NVAlloc:重新思考持久性内存分配器中的堆元数据管理

郑当
浙江大学 浙江杭州，中国

何水兵
浙江大学 浙江杭州，中国

洪佩仪
浙江大学 浙江杭州，中国

李振新
浙江大学 浙江杭州，中国

Xuechen Zhang 华盛顿
州立大学，美国，温哥华

孙贤和
美国伊利诺伊州芝加哥理工学院

陈刚
浙江大学 浙江杭州，中国

ABSTRACT

持久性内存分配是开发高性能和内存应用的一个基本构件。现有的持久性内存分配器受到次优堆组织的影响，在持久性内存中引入了重复的缓存线刷新和小的随机访问。更糟糕的是，许多分配器使用静态板块隔离，当分配请求的大小改变时，会导致内存消耗的急剧增加。在本文中，我们设计了一个新的分配器，名为NVAlloc，以同时解决上述问题。首先，NVAlloc通过将板块中连续的数据块映射到存储在不同缓存线中的交错的元数据条目，消除了缓存线的刷新。第二，它以顺序模式将小的元数据单元写入持久性记账日志，以消除持久性内存中的随机堆元数据访问。第三，它不使用静态的slab隔离，而是支持slab mor-phing，这使得slab可以在不同大小的类别之间转换，以显著提高slab的使用率。NVAlloc是对现有一致性模型的补充。6项基准测试的结果表明，NVAlloc将最先进的持久性内存分配器的性能提高了6.4倍和57倍，分别用于小型和大型分配。使用NVAlloc可以减少多达57.8%的内存使用。此外，我们将NVAlloc集成到一个持久的FPTree中。与最先进的分配器相比，NVAlloc将该应用的性能提高了3.1倍。

CCS概念

· **软件及其工程→分配/去分配策略**； **–硬件→非易失性存储器**
。

[‡]何水兵是通讯作者。

ASPLOS '22，2022年2月28日至3月4日，瑞士洛桑。

https://doi.org/10.1145/3503222.3507743

关键字

动态内存分配、持久性内存、内存碎片化

ACM参考格式。
党正，何水兵，洪培义，李振新，张学臣，孙宪和，陈刚。2022.NVAlloc:重新思考持久性内存分配器的堆元数据管理。在*第27届ACM编程语言和操作系统架构支持国际会议（ASPLOS '22）论文集，2022年2月28日至3月4日，瑞士洛桑*。ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507743

1 简介

持久性内存的动态分配被大量用于建立高性能的应用，从索引结构[8, 23, 24, 26-28]，事务性内存[13, 19, 20, 39]，到内存中的应用。数据库系统[1, 10, 25, 31]。内存分配器通常针对易失性内存（如DRAM）进行良好的调整，以实现低延迟、高扩展性和低碎片化[2, 17, 33]。持久性内存（如英特尔Optane DIMMs[11]）的采用使研究人员重新思考分配器的设计和实现。为持久性内存设计的分配器需要保持DRAM分配器的突出特点，以实现高性能内存管理。更重要的是，它们应该强制执行崩溃一致性，以便在故障发生后能够安全地恢复分配的内存对象。

许多分配器都是为持久性内存设计的[3, 4, 15, 30, 31, 37]。他们需要通过各种类型的元数据（例如，对象位图、板块结构、程度头和写头日志）来管理持久化的堆，以便有效地服务于内存分配和卸载。在本文中，我们把它们称为*堆元数据*。例如，PMDK[11]和nvm_malloc[37]使用位图来标记已分配的目标。PAllocator[31]使用日志来强制执行堆元数据的碰撞一致性。更新堆元数据会触发对持久性内存的频繁的小规模写入，范围从1比特到64B。所有这些持久性分配器都使用大小分离的算法来提供小规模分配请求，以减少内存碎片。

现有的为持久性内存设计的分配器有很多与堆元数据管理有关的问题。首先，小

对堆元数据的写入可能会导致高速缓存行的刷新。CPU缓存行的典型大小是64 B [31]。在nvm_malloc中，一个位图的大小是8 B。当位图被反复更新时，同一高速缓存行应该被刷新以保持持久性。缓存行刷新的延迟比写的延迟高7.5倍[7]。我们观察到，在四个著名的基准测试中，缓存行刷新的数量占到分配器引起的刷新操作总数的40.4%~99.7%（第3.1节）。频繁的缓存行刷新操作导致持久性内存分配器的性能下降。

其次，分配器的堆元数据往往在持久性内存中被随机访问。许多分配器（例如PMDK、PAllocator和Makalu[3]）将堆划分为固定大小的块（例如4MB），以方便管理。他们在每个块的头空间维护记账元数据，该空间与数据空间分开，以避免元数据被错误地修改。这种布局导致头被分布在整个堆空间中。在提供一连串的分配和取消分配请求后，分配者必须就地更新随机位于持久性内存中的头文件。最近的工作表明，持久性内存的随机访问性能比小写的顺序访问性能要差得多[39, 40]。因此，向堆元数据提供这些小的随机写入，使分配器无法实现最佳性能。

第三，静态板块隔离导致持久性内存碎片化。这个问题在持久性内存中更加严重，因为持久性堆是以文件的形式存储在DAX文件系统中的。它们不能通过重新启动系统来消除。所有持久性内存的分配器都使用大小分离的板块来分配小块。每个板块是一个多个空闲块的容器，处理一个特定大小类的内存分配。分配给一个大小类的板块不能再用于其他大小类，即使这些板块大部分是空的，而且其他大小类的板块中也没有空闲空间[38]。对于分配大小不断变化和频繁的 "删除 "操作的工作负载来说，这种隔离引起的碎片化增加了高达2.8倍的内存使用率（第3.2节）。

在本文中，我们介绍了一个快速和故障安全的持久性内存分配器，名为NVAlloc。它的设计强调有效地消除缓存线刷新和小的随机写入，并减轻堆元数据管理中的板块引起的内存碎片。首先，NVAlloc使用从数据块到其相应的堆元数据的交错内存映射，以及线程本地缓存中链接列表的交错布局，以避免重复访问同一CPU缓存行。其次，由于原地元数据更新会导致持久性内存中的随机访问，而写缓冲区的大小是有限的[40]，我们增加了一个持久性的记账日志，以顺序模式存储小的元数据更新。因此，我们从malloc()和free()的关键路径中完全移除随机元数据访问。第三，它支持slab morphing，在slab变换过程中，两个大小级别的块可以共同位于一个slab中。因此，低内存使用率的板块中的自由空间可以得到很好的利用，板块元数据管理的运行时间开销为4.5%。当一个板块大部分时间是空闲的，但不能用来为其他大小类别的请求提供服务时，板块变形是自动启用的。

NVAlloc目前同时支持基于日志和基于垃圾收集的崩溃一致性模型[1]。6项基准测试的结果表明，NVAlloc将最先进的持久性内存分配器的性能提高了6.4倍，对于小的分配，提高了57倍。使用NVAlloc可以减少多达57.8%的内存使用。我们还将NVAlloc集成到持久化的FPTree中[32]。与最先进的分配器相比，使用NVAlloc，该应用的性能提高了3.1倍。

2 背景情况

2.1 术语

我们首先定义持久性内存分配器中的常用术语。**Extents**是直接从持久性堆空间分配的连续字节序列，用于满足大型分配请求。**板块**是持久性内存中预先分配的外延，是固定大小的自由块的容器。本文中的板块大小为64KB。小的分配是根据它们的大小类别使用slab来提供的。**块**是持久性内存中连续的字节序列，由slab结构分配，用于满足小型分配请求。**slab位图**位于slab头文件中，每个位表示slab块的状态（分配或释放）。**堆文件**是驻留在DAX文件系统的持久性内存中的文件，被映射为持久性堆。

线程本地缓存（tcache）跟踪由本地空闲请求组成的独特的空闲块列表的地址，这些请求可能来自多个板块。当一个分配器收到一个请求时，它首先搜索tcache以满足该请求。当一个区块被释放时，它将进入释放它的线程的tcache，而不是之前分配它的那个线程。我们使用LIFO算法来管理tcache。当tcache为空时，它被重新填入来自slab的区块地址。

写入日志（WAL）[31, 37]是用来记录持久性内存分配器在使用跨行动进行故障安全恢复时对堆元数据/数据的改变。WAL条目被设计用来保存重要的元数据（例如，内存地址和当前值）。

2.2 持久性内存中的堆管理

小型分配。Slabs被广泛用于小型分配（例如，< 16 KB），以减少内存碎片。我们实现了一个新的slab结构，用于持久性内存中的小型分配，利用了现有slab结构（即jemalloc [17]和nvm_malloc [37]中的设计原则）。具体来说，每个板块都有一个持久化头和一个易失性头（称为*vslab*）。持久头存储了恢复所需的元数据，包括一个位图，其位被按顺序映射到下面的块。易失性*vslab*用于快速搜索空闲块。它可以在故障恢复期间被重建。

大型分配。分配器也需要管理大型分配（例如，16KB）。我们用jemalloc中的类似结构作为例子。在DRAM中使用虚拟范围头（VEHs）来管理Extents，以便有效地搜索、分割和凝聚堆的Extents。在NVAlloc中，有三个列表被用来管理VEHs。一个*激活的列表*存储已分配扩展的VEHs。一个*回收列表*存储已释放的具有物理持久性内存的外延的VEHs。

