

[12] 3015275 2025 Yeasir Rayhan 和 Walid G. Aref。. 重新审视主存数据库系统的页面迁移。CoRR abs/2503.17685（2025）。arXiv:2503.17685 doi:10.48550/ARXIV.2503.17685[13] Reza Salkhordeh、Felix Martin Schuhknecht、Hossein Asadi、Steffen Eiden 和 André Brinkmann。2025。刻不容缓：通过虚拟快照对大规模数据处理进行原位分析。发表于第 28 届扩展数据库技术国际会议论文集（EDBT 2025），西班牙巴塞罗那，2025 年 3 月 25—28 日，Alkis Simitsis、Bettina Kemme、Anna Queralt、Oscar Romero 和 Petar Jovanovic（编）。OpenProceedings.org，

[14] 438 450 10 48786 2025 35 –。doi: . / EDBT. . Felix Schuhknecht 和 Justus Henneberg。2023. . 利用部分虚拟视图加速主存表扫描。发表于第 19 届新硬件数据管理国际研讨会论文集（DaMoN 2023），美国华盛顿州西雅图，2023 年 6 月 18—23 日，Norman May 和 Nesime Tatbul（编）。ACM，89–93。 doi:10.1145/3592980.3595315[15] Felix Martin Schuhknecht。2024。抄近路：主动利用操作系统虚拟内存索引实现数据库索引的硬件加速。发表于第 14 届创新数据系统研究会议（CIDR 2024），美国夏威夷州夏米纳德，2024 年 1 月 14—17 日。www.cidrdb.org。 https://www.cidrdb.org/cidr2024/ papers/p58-schuhknecht.pdf [16] Felix Martin Schuhknecht、Jens Dittrich 和 Ankur Sharma。2016。RUMA 有戏：重接线用户空间内存访问完全可行！VLDB Endowment会议论文集 9, 10（2016），768–779。 doi:10.14778/2977797.2977803[17] Felix Martin Schuhknecht 和 Justus Henneberg。2023。迈向虚拟内存中的自适应存储视图。发表于第 13 届创新数据系统研究会议（CIDR 2023），荷兰阿姆斯特丹，2023 年 1 月 8—11 日。www.cidrdb.org。 https://www.cidrdb.org/cidr2023/papers/p65-schuhknecht.pdf

[18] Felix Martin Schuhknecht、Aaron Priesterroth、Justus Henneberg 和 Reza Salkhordeh。2021年。AnyOLAP：无需 ETL 即可对任意数据密集型应用进行分析处理。*VLDB Endowment*会议论文集 14, 12（2021年），2823–2826。http: // www.vldb.org/pvldb/vol14/p2823-schuhknecht.pdf