非易失性存储器上GC触发的数据移动时保存可寻址性

叶振中，华中科技大学，中国
徐元超和沈西鹏，美国北卡罗来纳州立大学
华中科技大学金海和廖晓飞，中国
YAN SOLIHIN, 中佛罗里达大学, 美国

这篇文章指出了应用级垃圾收集（GC）对非易失性内存（NVM）的使用造成的重要威胁。由GC引起的数据移动可能会使NVM上的对象指针失效，因此会损害跨执行的持久性数据的可重用性。文章提出了移动无关寻址（MOA）的概念，并开发和比较了三种新的解决方案，以实现解决可寻址问题的概念。它在五个基准和一个真实世界的应用中评估了这些设计。结果表明，所提出的解决方案，特别是硬件支持的多级GPointer，有希望以空间和时间效率的方式解决这个问题。

CCS的概念。- **计算机系统组织** → *架构*；- **软件及其工程** *软件组织和属性*；- **硬件** → *新兴技术*；*内存和密集存储。*

其他关键词和短语。持久性内存、垃圾收集器、内存管理

ACM参考格式。
叶晨曦，徐元超，沈西鹏，金海，廖晓飞，和闫索利欣。2022.非易失性存储器上GC触发的数据移动时保存地址能力.*ACM Trans.Arch.Code Optim.*19, 2, Article 28 (March 2022), 26 pages.
https://doi.org/10.1145/3511706

1 简介

字节可寻址的非易失性存储器（NVM）弥补了持久性存储和DRAM之间的差距，提供了比传统存储更好的性能，同时又比DRAM提供了数据的持久性。程序员可以使用NVM作为DRAM的替代品，同时享受数据持久性的好处。这些好处主要体现在数据的可重用性上：数据可以被重用。

这项工作得到了国家自然科学基金委员会（NSFC）61832006、62072198、61825202和61929103号资助，以及美国国家科学基金会（NSF）CNS-2107068、CNS-1717425、1900724和2106629号资助的共同支持。CNS-2107068, CNS-1717425, 1900724, 和 2106629。
作者的地址。C. Ye, H. Jin, and X. Liao, National Engineering Research Center for Big Data Technology and Sys- tem, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Computer Science and Technology, Wuhan, Hubei, China, 430000; emails:{yecc, hjin, xfliao}.@hust.edu.cn; Y. Xu and X. Shen, North Carolina State University, Raleigh, North Carolina, USA, 27695; emails:{yxu47, xshen5}@ncsu.edu; Y. Solihin, University of Central Florida, Orlando, Florida, USA, 32816; email:Yan.Solihin@ucf.edu。

https://doi.org/10.1145/3511706

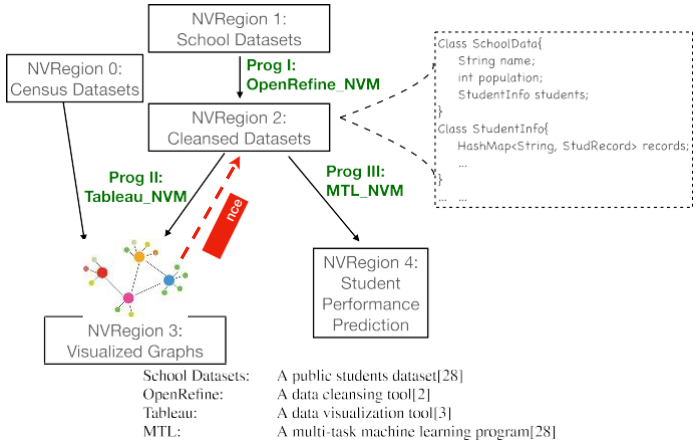


图1.一个例子说明了在NVM上由GC引起的数据寻址能力的损失。每个实线框代表了一个NVRegion。当程序3运行时，它的GC在NVM上打包并移动清理过的数据集中的学生记录，这就破坏了从可视化图形到清理过的数据集中的学生记录的引用。

在相同或不同程序的执行中重复使用，而不需要经过传统持久性存储中所需要的对象序列化和反序列化。为了实现这个机会，必须在不同的执行和程序中保持对象的可寻址性，这样，当一个程序在某个时候运行时，它可以在NVM上找到它应该访问的对象。然而，当使用应用程序级的垃圾收集（GC）时，持久性对象的这一重要属性就会丢失。垃圾收集器是管理型编程语言（Java、C#等）的一个重要部分；这种语言是最受欢迎的语言之一[1]。当程序运行时，GC会自动管理程序所使用的内存。它可以检测到死的对象，收回分配给它们的内存，并通过将空闲的内存空间聚集在一起来减少内存碎片。

