持久性内存I/O基元

Alexander van Renen 慕尼黑工业大学 renen@in.tum.de
卢卡斯-沃格尔 慕尼黑工业大学 vogell@in.tum.de
Viktor Leis Friedrich-Schiller-Universität Jena viktor.leis@uni-jena.de
Thomas Neumann 慕尼黑工业大学 neumann@in.tum.de
阿方斯-坎普 慕尼黑工业大学 kemper@in.tum.de

ABSTRACT

I/O延迟和吞吐量是基于磁盘的数据库系统的主要性能瓶颈之一。即将推出的持久性内存（PMem）技术，如英特尔的Optane DC持久性内存模块，有望弥补基于NAND的闪存（SSD）和DRAM之间的差距，从而消除I/O瓶颈。在本文中，我们提供了PMem在带宽和延迟方面的首批性能评估。基于这些结果，我们制定了有效使用PMem的准则，以及为PMem调整的两个基本I/O原语：日志写入和块冲刷。

ACM参考格式。
Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. .持久性内存I/O基元。在.ACM，纽约，美国，7页。

1 简介

今天，数据管理系统主要依靠固态驱动器（NAND闪存）或磁性磁盘来存储数据。这些存储技术以低成本提供持久性和大容量。然而，由于访问延迟较高，大多数系统也使用DRAM形式的易失性主存储器作为缓存。这就产生了传统的两层结构，因为DRAM由于其波动性、高成本和有限的容量而不能被完全使用。

新的存储技术，如相变内存，即将缩小内存和存储之间的这种基本差距。具体来说，英特尔即将推出的*Optane DC持久性内存模块*（Optane DC PMM）提供了内存和存储的最佳特性的组合--尽管正如我们在本文中所展示的，有一些权衡。这种持久性内存（PMem）像存储一样耐用，又像内存一样可由CPU直接寻址。我们还希望其价格、容量和延迟介于DRAM和闪存之间。

PMem有望大大改善存储技术的延迟，这反过来又会大大增加数据管理系统的性能。然而，由于PMem从根本上不同于现有的知名技术，它的性能特征也与DRAM和闪存不同。在这项工作中，我们展示了如何有效地实现原子化的日志写入和页面刷新--数据库系统的两个关键I/O基元。

虽然我们在数据库背景下进行了评估，但这两个I/O基元是可以转移到其他系统的，这一点从持久性内存开发工具包（PMDK）[1]也实现了这些基元就可以看出。报告的结果是基于英特尔的Optane DC PMM的原型，而不是基于软件或硬件的仿真。我们的贡献可以总结为以下几点。

- 我们提供了对英特尔Optane DC PMM的原生型的PMem的首批分析之一。我们强调了PMem的物理特性对软件的影响，并得出了有效使用PMem的指导方针。
- 我们介绍了一种持久化小数据块（事务性日志条目）的算法，与最先进的算法相比，延迟降低了2倍。
- 我们研究了将大数据块（数据库页面）以故障原子方式持久化到PMem的不同算法。 通过将写时复制方法与临时delta文件相结合，我们实现了显著的速度提升。

2 pmem的特点

在本节中，我们首先描述了我们是如何配置我们的系统的，然后再介绍延迟和带宽的结果。

2.1 设置和配置

有两种使用PMem的方式：内存模式和应用直接模式。在**内存模式**下，PMem取代DRAM作为（易失性）主内存，而DRAM作为一个额外的硬件人工缓存层（"L4缓存"）。这种模式的优点是它对遗留软件的工作是透明的，因此提供了一种以低成本扩展主内存容量的简单方法。然而，这并不利用持久性，而且由于PMem的低带宽和高延迟，性能可能会下降。 事实上，正如我们后面所显示的，当DRAM作为L4缓存而不是通常情况下的L4缓存时，访问数据有10%的开销。

因为在内存模式下不可能利用PMem的持久性，所以在本文的其余部分，我们专注于**应用直接模式**。与内存模式不同的是，应用程序直接模式没有触及常规的内存系统。它允许程序以内存映射文件的形式使用PMem。我们在下文中从开发者的角度描述这个过程。

我们正在使用一个双插槽系统，每个节点上有24个物理（48个虚拟）核心。该机器运行的是Fedora，Linux内核版本为4.15.6。每个插座有6个PMem DIMMs，每个128GB，6个DRAM DIMMs，每个32GB。