[1] NVAlloc的源代码可在https://github.com/ISCS-ZJU/NVAlloc。

表1：Fragbench中的工作负载配置。

工作量	之前	删除	之后
W1	固定100B	90%	固定130B
W2	统一的100-150 B	0%	统一的200-250 B
W3	统一的100-150 B	90%	统一的200-250 B
W4	统一的100-200B	50%	统一的1000-2000 B

被映射到虚拟地址。而一个*保留的列表*则存储了
只分配了虚拟地址的空闲Extents的VEH，其物理内存在进程地址空间中已被取消映射。

在提供一个大的分配时，分配器使用第一适合算法搜索回收列表和保留列表。如果找到一个区块，它的VEH将被移到激活列表中。如果没有找到，就会创建一个新的VEH并添加到激活列表中。当一个区块被释放时，它将被返回到回收列表中。NVAlloc使用一种基于衰减的方法来管理回收列表和保留列表中的空闲扩展[17]。它使用一个*smootherstep*函数来计算列表可以使用的最大内存量$THmax$。如果回收列表的内存使用量高于$THmax$，它的extents将从回收列表移到保留列表中。同样地，如果保留列表的内存使用量高于其阈值，其外延将被移至操作系统。当一个VEH被从保留列表中移除时，其对应的extent在进程地址空间中被取消映射，其头和extent在持久性内存中被释放。现有的工作中也使用了类似的方法（例如jemalloc）。我们使用与jemalloc中设置的相同的*smootherstep*函数参数和时间间隔（即50ms）。

3 激励

我们通过实验研究了现有分配器中糟糕的堆元数据管理所引起的性能问题和内存碎片。我们使用不同的应用程序来产生工作负载，暴露出各种内部问题。

3.1 分配器引起的高速缓存线重新刷新

当反复将同一个CPU缓存行冲到持久性内存中以实现堆数据/元数据的持久性时，就会发生缓存行重新刷新。缓存行重新刷新的延迟由两次访问同一缓存行之间的重新刷新距离决定。当获取持久性内存成为分配器的性能瓶颈时，我们可以将重新刷新的距离量化为访问独特缓存行的次数。例如，给定一个连续刷新的缓存行序列（A，B，C，D，A），缓存行A的重新刷新距离为3。我们的实验表明，当重新刷新距离从0增加到3时，缓存行重新刷新的延迟从800 ns减少到500 ns。 在本文中，我们假设当一个缓存行的重新刷新距离小于
4.否则，就会发生一次常规的刷新。我们选择4作为代表性的刷新距离，因为我们观察到大多数缓存行的刷新距离都小于4，而且较大的距离会导致较小的性能下降。缓存行重新刷新的平均延迟比持久性内存中的随机和顺序写入分别高3倍和7倍[7]。

为了研究分配器引起的缓存行刷新数量，我们运行了四个著名的基准测试，包括*Threadtest*、*Prod-con*。

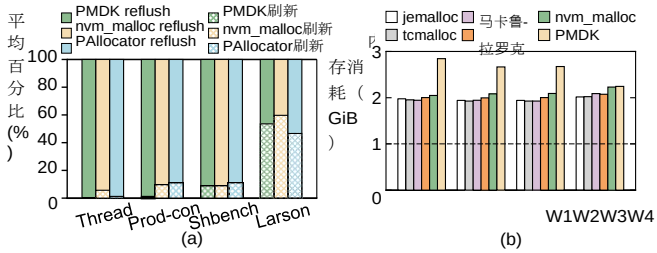


图1：(a)缓存线刷新的比率；(b)峰值内存消耗。

*Shbench*和*Larson*。实验设置的细节将在第6节介绍。图1(a)显示了高速缓存行刷新和常规刷新的百分比。我们观察到，在运行PMDK、nvm_malloc和PAllocator时，缓存行的重新刷新次数分别占到了刷新操作总数的99.7%、94.4%和98.8%。这是因为它们会安全地更新slab headers或WALs中的小元数据对象，或者两者都更新，以保持强一致性。这些缓存行的刷新减缓了分配和去分配的操作。

3.2 静态板块隔离引起的破碎现象

对于分配小对象，板块被广泛用于现有的分配器中，包括易失性内存分配器（如jemalloc-）。
5.2.1 [17] 和 tcmalloc-2.9.1 [18]）和持久性内存分配器（例如Makalu [3], Ralloc [4], nvm_malloc [37] 和 PMDK-1.11 [11]）。板块是根据尺寸等级来隔离的。当一个板块被初始化时，大小类被确定，并且在运行时不能被改变。然而，在服务器应用程序的执行过程中，内存分配的请求大小是变化的[35, 38]。我们运行fragmentation基准测试[35]（我们称之为Fragbench）来研究流行的分配器的内存使用。Fragbench有三个执行阶段。*之前*，*删除*，和*之后*。在*之前*和*之后*阶段，Fragbench使用预先定义的大小分布中的对象分配5GB的内存，并随机删除现有的对象，以保持实时数据量不超过1GB。在*删除*阶段，Fragbench随机地删除了对象。这三个阶段是按顺序执行的。我们改变了对象的大小分布和删除对象的比例，在四个有代表性的工作负载[2]（W1-W4，如表1所示）中，这些工作负载来自于基准，以涵盖现实世界应用的广泛特点。类似的工作负载已被用于先前的研究（即RAMCloud[35]、PAllocator[31]和日志结构的NVMM[20]）。

图1(b)中显示了峰值内存消耗。对
管理1GB的活堆数据，现有的分配器需要的内存用量高达2.8GB。这个结果表明持久性内存的利用率严重不足。原因是现有分配器中使用的静态板块隔离，通过分配更多的其他大小类别的板块来响应请求大小的变化[21]。它不能使用不同大小等级的现有板块中的自由空间。这是因为分配器在运行时不能改变一个板块的大小类别，直到它完全空闲。由以下原因引起的内存碎裂

[2] 虽然原始的Fragbench中有八个工作负载，但我们只选择了四个工作负载，因为其他工作负载显示了类似的问题，而且空间有限。

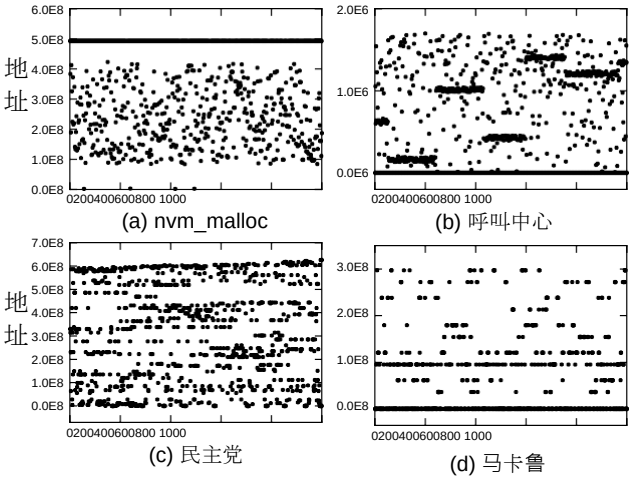


图2：持久性内存中的小型随机写入。X轴表示刷新的次数。

持久性内存中的静态板块隔离比易失性内存中的影响更大，因为内存碎片不能通过重新启动来消除。

3.3 对于大型分配，大多数现代分配器（如PMDK和Makalu）在大的头空间中存储记账元数据。

内存区域（例如，4MB）。簿记元数据跟踪该区域的所有扩展。头部空间通常被放置在一个专门的位置，与堆数据空间分开。这种布局避免了头空间被用户错误地修改。更新头空间中的元数据（例如位图和日志）需要对持久性内存进行少量的写入。为了研究它的访问模式，我们在使用4个分配器（包括nvm_mallocator、PAllocator、PMDK和Makalu）运行DBMStest基准[16]时，对元数据的前1000次刷新操作的内存地址进行了分析。我们在图2中显示了结果。我们观察到，为了管理记账元数据，分配器向持久性内存发出了大量的小型随机写入，请求地址分布在整个堆空间中。原因是，为了服务于一个大的请求，分配器通常使用分配算法（即，最佳匹配，第一匹配，或它们的变体）来找到一个最合适的范围。之后，他们就地更新头空间中的记账元数据。在一连串的分配和删除之后，最佳的候选范围可以位于堆空间的任何内存区域，从而导致更新其记账元数据的小型随机访问。

4 虚无缥缈

在本节中，我们介绍了NVAlloc的编程模型，并描述了其主要组件的设计：小型分配器和大型分配器。NVAlloc软件是用三种优化方法开发的，包括交错映射，减少了缓存行的刷新；板块变形，缓解了隔离引起的碎片；以及日志结构的簿记，改善了写入的定位。我们在图3中说明了NVAlloc的所有组件和每个优化的应用。

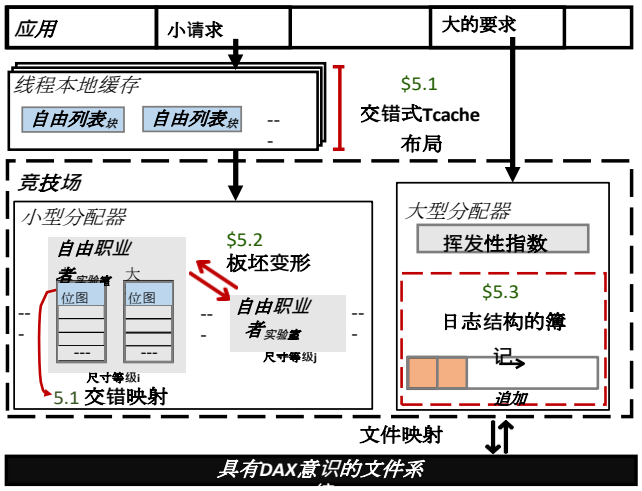


图3：NVAlloc的概述。

4.1 编程模式

我们使用nvalloc_init()来创建一个新的NVAlloc实例，使用nvalloc_exit()来安全退出。为了避免内存泄漏，我们采用了其他分配器中使用的nvalloc_malloc_to()和nvalloc_free_from()API[11, 31, 37]来分别原子化地分配和释放持久性内存上的对象。函数nvalloc_malloc_to()根据用户指定的*size*在持久化堆中分配一个块或一个程度，并将其持久化地附着在用户指定的*address*。我们使用基于偏移量的指针表示，以允许持久性结构在故障恢复时被映射到不同的虚拟地址。同样的技术已经在以前的项目中使用过[4, 6, 9]。nvalloc_free_from()返回一个由*addres*指定的块或范围到持久化内存堆。