这种应用层面的GC对NVM上对象的寻址性造成了特殊的复杂性。一个NVM可能由许多内存区域（称为NVRegion或NVPool）组成，每个区域是独立的NVM块，用于映射、解映射和进程间共享。一个对象可能被不同NVRegion中的多个对象所指向。由于一个NVM对象的寿命可能会超过一个程序的寿命，在一些程序的早期执行中可能会创建对一个对象的一些跨区域引用。当一个程序使用一个NVM对象时，它可能不知道所有指向该对象的指针--其中一些指针可能在一个NVRegion上，而这个程序甚至可能没有访问权限。因此，当这个程序中的GC线程移动该对象时，GC没有办法更新现有设计中指向该对象的所有指针。这些指针会被破坏。

*例子。*图1用一个涉及实际数据集和三个应用程序的场景来说明这个问题，每个应用程序都对应于一个真实世界的软件程序（假设已经被修改为利用NVM）。

第一个程序，`OpenRefine_NVM`，采用存储原始数据的NVRegion 1，并产生一个新的NVRegion，即NVRegion 2，以Java对象的形式保存经过清洗的学校数据集。这些数据包含学生的基本信息，如姓名和出生日期。NVRegion

[1]http://pypl.github.io/PYPL.html。

2 存储从原始数据反序列化的Java对象，以方便从Java程序访问数据集。

第二个程序，Tableau_NVM，结合了清洗过的数据和一些人口普查数据，如整个学校的人口，学生之间的关系，以及每个性别、国籍的人口等等，产生了一个新的NVRegion，即NVRegion 3，它持有生成的图表，捕捉到收入和学生成绩之间的相关性。执行过程中，在每个图形节点中建立一个参考，将其与NVRegion 2中的学生记录联系起来，这样用户就可以通过点击图形节点查看详细信息。同时，当有新学生入学或学生转学时，程序I会更新NVRegion 2，导致Java对象的创建、解构和更新。这些操作使NVRegion 2变得支离破碎。

随后，程序三MTL_NVM运行，从清洗过的学生数据集中建立起学生成绩预测模型。然而，当MTL_NVM分配一些数据时，Java GC被自动触发了。GC会打包和移动内存对象，包括NVRegion [22]上的学生记录。因为MTL_NVM的视图只包括NVRegion 2和它自己的NVRegion 4，GC不更新NVRegion 3的引用，使它们指向记录的过时位置。

这个例子说明了一个在现实世界的计算系统中常见的现象，其中持久性数据（如照片、联系人、日志和人口普查数据）通常可以被许多应用程序访问，而这些应用程序又会产生具有交叉引用的衍生数据。前面提到的参照物损坏问题在传统的文件系统中并不是一个问题，因为持久性数据在每次运行中都被序列化和反序列化。当字节寻址的NVM在不久的将来被广泛采用时，这个问题会立即显现出来。迫使所有的对象（由许多不同的程序）与交叉引用驻留在一个巨大的NVRegion上，并不是一个实用的解决方案。例如，在我们的例子中，程序II甚至可能没有权限将数据写入区域2，这是传统的GC所没有面对的情况。

虽然最近有许多关于NVM的研究（例如，[24, 27, 34, 36, 43, 46, 49]），但之前没有工作研究这个问题。事实上，这个问题在以前的文献中甚至从未被指出过。最近有几个关于*位置无关的指针的*研究[19]（也称为*relocat- able对象*[55, 56]）。然而，这些研究中考虑的情况只是整个NV区域的起始地址的变化。当对象被移动到NV区域内的不同位置时，他们的解决方案并没有保留对象的可寻址性，因为他们假设对象对其NV区域的起始地址的偏移量保持不变。

在这篇文章中，我们提出了关于这个问题的第一个系统研究。我们提出了*移动遗忘寻址*（MOA），这是一个在GC触发的移动中保留持久性对象的寻址能力的方案。图2以一种非常简化的方式说明了这个基本思想。MOA通过一个新设计的指针结构将不同区域的对象之间的直接引用替换为间接引用，这样，在对象移动时，GC只需要更新对象所在的NVRegion内的引用，而其他区域的间接引用仍然可以到达该对象。