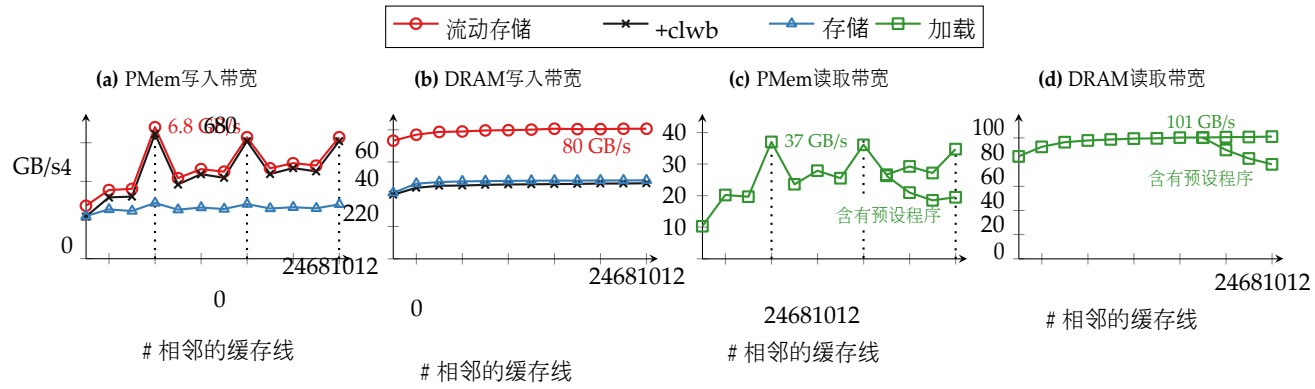


图1：PMem带宽：不同的访问粒度--24个线程的PMem带宽（**a**，**c**）与DRAM带宽（**b**）相比。**d**) 具有不同数量的相邻访问的高速缓存线。我们使用一种随机访问模式，允许失序执行。

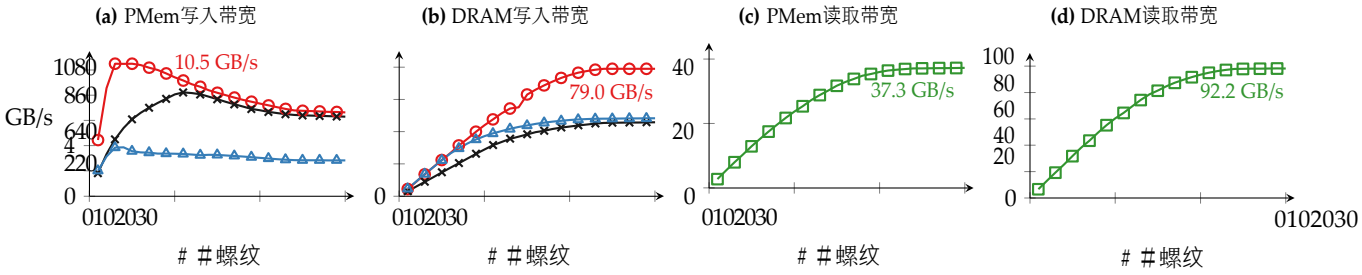


图2：PMem带宽：不同的线程数 - PMem带宽（**a**，**c**）与DRAM带宽（**b**，**d**）的比较，4个相邻的高速缓存线的线程数越来越多。我们使用一个随机访问模式，允许失序执行。

为了访问PMem，物理PMem DIMMs首先必须用ipmctl[1] 分组为所谓的*区域*。

```
ipmctl create -f -goal -socket 0 MemoryMode=0\
PersistentMemoryType=AppDirect
```

为了避免下面的实验因讨论NUMA效应（与DRAM上的效应相似）而复杂化，我们在socket 0上运行所有的实验。 一旦一个区域被创建，ndctl[2] 被用来在其上创建一个命名空间。

```
ndctl create-namespace --mode fsdax --region 28
```

接下来，我们在这个命名空间上创建一个文件系统（mkfs.ext4[3] ），并使用dax标志对其进行挂载（mount[4] ），这使得CPU可以直接对设备进行高速缓存线的访问。

```
mkfs.ext4 /dev/pmem28
mount -o dax /dev/pmem28 /mnt/pmem28/
```

程序现在可以在新挂载的设备上创建文件，并使用mmap[5] 将它们映射到他们的地址空间。

```
fd = open(("/mnt/pmem28/file", O_RDWR,
0); res = ftruncate(fd, SIZE);
ptr = mmap(nullptr, SIZE, PROT_WRITE, MAP_SHARED, fd, 0)。
```

指针可以用来直接访问PMem，就像普通内存一样。第3节讨论了如何确保写入PMem的值实际上是持久的。在本节的其余部分，我们讨论PMem的带宽和延迟。

[1] **ipmctl:** https://github.com/intel/ipmctl
[2] **ndctl:** https://github.com/pmem/ndctl
[3] **mkfs.ext4:** https://linux.die.net/man/8/mkfs.ext4
[4] **安装**：https://linux.die.net/man/8/mount
[5] **mmap:** http://man7.org/linux/man-pages/man2/mmap.2.html

2.2 带宽

重要的是要知道，PMem硬件在内部是在256字节的块上工作。一个小的写组合缓冲器被用来避免写放大，因为PMem和CPU之间的传输大小和DRAM一样，是64字节（缓存线）。

PMem的基于块（4条缓存线）的设计导致了一些有趣的性能特征，我们在图1中展示了这些特征。该实验测量了从PMem和DRAM的独立随机位置加载/存储的带宽。我们使用了一个插座的所有24个物理核心，以最大限度地增加并行访问的数量。图中显示了存储（PMem：（**a**），DRAM：（**b**））和加载（PMem：（**c**），DRAM：（**d**））的基准测试。在PMem上，性能明显取决于连续访问的缓存线的数量，而在DRAM上则没有明显的区别。只有在使用块大小的倍数（4条高速缓存线=256字节）时才能达到峰值吞吐量。