目前，我们实现了两个变体。NVAlloc-LOG支持基于日志的事务模型，NVAlloc-GC支持基于GC的模型（见第7节）。作为未来的工作，我们希望实现另一个变体，使用内部收集进行故障恢复。我们将这三种新的优化技术应用于这些分配器中的各种组件。作为一个例子，我们将交错映射应用于NVAlloc-LOG的WAL、记账日志、位图和tcache。表2显示了细节。

对于小型分配，NVAlloc-LOG将所有元数据更新写入WALs，并将其刷新到持久性内存，以强制执行一致性。所有的内存泄漏都可以通过重放WALs来解决。在NVAlloc-GC中，对于小型分配不使用元数据或WALs刷新，以实现最佳运行时性能。然而，它需要在恢复期间执行崩溃后的GC，以重建堆元数据和检查内存泄漏，这就阻碍了应用程序的正常执行[3]。对于大型分配，NVAlloc-GC的代码路径与NVAlloc-LOG相同。

4.2 小型分配器

对于小的分配（< 16 KB），NVAlloc实现了*arena*和*tcache*，以减少线程的竞争。每个CPU核拥有一个竞技场，而每个线程拥有一个tcache。每个线程将被分配到一个分配线程数量最少的竞技场。一

ASPLOS '22，2022年2月28日至3月4日，瑞士洛桑。

个竞技场

党正，何水兵，洪佩仪，李振新，张学臣，孙宪和，陈刚

表2：NVAlloc的两个变体中使用的技术（IM指交错映射）。

拨号器	小额拨款	大额拨款
NVAlloc-LOG	IM(WAL,bitmaps,tcache) 板块变形	IM(WAL,记账日志) 日志结构的簿记
NVAlloc-ÃĂĂ	板块变形	IM(WAL,记账日志) 日志结构的簿记

保持一个板块的自由列表（$f\ relist\ \ _{slab}$)的每个尺寸等级。freelists中的板块是部分满的。一个tcache为每个大小类维护一个块的freelist（$_{fblock}$）。自由列表中的每个区块都可以为分配服务。

当一个特定大小的小块被请求时，工作线程会得到它的大小类，然后尝试从tcache中相应的$_{fblock}$，得到一个块。如果$_{freel}$是空的，工作线程将使用其对应的$frelist_{slab}$在竞技场中填充它直到满。这里需要线程同步，因为多个线程可能连接到同一个竞技场。如果在$frelist_{lab}$中没有slab，它将首先使用slab morphing（第5.2节）来寻找其他尺寸类别的块来填充tcache。当使用slab morphing找不到块时，它将通过执行一个大的分配来要求一个新的slab。一旦$_{fblock}$被填满，用户可以立即从tcache中检索块。

当用户释放一个小块时，工作线程将首先使用R-树来寻找它的大小类。然后，将其返回到其对应的tcache中。当$_{fblock}$满时，工作线程将绕过tcache，直接将小块返回到其slab。

NVAlloc使用slab位图的交错映射和tcache的交错布局（第5.1节），以避免在需要小的堆元数据访问时对cache行进行刷新。

4.3 大型分配器

NVAlloc中的大型分配器负责分配从16KB到2MB的板块和扩展。对于大于2MB的对象，NVAlloc调用mmap()来分配一个给定大小的范围。大型分配器的结构如图7所示。当nvalloc_malloc_to()被调用时，它首先使用最佳匹配算法搜索回收的列表。如果没有找到，就用保留列表重复搜索。如果找到了一个extent，它的虚拟extent头（VEH）会被移到激活列表中。如果现有的范围大于请求的大小，可能需要对该范围进行分割。为了便于分割和凝聚，NVAlloc在DRAM中维护了一个R树，以帮助搜索相邻的extents。R-树中的每一项都是一个键值对，其键是一个extent的开始/结束地址，值是一个指向其相应VEH的指针。如果在回收列表或保留列表中没有可用的extent，NVAlloc调用mmap()来分配一个4MB的新extent，它被分成两部分。NVAlloc将第一部分返回给用户，并将其添加到激活列表中。第二部分被添加到回收的列表中。最后，对于每个部分，NVAlloc在R树中添加一个项目，指向该部分的VEH。

当调用nvalloc_free_from()来释放一个大的内存区域时，NVAlloc使用其内存地址在R树中搜索其VEH。该VEH被从激活列表中移到回收列表中。NVAlloc使用一种基于衰减的方法来管理R树中的VEH。

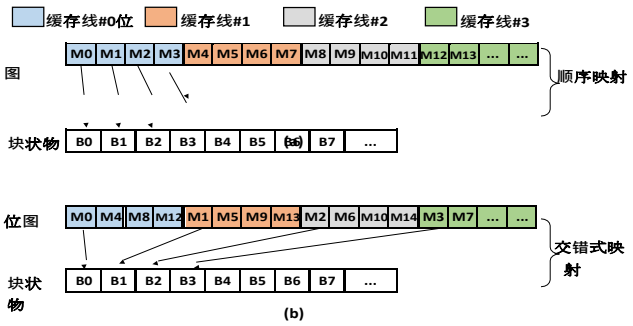


图4：将位图映射到数据块。

回收列表和保留列表（见第2.2节）。对于故障恢复，当VEH被创建或更新时，其基本元数据被添加到持久性记账日志中。持久性记账日志的操作在第5.3节描述。

4.4 恢复

在恢复过程中，分配器必须确保没有持久性内存泄漏，并且分配器的元数据是一致的。然后，应用程序可以再次在持久性堆中进行正常的分配和去分配。

我们使用一个每个竞技场的标志来标记竞技场的状态，包括*运行*、*正常关闭*和*恢复*。当nvalloc_exit()完成后，我们将该状态改为正常关闭。如果恢复过程中发现该标志是*运行*或*恢复*，则表明在运行或恢复过程中发生了故障。在这种情况下，我们需要做一个额外的理智检查以确保一致性。

对于正常的关机恢复，我们首先为每个CPU核重新创建一个竞技场，然后打开并映射它们各自的堆文件和日志文件。之后，对于每个竞技场，我们对持久性记账日志进行缓慢的GC，以清理其墓碑条目（见5.3节）。然后，我们扫描并处理每个日志条目。具体来说，对于每个日志条目，我们首先检查其类型，以确定其对应的范围是否是一个板块。对于板块，我们根据板块头中的元数据重建其易失性$vslab$，并将其加入$_{freel}$。接下来，我们读取它的$f\ \ lag$字段，以确定在正常关闭时，板块是否正在变形。如果它是一个$slab_{in}$（见第5.2节），我们将重建其$_{block}$，并且cnt_{slab}另外。对于正常的外延，我们重建它们的VEH，并将它们添加到激活列表中。我们还将活动扩展之间的空间间隙视为自由扩展，并将其VEH插入到DRAM中的回收列表。

在故障恢复时，我们首先进行正常关机恢复。然后，我们另外使用不同的方法来进行内存理智检查，根据分配器的一致性模型来解决可能的内存泄漏问题。对于NVAlloc-LOG，我们像在nvm_malloc中一样重放WALs。对于NVAlloc-GC，我们像在Makalu中一样进行保守的垃圾收集[3]。对于板块，我们将读取板块头中的$f\ \ lag$字段，以识别在板块变形过程中是否存在故障。如果检测到故障，我们将撤销元数据转换的所有操作。

5 优化元数据管理

在这一节中，我们介绍了三个优化，解决了持久性内存分配器中的元数据管理问题。

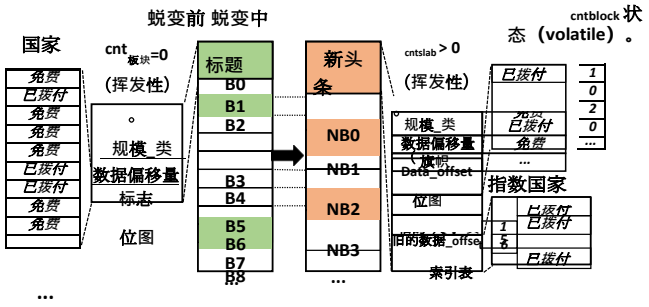


图5：板块变形的说明。

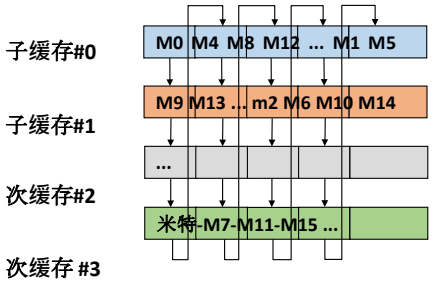


图6：交错式tcache布局。

5.1 交错式映射

使用板块结构，来自同一板块的连续的小分配需要更新板块位图中的连续位。因为这些位很可能被存储在一个CPU缓存线中，它可能会导致分配器引起的重复缓存线刷新，导致更长的请求延迟。一个天真的方法是在随机偏移处分配块。因此，多个高速缓存线可以以随机的方式被访问，避免了对同一高速缓存线的重复刷新。然而，这种方法损害了持久化堆中块的空间定位。在以前的工作中使用的另一种方法[3, 4]是用一个链接列表而不是位图来管理板块中的空闲块。每个空闲块都有一个嵌入式链接指针。这种设计有三个问题。首先，在分配的数据块之前放置一个头，很容易造成内存损坏的元数据损坏[15]。第二，链接指针的大小比缓存行的大小小得多。当链接指针和它们对应的数据块被存储在同一个缓存行中时，分配器引起的刷新仍然是可能的。第三，tcache中的块仍然可能被映射到同一高速缓存行中。因此，现有的工作都没有完全解决这个问题。