这个基本想法很简单，但将其付诸实施却面临着一些重大挑战。一个直接的设计可能会导致91%的速度下降，这是因为对象解除引用的额外步骤和相关的高速缓存错过的大量增加。

在这项工作中，我们进行了深入的探索。我们提出了新的地址转换机制，它可以在GC引起的运动中保留物体的信息。我们从以下方面着手

2Java GC使用引用计数器来跟踪对象的有效性；它经常移动活的对象以减少内存碎片。

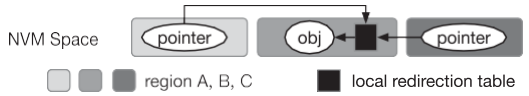


图2.以一种非常简化的方式说明了运动遗忘寻址（MOA）的基本思想。三个方框表示三个NVRegions；两个外部指针指向中间区域的一个对象。通过用间接引用重新放置直接引用（白框代表本地重定向表），GC只需要更新本地重定向表项，即使对象移动，其他区域的外部指针仍然可以到达该对象。

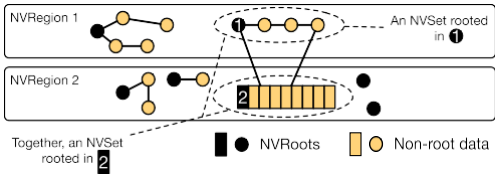
一种新的指针设计（OPointer），它将区域ID和对象ID嵌入指针中，然后查找两个独立的表来定位对象地址。当对象的数量不断增加时，这样的设计是低效的。因此，我们提出了SGPointer来使用对象组来控制表的大小，以及（受多层页表设计的启发）多层GPointer来增加对象分组的灵活性。这些解决方案使传统的GC具有跨区域寻址能力，即使在对象移动的情况下。集成到JVM中是很简单的：用本解决方案中提出的寻址模式覆盖持久性指针的解引用操作。主流的GC，如Java默认的G1 GC，提供了引用屏障，允许JVM开发者定制Java引用的解引用操作。因此，整合所提出的技术只需要对引用屏障和对象创建、移动和销毁的接口进行少量修改。本工作中提出的纯软件实现和基于硬件的实现提供了不同的选择，以满足不同的运行时效率和易于采用的需求。实验证明，这些解决方案可以实现MOA，同时大大减少了替代解决方案的运行时间开销（最高可达60%）。

据我们所知，这是第一项提出MOA指针支持的工作。
在NVRegions之间的引用。它有以下主要贡献。

- 这项工作首次指出了GC对持久化对象的可重用性造成的可寻址性问题。
- 它介绍了运动无关寻址（MOA）的概念，开发了第一套将该概念具体化以解决可寻址问题的解决方案，并对它们进行了比较。
- 它在六个基准上对设计进行了评估，并表明一个适应性的方法是必要的，以避免碎片化，而压实是必要的，以获得最高的灵活性。

2 场地

NVM访问模型。每个NVRegion都有一个唯一的整数ID，存储在其头部。对NVM的访问遵循先前工作[19]中描述的模型。一个NVRegion需要通过API调用打开，然后才能访问它的数据；该调用将该区域映射到虚拟地址空间。然而，对NVM数据的访问不需要通过特殊的API。相反，它们是通过直接的指针解除引用，其方式与访问标准的DRAM数据相似；这对生产力和代码的兼容性至关重要。有三种类型的指针可以指向NVM数据：一种是从DRAM指向NVM的指针，一种是从NVRegion的一个地方指向同一NVRegion的另一个地方（称为区域内指针），还有一种是从一个NVRegion指向另一个NVRegion（称为区域间指针）。这三种类型的指针对位置独立性有不同的要求，这在以前的工作中已经讨论过[19]。先前的工作为每种类型的指针提出了一种寻址方法。图3说明了该组织的概念视图和访问持久集的一个节点的代码片段，即NVSet。这些API




```
Find the virtual address of node ❶ :
NVRegion* r = open_region("region2")
node* array = get_root_node(r) //base address of array in NVRegion 2
OPointer n1 = array[2] //OPointer of the node in NVRegion 1
node* n = Deref(n1) //translates OPointer to virtual address
return n
```