如同在DRAM上一样，在PMem上流式（非时间性）存储更有效率，因为修改后的缓存行不必先被加载，从而节省了内存带宽。然而，在PMem上，通过在每次存储后发出clwb（高速缓存行回写）指令，可以将常规存储的性能提高到流式存储的性能。clwb强制将数据缓存中的脏缓存行写到底层内存系统中（不驱逐缓存行）。虽然这对PMem（**a**）是有利的，但它并没有改变DRAM（**b**）的吞吐量。

图2进一步研究了这种影响，它显示了同样的实验，但我们没有改变加载/存储的缓存线的数量，而是改变了线程的数量。图中显示，clwb

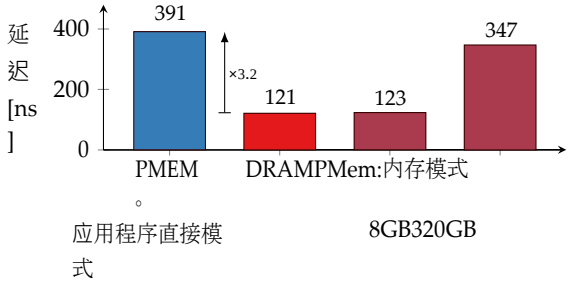


图3：读取延迟 - 随机访问读取延迟。

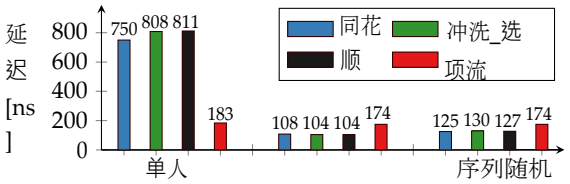


图4：持久性写入延迟 - 写入的访问延迟

只有在多个线程向PMem写入时，才有必要使用这个指令。随着线程的增多，缓存线从上一级的CPU缓存中被随机驱逐，因此越来越多的线程不按顺序到达PMem的写入缓冲器中。似乎到了一定程度（4个线程），缓冲器就不能再将缓存线合并到一个PMem块中写入。使用clwb指令，我们可以强制调整缓存行到达PMem写缓冲区的顺序，从而使它能够将相邻的缓存行合并成一个单一的块写。

我们观察到的另一个效果是，流媒体的吞吐量在3个线程左右达到峰值（使用clwb的商店为12个）。使用额外的线程会使吞吐量略微下降。

最后，图1（c）和（d）显示了硬件预取器的一个基本不相关但有些有趣的效果。从相邻的10个缓存行开始，预取器开始活动并取走额外的缓存行。如果这些是不需要的，就像我们的实验中一样，有效的吞吐量就会受到影响。

总之，从我们的实验结果来看，我们建议为带宽关键型应用制定以下准则。

- 算法不应该再被设计成适合在单个缓存行（64字节）上的数据，而是在PMem块（256字节）上。
- 在可能的情况下应使用流操作，其他情况下应使用clwb存储。
- 过度饱和的PMem会导致性能下降。实验表明，PMem的读
- 取带宽为

  比DRAM低2.6，写带宽低7.5。因此，对性能要求很高的代码应该选择DRAM而不是PMem（例如，通过在DRAM缓存中缓冲写入）。

2.3 延迟

虽然带宽对OLAP式的应用很关键，但延迟对OLTP工作负载更重要，因为访问模式从大型扫描操作（顺序I/O）转变为点查询，这基本上是对内存的随机访问。这些随机访问的性能被底层设备的延迟所支配。

为了测量PMem上加载操作的延迟，我们使用一个单线程并从随机位置执行加载。为了研究这种影响，我们通过链式加载来防止失序执行，使步骤*i*中的加载地址取决于步骤*i* 1中读取的值。

我们可以看到，DRAM的读取延迟比PMem低3.2倍。请注意，这并不意味着对PMem的每次访问都会慢很多，因为许多应用程序仍然可以从常规的CPU上的L3缓存中受益。当PMem在内存模式下使用时，它取代了DRAM作为主内存，而DRAM则作为L4缓存。在这个配置中，数据大小很重要：当使用8GB时（和其他模式一样），性能与DRAM相似，因为DRAM缓存捕获了所有的访问。然而，当我们把数据量增加到360GB时，DRAM缓存（在我们使用的插座上大约为200GB）就不会被频繁地击中，性能就会下降。

要在PMem上持久地存储数据，必须先写入数据，驱逐缓存线，然后用一个sfence来等待数据到达PMem。 这个过程在第3.1节有更详细的描述。为了测量PMem上持久化存储操作的延迟，我们使用一个单线程，将数据持久化存储到一个10GB大小的数组中。每一次存储都与缓存行（64字节）边界对齐。结果显示在图4中。

左边的四个条形图显示了连续写入同一缓存行的结果，中间的条形图是我们按顺序写入缓存行的结果，右边的条形图是随机写入的结果。在每种情况下，我们使用四种不同的方法来冲刷缓存行（从左到右：flush，flushopt，clwb，和流式存储）。