我们设计了一个两级交织方案，以产生一个元数据布局，在保持块的空间定位的同时，消除了高速缓存行的刷新。

板块位图的交错映射。假设我们有一个位图，它总共有N个比特。我们把位图分为位条，每个位条被映射到一个高速缓存行。条带大小d是一个条带中的比特总数，并以缓存行大小为上限。然后我们以交错的方式将连续的块映射到不同条带的位上。我们用图4来说明。在这个例子中，我们假设比特条带的数量是4。 在基线中，比特被依次映射到数据块上。例如，位$M0$、$M1$和$M2$分别被映射到数据块$B0$、$B1$和$B2$。由于分配器需要在每次分配时坚持位图以保证崩溃的一致性，对$B0$、$B1$和$B2$的连续分配导致重新刷新存储位$M0$、$M1$和$M2$的同一高速缓存行。在交错映射中，$M0$、$M1$和$M2$被放置在不同的位条和高速缓存行。因为$M0$、$M1$和$M2$分别存放在缓存线#0、#1和#2中，所以当$B0$、$B1$和$B2$被分配到板块中时，不会出现缓存线刷新。

tcache的交错布局。当使用tcache时，块的分配顺序是由管理tcache的LIFO算法决定的。因此，如果tcache选择的块的位被映射到同一个缓存行，仍然有可能出现缓存行刷新的连续分配。为了避免缓存行刷新问题，我们设计一个新的交错式高速缓存布局（如图6所示）。具体来说，我们把一个高速缓存分成多个子缓存。子缓存的数量是由位条的数量决定的。每个子缓存都会缓存那些对应位被映射到同一缓存行的块的地址。我们维护一个游标来指示哪个子缓存被用于当前的分配。该游标在一次分配后指向下一个子缓存，这确保了映射到不同缓存行的子缓存被用来服务于连续的分配。例如，假设tcache被填满了与图6中0到15位对应的块。因为tcache从4个子缓存中交替选择块来服务于连续的小分配，我们可以保证tcache不会选择映射到同一缓存行的位。因此，缓存线的刷新被有效地消除了。

5.2 板坯变形

现有的分配器使用静态板块隔离来管理板块，导致了内存碎片化。我们设计了一种新的技术，名为slab morphing，来解决这个问题。这个想法是，当一个板块的内存使用量较低时，NVAlloc允许它转变为另一个大小等级的板块。在转换过程中，该板块可能会存储两种不同大小的数据块。在板块变形的设计中，我们需要解决两个挑战。(1) 该方案需要保证对属于不同大小类的两类块进行索引的正确性。(2) 我们需要尽量减少管理这些块的开销。

使用slab morphing进行区块分配。我们使用LRU列表来管理所有的板块。最近访问次数最少的板块被放在列表的首位。只有当一个小的对象请求来了，但请求大小类的现有板块没有空间时，才会启用板块变形。NVAlloc将选择一个板块进行变形并转换其元数据。

*选择一个候选板块进行变形。*NVAlloc从头到尾扫描LRU列表，并选择一个板块进行变形，当其$ooccup\ y$低于空间利用率的阈值（SU），其中$_{ooccup\ y}$定义为分配的块数与板块中总块数的比率。在其目前的设计中，我们将SU设定为20%（见第6.5节）。因为板块变形需要改变板块头的格式，如果新的头空间与有活数据的块空间重叠，那么板块将不会被选中。

*改造板块元数据。*然后，NVAlloc需要重置所选板块的元数据。为了讨论的方便，我们把变形前、变形中和变形后的板块称为$sla_{bbe\ for}$。

分别是*slabin*和*slabafter*；我们指的是分配的区块。在*板矿石*作为*封锁的ef*.*板坯矿石*和*板块前三矿石*是正规的slabs，其头部由一个*size_class*字段、一个*data_offset*字段（数据区域的起始地址相对于slab的起始地址的偏移）和其*bitmap*字段组成。*Slabin*需要支持对两个大小类的块进行索引。因此，我们增加了额外的元数据来帮助实现这一功能。具体来说，我们在*slabin*的头部添加了一个*old_size_class*字段和*old_data_offset*字段，以支持*blockbefore*的索引。我们还添加了一个*index_table*来确保*blockbefore*的可恢复性。每个*块before*有一个索引表项，它存储了它在*slabbefore*中的块索引和它的分配状态（即，分配或释放）。索引表的内存占用很小，因为（1）每个表项只有2B，（2）我们只有有限数量的*块before*，因为我们只在板块使用率低时选择板块进行变形。

最后，我们在易失性头文件*vslab*中添加了一个计数器*cntslab*，以确定注意板块中被分配的*块状资产的*数量。如果*cntslab*>0，该板块就是一个*slabin*，否则就是一个普通板块。我们还在易失性内存中为*slabin*中的每个块维护一个计数器*cntblock*，以表示占用该块的*块before的*数量。*cntblock*用来表示一个新的区块是否可以被安全释放。

我们按以下步骤转换元数据。步骤1：设置*old_size_class*和*old_data_offset*；步骤2：设置*index_table*；步骤3：设置新板块头中的*size_class*、*data_offset*和*bitmap*。由于板块转换涉及到元数据的多个修改步骤，我们添加了一个*标志*字段来指示转换的步骤，以确保碰撞的一致性。对于*slabin*和*slabafter*，标志被设置为0。在转换过程中，我们在每一步之后将*flag*原子化地增加1。当我们在*old_size_class*、*old_data_offset*和*index_table中*拥有一个拷贝后，*位图*中的*size_class*、*data_offset和*分配信息将被改变。如果在转换过程中发生崩溃，我们可以使用*flag来*撤销变形，*flag表示*哪一步已经完成。在元数据转换之后，*slabin*被从LRU列表中移除，因为它不能再变形了。它也被从*old_size_class*的slab列表中移除，并插入到*size_class*的slab列表中。

图5显示了一个用板块变形技术将小尺寸类的板块转化为大尺寸类的板块的例子。在变形之前，B1、B5和B6被分配到*slabbefore*。在变形过程中，为每个块设置*cntblock*。对于NB0，它的*cntblock*被设置为1，因为在NB0中只有*slabbefore*的B1被占用。对于NB2，它的*cntblock*被设置为2，因为在NB2中B5和B6都被占用。请注意，板块变形也支持从大尺寸类到小尺寸类的板块转换。

区块释放。当一个块被释放时，NVAlloc通过查询*cntslab*和*cntblock*确定它是否是*slabbefore*中的一个块。*Blockbefore*将被直接放回*slabin*，*绕*过tcache，其状态在*index_table*中设置为*free*。当*cntslab*变为0时，*slabin*被重置为一个普通的*slabafter*，并再次插入LRU列表中。

板块变形方案引入了一个小的开销，因为它是不经常启用的。对于分配和释放新大小类的块，*slabin*中的块可以被用来填充tcache，作为nor-mal块而没有额外的开销。对于*块*的释放*before*，NVAlloc需要修改它在index_table中的状态并刷新它。

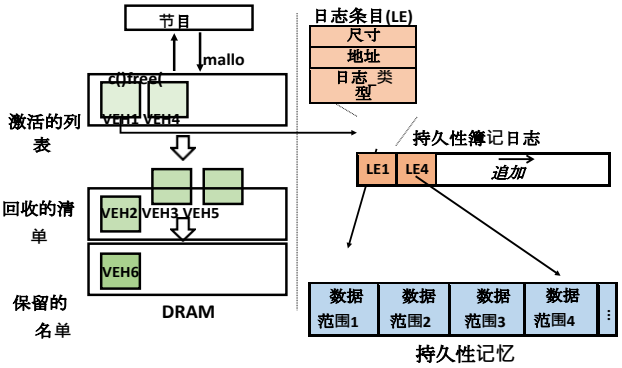


图7：日志结构记账法的说明。

这些操作的成本很低，因为*块状ef矿石*只需要在一个小时内完成。在我们的实验中，对总块数的20%进行统计。我们在第6.4节对其开销进行量化。

5.3 日志结构的簿记

对于大量的分配和释放，NVAlloc使用DRAM中的虚拟范围头（VEH）来管理持久性内存中的每个范围。VEHs在激活列表、回收列表和保留列表之间移动。基本的元数据（例如，大小和地址）需要在持久性内存中相应的范围头中进行更新。由于就地更新的程度头，对程度头的随机访问是不可避免的。为了消除大量分配带来的随机访问，我们设计了一个日志结构的记账方案，如图7所示。具体来说，当一个虚拟范围头（例如，*V EH* 1）被更新时，它的基本元数据被附加到一个持久的记账日志中。该日志是按顺序写入的，当它满了之后就会被清理掉。我们用持久的内存空间换取更好的空间定位。