图3.NVM组织的一个概念性视图。一个NVM由多个NVRegions组成，并可能有跨区域的引用。代码片断显示了API和NVM区域的使用。

*open_region*和*get_root_node*是由先前的工作提出的。API *Deref*将一个OPointer翻译成一个虚拟地址。

请注意，尽管这项工作假设一部分主存储器利用了NVM，如英特尔Optane DC DIMM，但访问模型和所提出的技术是通用的，可以应用于其他类型的主存储器。

垃圾收集（GC）。有各种各样的GC算法，如标记-扫除-紧凑GC，生成GC，等等。它们在程序执行中周期性地被调用，以自动回收内存。尽管有一些不移动对象的非移动GC算法[59]，但它们不能缓解内存碎片。大多数流行的GC在垃圾回收过程中打包活对象，导致数据在NVRegion内移动。GC不应该在NVRegion之间移动数据。

GC可以是在应用层面或系统层面。最常见的是应用级GC，它是一个生活在应用地址空间的线程，在应用执行期间为该应用回收内存。应用级GC是所有管理型编程语言（Java、C#、Scala等）的一个重要组成部分。系统级GC是一个独立于应用的过程；它试图为整个系统回收空间。一个例子是磁盘空间优化器。

对于NVM来说，两种GC都可以在不同的方面发挥作用。系统级GC可以看到所有区域间的点对点关系，但是它需要穿过整个系统的空间。因此，它是一个缓慢的过程，不经常被调用。为了便于有效性分析和对象移动，系统级GC可能要推断出所有数据的类型，这有时是很困难的。应用级的GC需要经常运行，以回收在应用执行过程中快速分配和释放的空间。它可以通过运行中的程序的知识来推断数据类型。在设想的用法中，应用级GC回收不是区域间指针目标的对象，而系统级GC偶尔被调用以回收其他对象。(这两种对象可以通过与指针类型相关的标记来区分）。事实上，许多现实世界的应用[42, 44, 57, 58]，尽管它们目前还没有利用持久性内存的优势，但在程序之间共享对象以实现快速的进程间共享。他们必须在移动数据之前同步程序，或者使用一个集中的进程来跟踪所有的对象。这样的合作方案不适用于NVM对象，因为NVM对象可能被所有运行中的程序不知道或没有访问权限的对象所指向。

请注意，NVRegion中的所有对象（无论是否有区域间指针）都会受到移动的影响。
当应用级GC将孔洞打包到大的连续自由空间中时，就会出现这种情况。由于应用级的GC通常是作为应用地址空间内的一个线程来实现的，它只对其地址空间内的指针保持可寻址性。在传统系统中，如果一个对象在共享内存中（可能由多个进程共享），GC通常不会移动它。在一个合作的环境中，GC可以在共享对象上工作[57]，但指针的更新需要通过进程间的显式同步来完成。

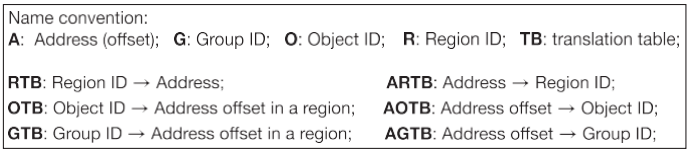


图4.缩略语参考。

对于NVM上的对象，出现了新的复杂情况。因为一个对象可能跨越程序的生命周期，一些跨区域的引用可能是在其他程序的早期执行中创建的，当一个程序使用一个NVM对象时，它可能不知道所有指向该对象的指针--其中一些指针可能是在这个程序甚至没有权限访问的NVRegion上。因此，当这个程序中的GC线程移动该对象时，GC没有办法更新现有设计中指向该对象的所有指针。下一节将介绍我们对这个问题的解决方案。

3 MOA的设计和实施

在本节中，我们首先提供了MOA的正式定义，明确了我们的工作范围，然后介绍了三种机制来实现NVM上对象的MOA。这些机制有一个共同的基本想法，即通过新的指针结构和辅助寻址方案，用间接引用取代直接引用。它们形成了一个进步，一个建立在另一个之上，具有不同的权衡和灵活性。

3.1 医学部

MOA是我们在这项工作中提出的一个概念，用来描述即使对象被移动也能保持数据对象的可寻址性的机制。为了清楚起见，我们提供了一个MOA的正式定义如下。

*定义3.1.*设O是一个数据对象，设S是在时间t*指向*O*的*整个数据引用集，即p，$p\ S\ L_1 == T\ (p)$，其中L_1是对象O的虚拟地址，$\forall p \in S \Rightarrow T\ (p)$返回一个引用$p$的目标地址。如果一个寻址机制满足以下条件，它就是**移动遗忘寻址**。当对象O在没有其他变化发生的时候，在时间t将其虚拟地址从L_1 (L_2 * L_1)改变为L_2，$T\ (p)$的目标地址，$p\ S$在t之后立即改变为L_2。