当数据被写入同一条缓存线时，应该首选流式存储。这种模式出现在许多数据结构（例如，带有大小字段的数组式结构）或算法（例如，用于时间标记的全局计数器）中，这些结构或算法都有一些经常被修改的全局变量。因此，为了有效地使用PMem，类似于为避免多线程编程中的拥堵而开发的技术也必须应用于PMem。在非流媒体指令中，没有明显的区别，因为Cascade Lake CPU没有完全实现clwb。英特尔已经添加了操作码，允许软件使用它，但目前将其实施为flush_opt。因此，流式操作和clwb应该优先于flush和flush_opt。

3 pmem的存储基元

PMem的低写入延迟（与其他存储设备相比）使其成为数据库系统、文件系统和其他系统软件中的理想候选者。然而，由于CPU缓存的存在，对PMem的写入只有在相应的缓存行被刷新后才会持久。算法必须明确地安排存储和刷新缓存行的顺序，以确保一个持久的数据结构总是处于一致的状态（在崩溃的情况下）。我们把这一属性称为*失败的原子性*，并在第3.1节中讨论它。英特尔的持久性内存开发工具包（PMDK）[1]，一个用于Pmem的开源库，通过提供两个故障原子性I/O原语来抽象出这种复杂性：日志写入（*libpmemlog*）和块/页刷新（*libpmemblk*）。在第3.3节和第3.2节中，我们应用前面制定的准则（第2节），将其应用于这两个问题，并分析其性能。

3.1 失败的原子性

如前所述，当数据被写入PMem时，存储不会立即传播到PMem设备上，而是在CPU上的常规缓存中缓冲。虽然程序不能阻止驱逐，但他们可以使用明确的回写或刷新指令来强制驱逐。这意味着PMem上的任何持久性数据结构都需要处于一致的状态，否则系统崩溃--中断更新操作--会导致重新启动后的不一致状态。下面的代码片段显示了一个元素是如何被附加到一个预先分配的缓冲区的。

```
结构 Buffer {
  int eles[128];
  int next;
};
```

```
空白的append(Buffer* buf, int ele)
{
  buf->eles[buf->next] = ele;
  clwb(&buf->eles[buf->next]);
  sfence () 。
  buf->next++;
  clwb(&buf->next);
  sfence()。
}
```

新的元素首先被复制到下一个空闲的槽中，并且相关的缓存行被强制写回PMem。 没有使用常规的刷新操作，而是使用clwb（缓存行回写），这是一个为PMem设计的高效的刷新操作，在刷新缓存行的同时不会使其失效。在缓冲区的大小指示器（next）被改变之前，必须发出一个sfence（存储栅栏）以防止编译器或硬件的重新排序。一旦next被写入，它将以同样的方式持久化到内存中。请注意，持久化下一个字段对于单个追加操作的失败原子性来说是没有必要的。然而，它是方便的，而且往往是后续代码所需要的（例如，另一个追加操作）。在下文中，我们将使用持久化屏障这个术语，持久化是指clwb和后续sfence的组合。

```
void persist(void* ptr) { clwb(ptr); sfence(); }
```

一般来说，持久性障碍是一种昂贵的操作，因为它迫使人们对PMem（或者更准确地说，对其内部电池支持的缓冲器）进行同步写入。因此，除了第2节中提出的准则外，在保持故障原子性的同时，尽量减少持久性障碍的数量也很重要。在下面两节中，我们展示了一个手动调整的记录和刷新页面的实现。

3.2 页面传播

除了日志，另一个需要I/O的基本存储引擎组件是缓冲区管理器。它负责在查询引擎访问页面时，从SSD/HDD加载（交换）页面到DRAM。当缓冲池已满时，缓冲管理器需要驱逐页面，以便为新的请求服务。当一个脏页被驱逐并被修改后，在从缓冲池中删除之前，它需要被刷新到存储空间，以确保耐久性。这个过程必须与事务和日志控制器仔细协调，也就是说，只有当所有未提交的修改的撤销信息被保存在日志文件中时，页面才能被刷新（否则崩溃会导致数据损坏）。此外，刷新一个页面需要是失败的原子性的。在崩溃之后，恢复组件需要一个一致的页面快照。

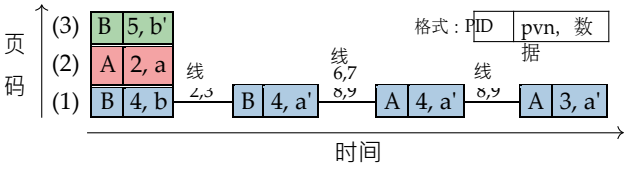
冲洗页面到持久性存储是一个固有的I/O绑定任务。为了减少页面请求的延迟，缓冲区管理器不断地将脏的页面刷到后台的持久性存储中。这样一来，它就可以一直为请求提供服务，而不需要先刷新一个页面。此外，这使得刷新页面（在后台线程上）成为一个主要的带宽关键问题（相比之下，写入日志的延迟是最重要的）。

对于SSDs/HDD来说，这种架构是绝对必要的，因为在CPU可以读或写之前，页面必须被复制到DRAM中。当使用PMem时，缓冲池就变得可有可无。
然而，正如最近的工作[6，33]所显示的那样，使用以下方法仍然是有益的

此外，这种架构被用于大多数现有的基于磁盘的数据库系统中。为了将PMem集成到现有的系统中，页面刷新算法需要正确（故障原子性）和高效（高带宽）。在下文中，我们描述了两种针对故障的算法