由于以下原因，分配器中的日志结构记账的开销非常低。首先，持久的记账日志只存储小的基本元数据。每个日志条目只有8B，由26比特的 "size"、36比特的 "addr "和2比特的 "log type "组成。对于 "addr"，我们只需要36位，因为（1）在英特尔x86处理器的64位地址空间中只使用低阶48位[31]；（2）我们的地址是4KB对齐的，因此在日志条目中不需要低阶12位。这与传统的日志结构的文件系统不同，后者的日志条目可以和请求大小一样大。因此，元数据日志的空间开销要比传统的日志结构文件系统的数据日志小得多。因此，我们可以用更多的空间来换取更好的空间定位，而不会产生垃圾收集的开销。第二，在持久性记账日志中，日志条目的大小是统一的，这导致了一个简化的日志管理过程。

一个主要的挑战是写小的日志条目时的缓存行刷新。我们介绍了持久性记账日志的布局，以及如何防止日志中的缓存行刷新和如何减少GC开销。

持久性记账日志的布局。持久性记账日志有两个组成部分，分别在DRAM和持久性存储器中。它的布局如图8所示。在初始化的时候，NVAlloc在持久性存储器中创建了一个100MB的文件来存储日志条目。一个日志文件被分为1KB的小块，每个小块

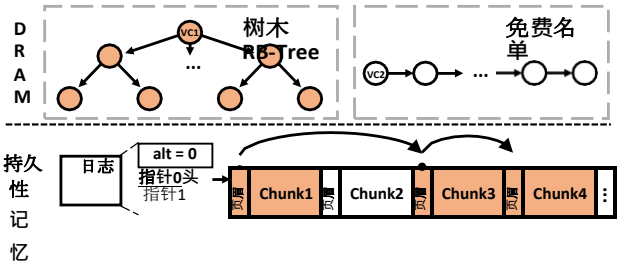


图8：持久性记账日志的内存布局。VC表示vchunk。橙色和白色的块分别表示活动块和空闲块。

其中可以存储128个日志条目。这些块被作为一个链接列表来管理。日志文件有一个日志头，它存储了两个指针和一个*alt*位。其中一个指针指的是恢复时活动日志块的链接列表的头部；另一个指针只被GC用来建立一个新的链接列表。*alt*位表示这两个指针中的哪一个是活动的。每个块都有一个块头，它存储了它的ID号，一个激活位，以及一个指向下一个激活块的指针。

为了加快日志操作，每个日志块都有一个对应的*易失性块，即*DRAM中的*vchunk*。它存储了一个位图，表明该块中的有效日志条目。此外，NVAlloc使用一个红黑树来管理分配的块的vchunk。在GC之后，所有被释放的块被保留在一个链接列表中，以便在将来快速分配。当需要一个新的日志块时，它首先从自由列表中检索出来。如果自由列表是空的，就会创建一个新的块，并追加到持久性内存中的日志文件的尾部。

基本的日志操作。在NVAlloc中，日志条目有两种不同类型：正常条目和墓碑条目。当分配一个大块时，一个正常条目将被创建并添加到当前块中。为了避免缓存线刷新，我们以交错的方式将连续的日志条目映射到块中，类似于5.1节中的方法。然后，其vchunk的位图中的相应位将被设置。与普通条目类似，当释放一个大的范围时，将添加一个墓碑条目。此外，墓碑条目将存储要删除的普通条目的指针，并在vchunk的位图中清理其对应的位，以便快速进行垃圾收集。**垃圾收集（GC）。**为了控制日志文件的大小，我们需要执行垃圾收集（GC）来丢弃那些被墓碑标记为删除的日志条目。NVAlloc支持两种GC算法包括*快速GC*和*慢速GC*[20]，它们被设计为在GC开销和内存效率之间做出不同的权衡。

快速GC算法会扫描红黑树中每个vchunk的位图。如果一个vchunk的位图是空的，它将被移到自由列表中。因为快速GC算法不需要访问持久性内存，所以它的开销是微不足道的。NVAlloc在大多数时候都执行快速GC，只有当日志文件的大小大于内存使用阈值时才切换到执行慢速GC。

*U sagepme.*当慢速GC算法执行时，将创建一个新的活动块列表$list_{new}$来存储活动日志条目。慢速GC算法扫描现有活动块列表$list_{old}$中的所有日志条目，并通过位图检查日志条目是否活着。在$list_{old}$中活着的日志条目将被复制到$list_{new}$中的块。墓碑条目将被过程。当扫描完成后，NVAlloc通过翻转*alt*位将$list_{new}$作为当前活动块列表。然后，它回收了$list_{old}$中的所有小块。

6 评价

6.1 实验设置

实验平台。我们在一台有两个英特尔至强黄金5218R CPU的Linux服务器（内核5.3.0-50300-generic）上运行实验。每个CPU有20个物理核心（40个超线程），64GB DRAM和两个英特尔Optane DIMM（每个DIMM 128GB）。连接到CPU上的每一对DIMM都安装了Ext4-DAX文件系统，并配置为App Direct模式。为了避免NUMA效应，我们使用numactl工具将每个线程绑定到第一个套接字的一个内核上。所有的源代码都是用g++7.5编译的，带有-O3.

比较了分配器。我们将NVAlloc与最先进的持久化分配器进行比较，包括PMDK[11]、nvm_malloc[37]、PAllocator[31]、Makalu[3]和Ralloc[4]。由于除PAllocator外，所有这些都是开源的，我们使用它们的公共实现进行测试。我们根据论文中的描述，尽可能忠实地重新实现PAllocator。我们排除了jemalloc[17]、Hoard[2]和tcmalloc[18]，因为它们是不稳定的分配器。为了支持现有的一致性模型，我们实现了两个版本的NVAlloc。*NVAlloc-LOG*和*NVAlloc-GC*，它们分别利用WAL和GC来保持崩溃一致性和避免内存泄漏。为了便于描述，我们把基于PMDK和WAL的分配器（即nvm_malloc、PAllocator和NVAlloc-LOG）称为*强一致性分配器*。相比之下，我们把基于GC的分配器（即。Makalu、Ralloc和NVAlloc-GC）*弱一致性分配器*。

6.2 使用基准进行评价

基准。我们在评估中使用了五个有代表性的基准，每个基准都有一个独特的分配模式。

Threadtest[2]测量一个分配器在i次分配中的多线程性能。在每个迭代中，每个线程分配n对象，大小为s，然后独立地释放所有对象。在实验中，我们设定i=10^4，n=10^5，s=64 B。

Prod-con[2，36]模拟了一个生产者-消费者的工作负载。19个线程。每对线程产生和消耗n对象，其总大小为s。每对线程中的一个分配对象，另一个释放对象。我们的实验设定n=2×10^7，s=2×10^7。

$\overline{t}$

移至

64 B.

Shbench[29]是一个分配器的压力测试。在每个迭代中，每个线程分配和释放从64B到1000B的不同大小的对象，较小的对象被更频繁地分配和释放。我们运行了^{105}次迭代。

Larson [22, 31] 模拟了一种行为，即一个线程分配的一些对象被另一个线程释放。在每个迭代中，每个线程随机地分配和释放^{103}个不同大小的对象。在^{104}次迭代之后，每个线程创建一个新的线程，从剩余的对象开始，重复同样的分配/释放程序。我们产生了两个工作负载。*Larson-small*管理小对象（64 B到256 B），*Larson-large*管理大对象（32 KB到512 KB）。我们运行了30秒的测试。