原子页刷新，然后评估它们。

3.2.1 写时复制。 **CoW**不会覆盖原来的PMem页，而是将DRAM页写入一个未使用的PMem页[5]（清单1的左侧，第1-3行）。一旦新的PMem页被持久化，它就被标记为有效（第5-10行），旧的PMem页可以被重新使用。在恢复过程中，所有PMem页面的标题被检查，以确定每个逻辑页面的物理位置。通过添加一个在每次刷新后增加的页面版本号（*pvn*），我们可以确定一个页面的最新版本。使用*pvn*，在写入新的PMem页面之前，就没有必要使旧的PMem页面失效了。这就把所需的持久性障碍的数量从三个减少到两个，从而使吞吐量增加10%。我们在下面的例子中说明了*pvn*。



绿色页槽(3)包含B页的最新持久性拷贝，红色的(2)包含A页的原始版本。蓝色页槽((1))的不同版本显示了冲洗A页新版本的每一步，可能发生转换的行号被写在箭头上。在每个步骤中，可以用*pvn*来计算出每个页面的最新版本。在数据库系统中，可以用日志序列号*lsn*来代替*pvn*，但是如果系统在第6行崩溃，日志条目可能会被重新应用到一个页面。

3.2.2 微观日志。 微日志技术使用一个小的日志文件来记录要对页面进行的修改。为了知道，哪些缓存行被改变了，页面需要跟踪自上次刷新以来的修改区域。在恢复过程中，所有有效的微观日志被重新应用，与页面的状态无关。这就迫使我们在改变内容（第5-7行）之前，先使日志无效（清单1的右侧，第1-3行），否则在崩溃的情况下，这些改变会被应用到前一个页面。只有在修改内容被写入后，我们才将其设置为有效（第8-10行），然后将其应用于实际的页面（第13-15行）。

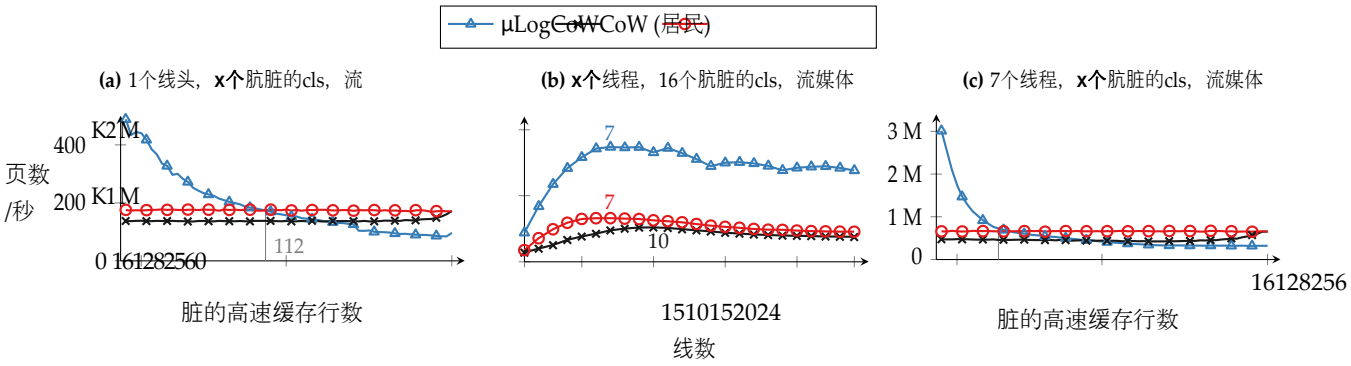


图5：故障原子页刷新 - 以故障原子的方式从DRAM到PMem刷新16 kB的页面（每个256条缓存线）。

清单1：故障原子性--将DRAM页（page_v）冲到PMem页（page_nv）的伪代码。CoW：左边，µLog：右边。

```
1  // 1.写入数据
2  page_nv.data = page_v.data
   。
3  persist(page_nv)。
4  // 2.使PMem页面有效
5  page_nv.pid = page_v.pid。
6  sfence()。
7  page_nv.pvn = page_v.pvn。
8  persist(page_nv.pid
9        , page_nv.pvn) 。
10
11
12
```

```
// 1. 无效的µlog
µlog.pid = INVALID;
persist(log.pid);
// 2. 写入µlog
µlog <-
page_v.dirty_cls
persist(µlog);
// 3.设定µlog有效
µlog.pid =
page_v.pid;
persist(µlog.pid);
// 4. 写入页面
page_nv <-
page_v.dirty_cls
persist(page_nv);
```

*3.2.3 实验。*图5详细介绍了页面刷新性能。所有的技术都是使用流式（也称为非时间性）写入的微观基准来实现的，这在第2节中已经被证明可以提供最高的吞吐量。当使用 "写时复制 "时，我们区分了DRAM中是否有所有的缓存线（）或只有脏的缓存线（）。作为一个性能指标，我们选择了每秒可以刷新到PMem的页面数量。在**(a)**中，我们改变了单线程和**(c)**中7个线程的脏缓存线的数量。在**(b)中**，我们改变了线程的数量，以显示扩展的行为。

结果显示，当需要刷新的缓存线的数量较少时，微型日志是有效的。我们可以在（**a**）中观察到单线程的这种效果。使用micro log可以为多达112个脏的缓存行带来性能提升。一个多线程的

实验显示在（**c**）。在这里，当少于32条高速缓存线被弄脏时，微型日志只提供了吞吐量。因此，基于简单成本模型的混合技术应该被用来选择更好的技术，这取决于脏的数量。
缓存线（和单/多线程）。

第2节中的微观基准测试表明，流式指令应该比常规存储更受欢迎。我们能够在页面刷新实验中确认这一发现（未在图表中显示）。此外，在带宽实验中，我们可以看到当使用太多的线程时，性能会下降。为了获得最佳的吞吐量，重要的是要根据系统的情况来调整写作者线程的数量。如**(b)**所示，性能在达到7-11个线程左右的峰值后就会下降。

3.3 伐木

在数据库系统中，写前日志被用来确保交易的原子性和耐久性。这是通过记录

(记录)一个较大的事务的单个变化，以便在回滚时能够撤销它们。如果对数据的任何改变在交易仍在进行时被持久化，那么该日志也必须被持久化。在一个事务完成之前（从而向用户保证，该事务的所有变化都是持久的），该事务的所有日志条目都被持久化地写入。日志允许数据库只持久化修改的delta。例如，考虑向一个以B-Tree形式存储的表插入。使用日志，只有被改变的数据需要被持久化，而不是所有被修改的节点（页）。在重启过程中，恢复组件会读取日志文件，确定最新的完全持久化的日志条目，并将日志应用到数据库中。

登录构成了数据库的一个主要性能瓶颈

使用传统存储设备（SSD/HDD）的系统，因为每个事务必须等待记录其变化的日志条目被写入。作为一种缓解措施，减少了一致性保证，并实施了复杂的组提交协议。然而，使用PMem，可以实现一个低延迟的日志协议，在很大程度上消除了这个问题。

*3.3.1 算法。*在下文中，我们首先解释，然后评估三种日志技术。*经典*、*头*和*零*。

经典代表了数据库系统中常用的一种记录形式[32]。下面的列表显示了伪代码中的算法（左边）和文件布局语法（右边）。为了清晰起见，只描述了与协议相关的信息。

```
log << header << payload
persist(log);
log << footer
persist(log);
```

```
LogFile -> Entry*
Entry -> header ≺
         有效载荷页脚
```

一个日志条目分两步刷新。首先，标题和有效载荷被附加到日志中并持久化；其次，页脚包含了一份日志序列号（lsn；给每个日志条目的一个id）。页脚中的lsn可以在恢复过程中用来确定一个日志条目是否被完全写入，因此应该被认为是有效的并应用到数据库中。请注意，它需要两个持久性障碍。如果没有第一个屏障，即使在PMem中存在页脚，由于刷新被重新排序，有效载荷的部分也可能被丢失。

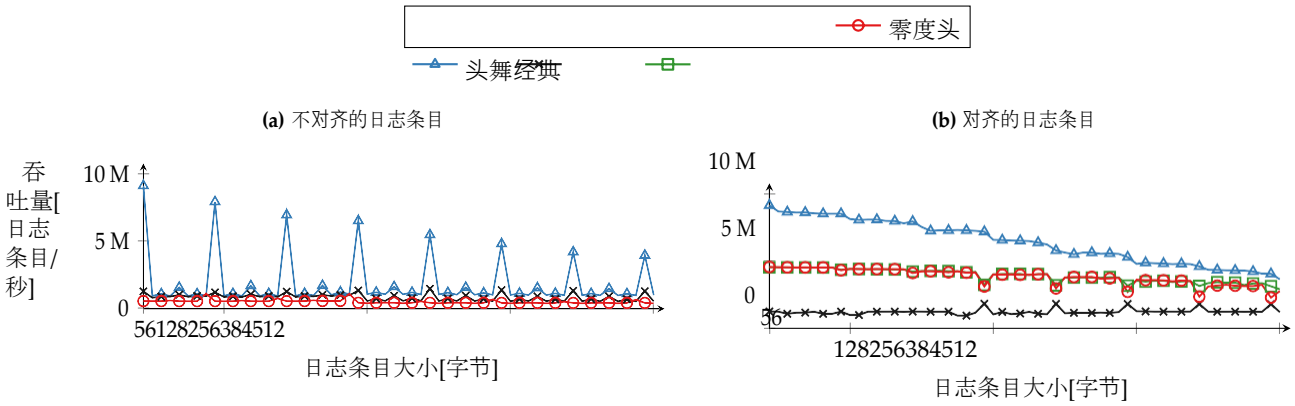


图6：交易日志 - 将不同大小的日志条目写入PMem的吞吐量。

Header使用了与PMDK[1]中*libpmemlog*相同的技术。它类似于将元素追加到数组中。

```
log << header <<
payload persist(log);
log.size +=
entry_size persist (
log.size) 。
```

```
LogFile -> size,
Entry* Entry -> header
payload
```