DBMStest[16]：它用TPC-DS基准模拟数据库中t线程的分配情况。在每个迭代中，每个线程

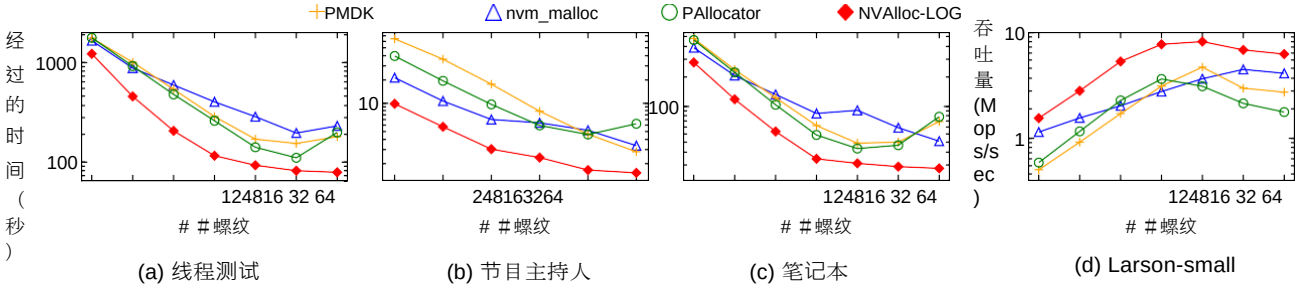


图9：使用强一致性分配器的小型分配器的性能（log10缩放）。

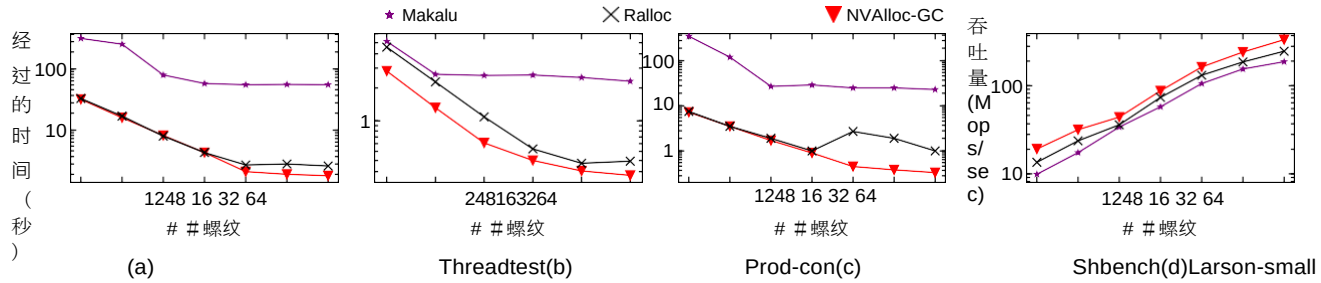


图10：使用弱一致性分配器的小型分配器的性能（对数10缩放）。

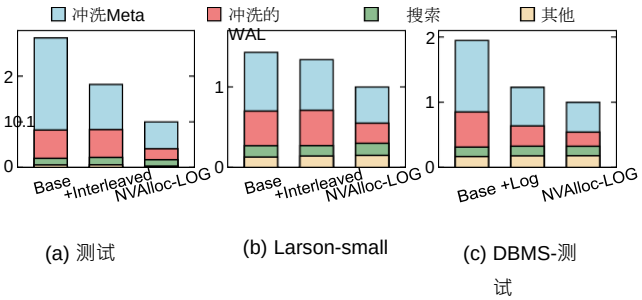


图11：性能分解分析。

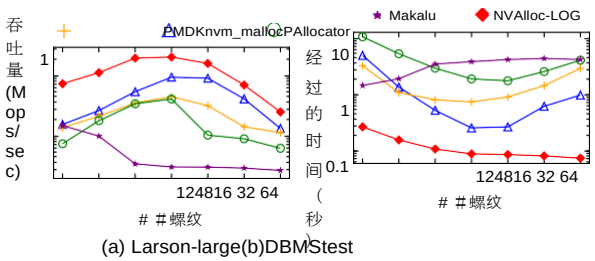


图12：大型分配的性能（对数10缩放）。

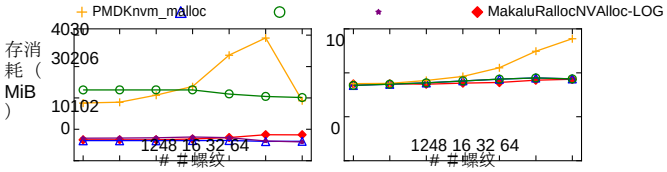
分配n大对象，这些对象的大小遵循32KB到512KB之间的泊松分布，然后随机删除其中的90%。我们选择$n=^{104个}$对象。我们运行了50次热身和50次评估迭代。

小型分配的性能。我们首先评估了Threadtest、Prod-con、Sbench和Larson-small上不同数量线程的小对象分配器的性能。为了公平比较，我们在图9和图10中分别显示了强一致性和弱一致性分配器的结果。总的来说，NVAlloc在所有基准上的表现和扩展性都优于所有的同类产品。

图9显示，在四个基准上，NVAlloc-LOG比PMDK、nvm_malloc和PAllocator分别快6.4倍、3.5倍和3.9倍。NVAlloc-LOG优于其同类产品，因为交错映射减少了元数据更新和WAL更新中的缓存行刷新次数。为了进一步分析这些结果，我们使用linux *perf* 工具[14]来测量不同基准的执行时间的细分。我们只评估了Threadtest、Larson-small和DBMS-test，因为其他基准测试显示了类似的结果，而且篇幅有限。执行时间包括对象搜索、分割、以及在分配/去分配中凝聚扩展（表示为搜索）、元数据冲洗时间（FlushMeta），WAL冲洗时间（FlushWAL），以及其他时间（Other）。我们在图11中展示了八个线程的结果。*Base*表示NVAlloc-LOG，没有启用第4节中描述的任何优化措施。*+Interleaved表示*只启用交错式缓存布局的版本。*+Log*表示只启用了日志结构的记账方式的版本。如图11(a)所示，FlushMeta和FlushWAL占Threadtest上*Base*的执行时间的87%。*+Interleaved*将FlushMeta的时间和*Base*的总执行时间分别减少了51%和35%。当交错映射被额外用于板块位图和WAL时，NVAlloc-LOG进一步将总的冲洗时间（FlushMeta和FlushWAL）减少了48%。

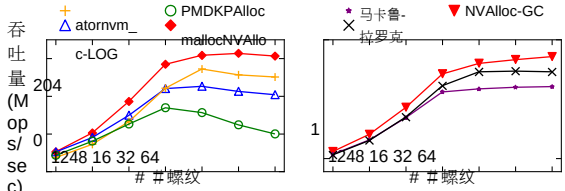
图11(b)显示了在Larson-small上执行时间的细分。采用交错式缓存布局，BASE-tcache的FlushMeta时间减少了14%，总时间减少了7%。*+Interleaved实现了*这个收益，因为tcache从每个子缓存中依次选择块，避免了更新持久位图时的缓存线刷新。在板块和WALs中启用交错映射后，NVAlloc-LOG比Base实现了31%的性能提升。另一个观察结果是，与Larson-small相比，NVAlloc-LOG在Threadtest

上获得了更多好处。原因是



(a) 线程测试(b)DBMStest

图13：空间消耗。



(a) 强一致性 分配器(b)弱一致性分配器

图14：FPTree的性能。

是，在Threadtest上有更多的缓存行刷新，因为它有固定的分配大小。

图10显示，NVAlloc-GC在四个基准上比Makalu和Ralloc实现了70倍和6倍的最大速度。NVAlloc-GC有更好的性能，因为Makalu和Ralloc使用嵌入式链表来管理持久化板块中的空闲块，而NVAlloc-GC使用位图来管理块，改善持久化存储器中的数据访问定位。NVAlloc-GC还在DRAM中维护了一个易失性的位图副本，以实现快速的自由块索引，并减少对持久性内存的访问。

大型分配的性能。图12显示了大对象分配的性能。因为NVAlloc-GC和NVAlloc-LOG在大型分配中表现相同，而且Ralloc在其开源实现中对大型对象不能正确工作，所以我们在图12中排除了它们。在Larson-large和DBMStest上，NVAlloc-LOG比PMDK、nvm_malloc、PAllocator和Makalu快40倍、18倍、55倍和57倍。NVAlloc-LOG比它的同类产品快，因为它使用了日志结构的簿记和WAL中的交错映射。

为了显示日志结构的簿记和叶间映射的影响，我们用DBMStest测量了执行时间的破损。图11(c)显示了结果。*+Log*是在*Base*中只添加了日志结构化记账的版本。它将总的冲洗时间（FlushMeta和FlushWAL）减少了45%，因为日志结构的簿记提供了对持久性内存的顺序写入模式。NVAlloc-LOG进一步减少了26%的刷新时间，因为消除了重复刷新日志项的缓存行。

空间使用结果。图13显示了不同分配器的内存消耗。我们只以Threadtest和DBMS为例，分别说明小对象和大对象的分配情况，因为其他基准表现出类似的结果。由于NVAlloc-LOG和NVAlloc-GC显示了相同的空间消耗，我们只包括NVAlloc-LOG。在两个基准上，NVAlloc-LOG产生的空间消耗与其他分配器相当或更好。我们在图13（b）中排除了RAlloc，因为RAlloc在其开源实现中对大型对象不能正确工作。

6.3 使用FPTree进行评估

我们还用一个真实世界的键值存储应用--FPTree[32]来评估NVAlloc。它是一个持久性的并发B+树，内部节点存储在DRAM中，叶子节点存储在持久性内存中。FPTree的每个节点包含64个子节点。为了支持不同大小的值，FPTree使用叶子节点中的原始值作为指向实际键值对的指针。我们将原始键和值的大小设置为8 B。由于大多数键值对在Facebook中都很小[5]。

我们将实际键值对的大小设置为128B。我们用50%的插入和50%的删除操作的混合工作负载来测量FPTree的性能。我们用50M个键值对对FPTree进行预热，然后用不同数量的线程执行50M个操作。

图14显示了使用不同分配器的FPTree的吞吐量。与PMDK、nvm_malloc和PAllocator相比，使用NVAlloc-LOG，FPTree的吞吐量分别达到1.2倍、1.5倍和3.1倍。使用NVAlloc-GC，FPTree的速度提高到35.4%。与其他分配器相比，带有NVAlloc的FPTree产生了相当的空间消耗，因为在给定的工作负载中没有触发slab morphing技术。

6.4 使用Fragbench进行评估

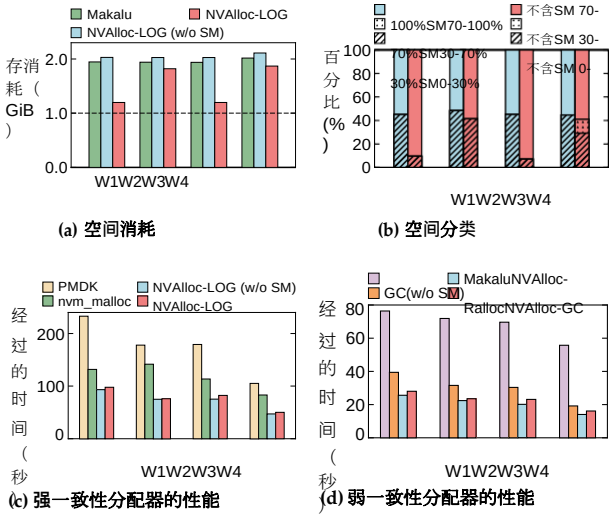
然后，我们在Fragbench[35]上对NVAlloc进行评估，其工作负载列于第3节的表1。图15(a)显示了不同分配器的空间占有率。我们排除了图1(b)中的那些，除了Makalu，以避免多余的表述。由于NVAlloc-LOG和NVAlloc-GC产生相同的空间消耗，我们只包括NVAlloc-LOG。为了比较，我们还评估了没有板块变形策略的NVAlloc- LOG（NVAlloc-LOG w/o SM）。结果表明，由于采用了板块变形技术，NVAlloc-LOG实现了最小的空间消耗。