日志条目也是分两步编写的。首先，标题和有效载荷被附加到日志的尾部并持久化。接着，在日志文件的头部设置新的日志大小并持久化。这样就不需要在恢复过程中扫描日志文件以寻找最后的有效内容，因为有效的大小直接存储在文件头中。

Zero是我们为PMem提出的一种新技术，只需要一个持久性障碍。

```
cnt = pop_count(header,
payload) log << header< cnt<
payload persist(log);
```

```
LogFile -> Entry*
Entry -> header ≤
pop_cnt payload
```

在日志记录开始之前，每个日志文件被初始化为零。这通常是由数据库系统（例如PostgreSQL）完成的，以确保文件系统实际分配给该文件的页面。当一个日志条目被写入时，设置的位数被计算出来（使用popcnt指令）。接下来，标题、数据和位数（cnt）被写到日志中，并一起保存。使用比特数，总是可以确定一个日志条目的有效性。要么包含比特数的缓存行没有被刷新，要么被刷新。在前一种情况下，该字段包含数字0（因为文件被清零了），该条目是无效的。在后一种情况下，位数字段可以用来确定属于该日志条目的所有其他高速缓存行是否也被刷新。

*3.3.2 实验。*在第2.3节中，我们表明，当同一个缓存行连续坚持了两次时，会有很大的性能损失。如图6所示，这种影响对于延迟关键型系统来说是非常重要的。我们使用了一个微观的测试，测量不同大小的日志条目的刷新吞吐量。左图显示的是一个天真的实现，而右图则是在每个日志条目上使用填充物，将条目与缓存行边界对齐，从而避免对同一缓存行的后续写入。虽然填充会浪费一些内存[6]，但吞吐量却大大增加(8)。

然而，即使有了填充，*经典方法*仍然优于*标题方法*，因为写的速度减慢了。

[6] *Zero*和*Header最多*有1条缓存线；*Classic*最多有2条缓存线

当大小被更新时，就会到头中的同一缓存行。这个问题可以通过使用一个*跳舞的*大小字段来解决。我们使用几个
在头的不同缓存线上的大小字段，并且只为每个日志条目写一个（轮流）。通过使用64个这些跳舞的大小字段，*Header*的吞吐量可以提高到*Classic*的吞吐量。然而，这两种技术仍然需要持久性障碍

因此，不能与"*零*"记录竞争（2更快）。

PMDK[1]的日志实现（*libpmemlog*）使用了与我们天真的*Header*实现相同的方法（因此产生了相同的吞吐量，当锁定被禁用时），没有对齐和跳舞。它的优点是日志文件很密集，可以作为一个连续的内存段呈现给用户。然而，这给用户留下了手动重构日志条目边界的任务。通过移动这个功能

到库中，可以实施一个更好的记录策略，并提高可用性。

为了验证，我们已经将所有的技术整合到我们的存储中

引擎原型HyMem[33]。在一个具有DRAM驻留表、*零*记录、*Header*和*Classic的*单线程上运行写量大（100%）的YCSB基准测试[10]，实现了2M的吞吐量。
每秒1.7M和1.5M的交易。

4 相关的工作

由于PMem最近才发布，这是在实际硬件上进行的两个[17]初步研究之一。我们的工作提出了低级别的优化，而Swanson等人则用各种存储引擎以及文件系统来评估PMem。到目前为止，基于软件或硬件的模拟，或者基于推测的性能特征的仿真，已经被用来评估可能的系统架构[4, 25, 27, 29]。的数量。
持久性索引结构[3, 9, 14, 21, 34, 37]是很大的，Götze等人[15]已经总结了这一点。类似的技术已经被用来直接在PMem上建立存储引擎[5, 24]。这些方法在PMem上使用就地更新，而PMem的性能低于DRAM。因此，一些索引[26, 36]以及存储引擎[2, 8, 11, 18, 19, 22, 23]将PMem作为一个单独的存储层或对恢复组件的扩展[28, 30]。此外，缓冲区管理的架构[6, 20, 33]已经被提出来，以更适应地使用PMem。恢复一直是数据库系统的一个重要（和性能关键的）组成部分[32]。对于数据库特定的日志[7, 13, 16, 31, 35]和文件系统[12]，已经提出了一些设计。

5 结论

在我们的评估中，我们发现了几个有效使用PMem的准则（参考第2.2和2.3节）：(1) 我们不能像在DRAM上那样对缓存行（64字节）进行优化，而必须对PMem块（256字节）进行优化。(2) 在多线程编程中，应该避免在时间上相近的情况下向同一高速缓存行写入数据。(3) 强制将数据从CPU上的缓存中取出（clwb或流）对于高写入带宽是至关重要的。此外，我们还评估了日志和页面传播的算法：（1）我们的日志实验表明，延迟关键的代码应该尽量减少持久性障碍的数量，并避免对同一缓存行的后续写入。(2) 我们的零日志算法将所需的持久性障碍从两个减少到一个，从而使吞吐量翻倍。
(3) 对于冲洗数据库页面，可以使用一个小的日志（μLog）来只冲洗脏的缓存行。引入的I/O原语使用了与PMDK[1]类似的接口，使其具有广泛的适用性。

参考文献

[1] PMDK：持久性内存开发工具包。http://www.pmem.io。Accessed: 2019-03-26.