为了验证这一点，图15（b）显示了NVAlloc- LOG的空间细分。我们根据内存利用率将板块分为三类。0-30%, 30-70%, 70-100%.图15(b)显示，与不使用slab morphing的方案相比，使用slab morphing的NVAlloc-LOG大大增加了高利用率的slab的数量。因此，它减少了总体内存消耗。

图15（c）和（d）显示了NVAlloc的性能。NVAlloc优于其他所有的分配器，因为它使用了交错映射技术，正如第6.2节所讨论的。我们还观察到，板块变形方法可能会带来平均4.5%的性能下降，因为它需要刷新板块元数据。尽管有轻微的性能下降，但板块变形可以减少高达41.9%的内存使用。

6.5 敏感度分析

位条纹的数量。交错映射的效率与位条的数量有关。较大的位条数可以减少刷新的次数，因为每个位条的位数较少，因此较少的块被映射到同一缓存行中。然而，这可能会增加刷新延迟，因为当大量的缓存行同时刷新时，我们可能会耗尽持久性内存中的XPBuffer[40]。为了探索数量的影响



(a) 空间消耗 (b) 空间分类

(c) 强一致性分配器的性能 (d) 弱一致性分配器的性能

图15：Fragbench的结果。SM表示板块变形。

我们在Threadtest上运行NVAlloc-LOG，以不同数量的线程作为研究案例。

如图16(a)所示，NVAlloc-LOG的执行时间并没有随着我们增加位条的数量而线性减少。这是因为执行时间是由软件参数（即位条的数量和线程的数量）和硬件参数（即XPBuffer的行数）决定的。

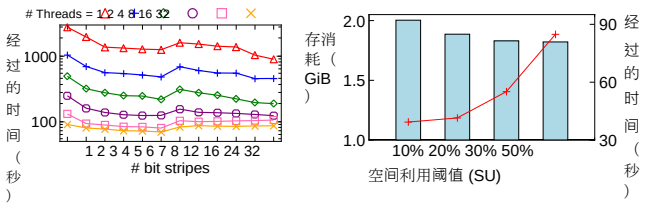
持久性内存和它的大小）。在本文中，我们选择比特条带的数量为6，因为它在大多数情况下实现了最佳性能。我们把在不同的线程并发水平下动态地选择条带的数量作为未来的工作。

蜕变参数。蜕变技术中的板块空间利用阈值（SU）也影响NVAlloc的效率。较大的SU允许更多的板块被变形，从而减少了内存消耗，而较小的SU则减少了变形的成本，从而提高了性能。图16(b)显示了SU对NVAlloc-LOG在W4工作负载上的影响。基于这些结果，我们根据经验将SU设定为20%，以便在内存消耗和分配器性能之间实现适当的权衡。虽然这个参数在我们的初始原型中运行良好，但使用一个更复杂的参数可能更有益。我们把这种探索留给未来的工作。

6.6 高空讨论

GC开销。为了评估日志清理对大型分配的日志结构化簿记的效率，我们在Larson-large和DBMStest上运行NVAlloc-LOG。图17显示，有了GC，当$U\ sa_{geme}$=0.2%时，Larson-large的吞吐量略有下降（只有3%），DBMS的吞吐量下降8%。GC开销是微不足道的，因为日志结构的文件是轻量级的，因为它只保留分配元数据，所以复制开销很低。

恢复。图18显示了开源分配器的恢复时间。我们首先创建一个有10M个节点的单一链表，这些节点的大小均匀地分布在64B和128B之间，然后我们使用一个单线程执行恢复。对于强一致性分配器，NVAlloc-LOG比PMDK和nvm_malloc慢，因为它需要扫描WALs和日志结构记账，而PMDK只需要旅行WALs，nvm_malloc推迟了



(a) 位条纹的数量对线程测试的影响 (b) 形态参数对Frag- bench的影响

图16：敏感度分析。

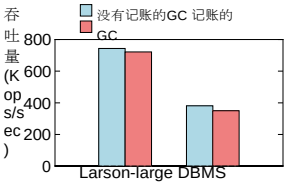


图17：GC的开销。

拨款人	时间
nvm_malloc	324us
PMDK	34ms
NVAlloc-LOG	45ms
Ralloc	552ms
Makalu	911ms
NVAlloc-GC	933ms

图18：恢复时间。

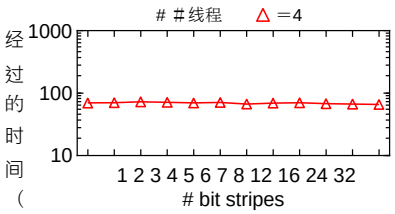


图19：交错映射对eADR的影响。

一些元数据重建到运行时去分配的过程。然而，NVAlloc-LOG在运行时性能上优于它们。对于弱一致性分配器，NVAlloc-GC的性能与Makalu相当。它比Ralloc慢，因为Ralloc在恢复时只需要扫描部分节点。

6.7 在仿真eADR平台上进行评估

eADR（扩展ADR）是第三代英特尔至强可扩展处理器支持的一项新功能，它可以确保CPU缓存的安全。

也是在电源故障保护域[12]。因此，在eADR上没有必要进行显式缓存行刷新。实施eADR需要更高的能量消耗、硬件成本和系统维护负担。考虑到这些问题，正如英特尔[34]所指出的，ADR和eADR平台将在可预见的未来并存。在本节中，我们在eADR平台上评估NVAlloc。由于eADR不是商业化的，我们通过删除ADR平台上所有被评估的分配器的冲洗操作（即*clwb*）*来模拟*它。我们只评估强一致性的分配器，因为弱一致性的分配器通过执行崩溃后的GC来移除大部分的冲洗操作。

首先，我们评估交错映射对eADR的影响。我们用4个线程运行Threadtest，而位条的数量从1增加到32。如图19所示，位条带的数量对NVAlloc-LOG的性能没有影响。由于交错映射增加了缓存的使用，我们在下面的实验中在模拟的eADR平台上禁用了交错映射。对于真实的eADR平台，我们使用

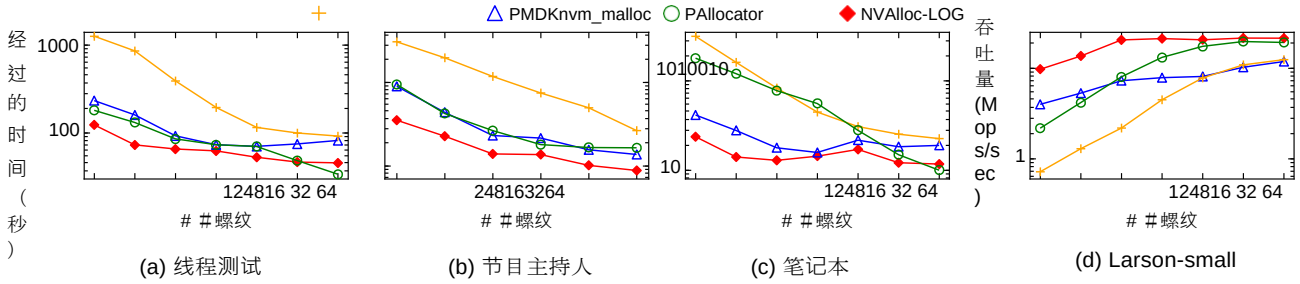


图20：在模拟的eADR平台上，小型分配的性能（对数10缩放）。

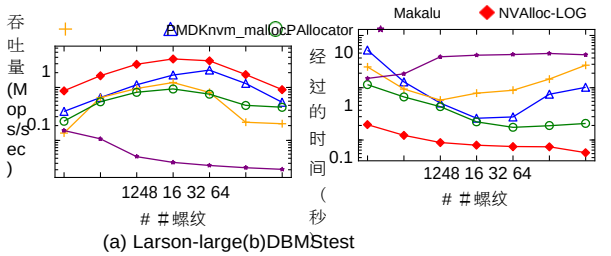


图21：在模拟的eADR平台上，大型分配的性能（log10缩放）。

pmem_has_auto_flush()在PMDK[11]自动检测eADR特征，然后禁用交错映射技术。第二，我们研究eADR上的小规模分配性能。图20中的结果显示，与其他强一致性分配器相比，NVAlloc-LOG平均提高了240%的基准性能。当线程数为64时，使用Threadtest的PAllo- cator的执行时间比使用NVAlloc-LOG的执行时间小27%。这是因为PAllocator为每个线程使用专用的小分配器。它为线程本地分配实现了更好的可扩展性，但导致Prod-con和Larson-small中频繁的跨线程操作的性能变差。第三，图21显示了NVAlloc-LOG在大型分配中的性能。我们可以观察到，它与Larson-large和DBMStest相比，平均有11倍的性能改进。这是因为我们设计的VEH和日志结构的记账方式减少了持久性内存访问的总数，提高了eADR的写入位置。

7 相关的工作

基于日志的分配器。支持事务性模型的持久性内存分配器在日志中记录内存地址和堆元数据的变化。在重放日志之后，分配器可以在崩溃之后重建其堆元数据。例如，nvm_malloc[37]将其堆元数据分为易失性和非易失性部分，以减少对持久性内存的数据访问。它对位图和日志的少量写入可能会导致缓存行的重新刷新。PAllocator[31]使用隔离匹配策略提供小块分配，使用索引树提供大块分配。由于访问页头和微型日志中的2-B块元数据，它也受到了缓存线刷新问题的影响。Poseidon[15]是第一个执行基于页面保护的持久性内存分配器。它也使用位图和日志进行堆元数据管理。

基于GC的分配器。为了避免写日志和刷新元数据的开销，最近的分配器[3, 4, 28]使用垃圾收集（GC）来重建崩溃后的堆元数据，通过遍历从持久化根指针中获得持久化堆。Makalu[3]是第一个使用离线GC来在线放松堆元数据持久性约束的分配器，导致更快的小块分配。Ralloc[4]将瞬时无锁分配器LRalloc变成一个持久性分配器。和Makalu一样，Ralloc使用了崩溃后的GC来避免缓存行的重新刷新。DCMM[28]通过简单地分配新块追加到现有的堆区和采用并行运行的后台恢复线程。

使用内部集合的分配器。PMDK中的分配器提供了非交易性的原子分配[11]。使用PMDK的接口（例如POBJ_FIRST()和POBJ_NEXT()），用户永远不会丢失持久性内存中对象的引用。因此，使用PMDK内部集合的分配器不需要维护写前日志。

NVAlloc中提出的方法可以用来实现基于日志、基于GC或基于内部收集的持久化模式。与现有的分配器相比，在这些模式中，我们可以消除分配器引起的缓存线刷新和对持久性内存的随机写入。此外，由于我们使用了slab mor-phing，NVAlloc不再有由静态slab segregation引起的内存碎片问题。

8 结论

在本文中，我们设计了一个新的分配器，名为NVAlloc，用于在持久性内存中定位/分配内存对象。NVAl- loc利用交错元数据映射、日志结构化记账和板块变形技术来消除分配器引起的缓存行刷新、小型随机写入和内存碎片化问题。我们的实验结果表明，NVAlloc可以显著提高分配器的性能和空间利用率。随着持久性存储器变得越来越流行，我们希望NVAlloc中的各种优化技术能够启发未来一代的持久性存储器系统。

鸣谢

我们真诚地感谢我们的牧羊人Angela Demke Brown和匿名审稿人的建设性建议。这项工作得到了国家重点研发计划（2021ZD0110700）、国家科学基金（62172361）、浙江实验室研究项目（2020KC0AC01）、美国国家科学基金会（CNS 1906541）以及阿里巴巴创新研究项目的部分支持。

参考文献

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor.2015.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。在*2015年ACM SIGMOD国际会议上，关于数据管理（SIGMOD）的论文集*。Association for Computing Machinery, 707-722.

[2] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson.2000.Hoard:用于多线程应用的可扩展的内存分配器。*ACM Sigplan Notices* 35, 11 (2000), 117-128.

[3] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm.2016.Makalu:非易失性存储器的快速可恢复分配。*ACM SIGPLAN通告*51，10 （2016），677-694。

[4] 蔡文涛，文浩森，H Alan Beadle，Chris Kjellqvist，Mohammad Hedayati，和Michael L Scott。2020.理解和优化持久性内存分配。在*2020年ACM SIGPLAN国际研讨会上，内存管理（ISMM）*。60-73.

[5] 曹志超，董思颖，Sagar Vemuri，和David HC Du.2020.表征、建模和基准测试Facebook的RocksDB关键值工作负载。在 *第18届USENIX文件和存储技术会议（FAST）上*。209-223.

[6] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu.2017.高效支持非易失性存储器上的位置独立性。In *Proceed- ings of the 50th Annual IEEE/ACM International Symposium Fon Microarchitecture （MICRO）*.191-203.

[7] 陈友民，陆友，杨帆，王庆，王阳，舒继武。2020.FlatStore:一个用于持久性内存的高效日志结构化键值存储引擎。*在第二十五届国际编程语言和操作系统架构支持会议（ASPLOS）论文集*。1077-1091.

[8] 陈章玉，黄宇，丁波，左鹏飞。2020.用于持久性内存的无锁并发级别哈希。在*2020年USENIX年度技术会议（ATC）论文集*。799-812.

[9] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson.2011.NV-Seaps:使用下一代非易失性存储器使持久性对象快速而安全。*ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105-118.

[10] 英特尔公司。2018.Redis。https://github.com/pmem/redis/tree/3.2-nvml/。

[11] 英特尔公司。2020.持久性内存开发工具包。http://pmem.io/。

[12] 英特尔公司。2021年。eADR：持久性内存应用的新机会。https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[13] Andreia Correia, Pascal Felber, and Pedro Ramalhete.2018.Romulus:用于持久性事务性内存的高效算法。在*第30届 "算法和架构中的并行性研讨会" （SPAA）上*。271-282.

[14] Arnaldo Carvalho De Melo.2010.新的linux perf工具。In *Slides from Linux Kongress*, Vol. 18.1-42.

[15] Anthony Demeri, Wook-Hee Kim, R Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. 2020.Poseidon:安全、快速和可扩展的持久性内存分配器。在*第21届国际中间件会议论文集 （中间件）*。207-220.

[16] Dominik Durner, Viktor Leis, and Thomas Neumann.2019.On the Impact of Memory Allocation on High-Performance Query Processing.在*第15届新硬件上的数据管理国际研讨会（DaMoN）会议上*。1-3.

[17] 杰森-埃文斯。2021年。jemalloc.https://github.com/jemalloc/jemalloc/。

[18] Inc 谷歌。2021年。tcmalloc。https://github.com/google/tcmalloc。

[19] 顾金玉，于倩倩，王夏阳，王兆国，臧炳玉，关海兵，陈海波。2019.Pisces:A Scalable and Efficient Persistent Trans- Actional Memory.In *2019 USENIX Annual Technical Conference (ATC)*.USENIX 协会，913-928。

[20] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda.2017.日志结构的非易失性主存储器。In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*.703-717.

[21] Mark S Johnstone和Paul R Wilson。1998.内存碎片问题：解决了？*ACM Sigplan通告*34，3（1998），26-36。

[22] Per-Åke Larson和Murali Krishnan.1998.长时间运行的服务器应用程序的内存分配。*在第一届内存国际研讨会论文集 管理（ISMM）*。176-185.

[23] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh.2017.WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems.In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*.257-270.

[24] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram.2019.配方。将并发的DRAM索引转换为持久性内存索引。在*第27届ACM操作系统研讨会论文集 Principles （SOSP）*。462-477.

[25] 联想。2018.Memcached-PMEM。https://github.com/lenovo/memcached-pmem/。

[26] 刘继航，陈世民，和王路军。2020.LB+树:优化持久性3DXPoint内存上的索引性能。*VLDB 捐赠会议论文集* 13, 7 (2020), 1078-1090.

[27] 卢宝通，郝向鹏，王天正，和罗志祥。2020.Dash: 可扩展的 Persistent Memory上的Hashing。*Proc.VLDB Endow.*13, 8 (2020), 1147-1161.

[28] 马绍南，陈康，陈世民，刘孟星，朱江浪，康洪波，吴永伟。2021.ROART: Range-Query Optimized Persistent ART.In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*.1-16.

[29] Inc MicroQuill.2014年。shbench.http://www.microquill.com/。

[30] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert.2013.为可字节寻址的非易失性主存储器提供一致的、持久的和安全的内存管理。在*第一届ACM SIGOPS操作系统及时结果会议（TRIOS）的论文集*。1-17.

[31] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes.2017.大规模持久性主内存系统的内存管理技术。*VLDB 捐赠会议论文集*10，11（2017），1166-1177。

[32] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner.2016.FPTree:一个用于存储类内存的混合SCM-DRAM持久性和并发性B-tree。In *Proceedings of the 2016 International Conference on Management of Data （SIGMOD）*.371-386.

[33] Bobby Powers, David Tench, Emery D Berger, and Andrew McGregor.2019.网格。压缩C/C++应用程序的内存管理。In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*.333-346.

[34] 安迪-鲁道夫2020.Persistent Memory Programming without All That Cache Flushing.*SDC*（2020）。

[35] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout.2014.基于DRAM的存储的日志结构化内存。在*第12届USENIX会议论文集 on File and Storage Technologies （FAST）*。1-16.

[36] Scott Schneider, Christos D Antonopoulos, and Dimitrios S Nikolopoulos.2006.可扩展的位置意识的多线程内存分配。In *Proceedings of the 5th International Symposium on Memory Management (ISMM)*.84-94.

[37] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner.2015. nvm malloc:NVRAM的内存分配。*adms@ vldb* 15 (2015), 61-72.

[38] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles.1995.动态存储分配。A Survey and Critical Review.在 *内存管理国际研讨会(IWMM)的论文集*。Springer, 1-116.

[39] 吴凯，任杰，彭毅，和李东。2021.ArchTM: 架构感知的、高性能的持久性内存事务。在*第19届USENIX文件和存储技术会议（FAST）论文集*。141-153.

[40] 杨健，Juno Kim，Morteza Hoseinzadeh，Joseph Izraelevitz，和Steve Swanson。2020.可扩展持久性存储器的行为和使用的经验指南。在*第18届USENIX文件和存储技术会议（FAST）论文集*。169-182.