[2] M.Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M.Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M.Wagle, and T. Willhalm.SAP HANA采用非易失性存储器。*pvldb*，10（12）：1754-1765，2017。

[3] J.Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson.Bztree:用于非易失性存储器的高性能 无闩锁范围索引。*pvldb*，11（5）：553-565，2018。

[4] J.Arulraj and A. Pavlo.如何建立一个非易失性内存数据库管理 ment系统。在 *SIGMOD*, 2017.

[5] J.Arulraj, A. Pavlo, and S. Dulloor.让我们来谈谈非易失性内存数据库系统的存储和恢复方法 。In *SIGMOD*, pages 707-722, 2015.

[6] J.Arulraj, A. Pavlo, and K. T. Malladi.非易失性存储器的多层缓冲区管理和存储系统设计。*arXiv*，2019。

[7] J.Arulraj, M. Perron, and A. Pavlo.Write-behind logging.*pvldb*，10（4）：337-348， 2016。

[8] M.Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang.SSD 数据库系统的缓冲池扩展。*pvldb*, 3(2):1435-1446, 2010.

[9] S.Chen and Q. Jin.非易失性主存储器中的持久性B+树.*pvldb*, 8(7):786-797, 2015.

[10] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears.用YCSB对 云服务系统进行基准测试。In *SoCC*, pages 143-154, 2010.

[11] J.Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson.使用SSD的DBMS缓冲池的涡轮增压。在*SIGMOD*, 2011.

[12] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J.Jackson.用于持久性内存的系统软件。In *EuroSys*, 2014.

[13] R.Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang.使用存储类内存的高性能数据库日志。在*ICDE*，第1221-1231页，2011年。

[14] P.Götze, S. Baumann, and K. Sattler.用于分析性 工作负载的NVM感知存储布局。In *ICDE Workshops*, 2018.

[15] P.Götze, A. van Renen, L. Lersch, V. Leis, and I. Oukid。 非易失性存储器上的数据管理。A perspective.*Datenbank-Spektrum*, 18(3), 2018.

[16] J. Huang, K. Schwan, and M. K. Qureshi.交易中的NVRAM感知日志 系统。*pvldb*，8（4）：389-400，2014。

[17] J.Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang。Y.Xu, S. R. Dulloor, J. Zhao, and S. Swanson.Intel optane DC持久性内存模块的基本性能测量 。*CoRR*, 2019.

[18] W.Kang, S. Lee, and B. Moon.闪存作为在线事务性 工作负载的缓存扩展。*VLDB杂志*，25（5）：673-694，2016。

[19] T.Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner.用英特尔事务性同步 chronization扩展提高内存数据库索引性能。在*HPCA*，2014。

[20] H.木村。FOEDUS:一千个内核和NVRAM的OLTP引擎。在*SIGMOD*， 第691-706页，2015。

[21] S.K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh.WORT：为持久性内存存储系统写出最佳弧度 树。In *FAST*, pages 257-270, 2017.

[22] X.Liu and K. Salem.数据库系统的混合存储管理.*pvldb*, 6(8):541-552, 2013.

[23] T.Luo, R. Lee, M. P. Mesnier, F. Chen, and X. Zhang. hStorage-DB: Heterogeneity-aware data management to exploit full capability of hybrid storage systems.*pvldb*, 5(10):1076-1087, 2012.

[24] I.Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm.SOFORT：一个混合用于快速数据恢复的SCM-DRAM存储引擎。在*DaMoN*, 2014.

[25] I.Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes。大规模持久性内存系统的内存管理技术。*PVLDB*, 2017.

[26] I.Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner.FPTree:用于存储类存储器的混合单片机-DRAM持久性和并发性B-树。在*SIGMOD*， 第371-386页，2016。

[27] I.Oukid and W. Lehner.字节可寻址非 挥发性存储器的数据结构工程。在 *SIGMOD*, 2017.

[28] I.Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis.主内存数据库的即时恢复 。在*CIDR*，2015。

[29] I.Oukid和L. Lersch.论内存和存储技术的多样性。*Datenbank-Spektrum*, 18(2), 2018.

[30] I.Oukid, A. Nica, D. D. S. Bossle, W. Lehner, P. Bumbulis, and T. Willhalm.启用SCM的数据库的自适应恢复。In *ADMS*, 2017.

[31] S.Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge。 NVRAM时代的存储管理。*PVLDB*, 2013.

[32] C.Sauer.*面向交易的数据库恢复的现代技术*。博士论文， Kaiserslautern科技大学，德国，2017。

[33] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato.管理数据库系统中的非易失性存储器。在*SIGMOD*， 2018。

[34] S.Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell.非易失性字节寻址存储器的一致和持久数据结构。在*FAST*中，第 61-75，2011。

[35] T.Wang and R. Johnson.通过新兴的非易失性存储器进行可扩展的记录。*pvldb*，7（10）：865-876，2014。

[36] F.Xia, D. Jiang, J. Xiong, and N. Sun.Hikv:用于 DRAM-NVM内存系统的混合索引键值存储。In *USENIX ATC*, pages 349-362, 2017.

[37] J.Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He.NV-Tree:降低基于NVM的单层系统的一致性成本。In *FAST*, pages 167-181, 2015.