这个定义是通用的，涵盖了由各种原因引起的所有类型的数据移动（由程序员发起的手动数据移动，由运行时间触发的自动数据移动，等等）。在这项工作中，我们专注于GC引起的数据移动。这种移动对程序员来说是隐性的（看不见的）。因此，与程序员在应用程序代码中发起的数据移动不同，在这种情况下，维护移动数据的可寻址性应该由底层系统而不是程序员自动支持。

接下来，我们将介绍我们所开发的MOA方案。为了便于参考，图4包含了以下讨论中经常使用的首字母缩写。我们从 OPointer开始描述，这是所有方案中最简单的。

3.2 基本建议。OPointer

*OPointer*解决方案的基本思想是，当有数据移动时，通过用间接引用替换直接引用，将需要的更新本地化。替换是通过一个新的指针结构和一些辅助系统数据结构和运行时操作实现的。

表1.OPointers的助理数据结构

	缩写
RoleImplementationSharingOperations	RTBmap RID to base addresses of region and OTB direct address table across whole systemP
ARTBmap base address of region to RIDsorted binary treeacrosswhole	systemA NOTBmap OID to intra-region offsetdirectaddress
tableperNVRegionP F	AOTBmap intra-region offset to OIDhash tableper NVRegionA N F FP
Poolmanage free OIDsmin-	heapper NVRegionA N F

最后一列显示了数据结构的操作使用，P为指针解读，A为指针赋值，N为新对象分配，F为对象释放。

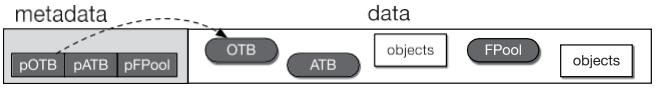
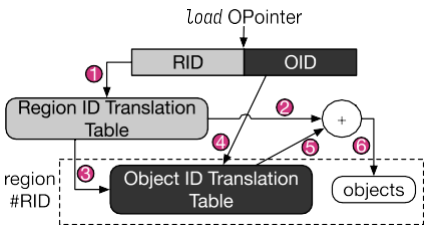


图5.OTB、AOTB和FPool驻留在NVRegions上，通过区域的元数据进行处理（第3.2节）。




```
Deref(OPointer):
1  RID = OPointer >> 32
2  region_addr = RTB[RID*2]
3  INT64* OTB = (INT64*)RTB[RID*2+1]
4  OID = OPointer & 0xFFFFFFFF
5  object_offset = OTB[OID]
6  object_addr = object_offset +
                 region_addr
7  return object_addr
```

图6.解除引用OPointer的说明（第3.2节）；右面圈起的数字与代码行#相关。

*3.2.1 设计。*OPointer将8B宽度的指针分成两个字段。前32位（从最重要的位开始）用于区域ID（RID），其余位用于对象ID（OID）。与与NV区域相关的RID类似，每个OID与需要MOA的对象相关。一些先前的NV指针设计[19, 56]也在指针内存储RID。然而，他们在指针的另一部分存储对象的偏移量而不是OID。这种区别很重要：当一个数据集移动时，它在NVRegion中的偏移量会改变，但它的OID却不会。

表1列出了助理数据结构和它们在支持OPointers方面的作用。它们包括四个映射表，RTB、ARTB、OTB和AOTB，以及FPool，一个基于最小堆的自由OID池。每个进程都有自己的RTB和ARTB。它们提供了RIDs和NVRegions的基址之间的映射。每当进程打开一个NVRegion时，就会在这两个表中加入一个条目。这两个表是暂时性的。OTB和AOTB是持久性的，住在它们帮助管理的NVRegion中，提供OID和区域上持久性对象的偏移量之间的映射。FPool是每个NVRegion，也住在它上面。它包含一个自由ID池，NVRegion上的新持久化对象可以选择使用。如图5所示，OTB、AOTB和FPool是通过区域的元数据来解决的。

*3.2.2 操作和启用的MOA。*对一个OPointer的解除引用包括几个操作，通过利用RTB和OTB，将OPointer中包含的RID和OID分别翻译成目标NVRegion的基地址和该区域中对象的偏移量，然后将它们相加成为目标对象的地址。图6显示了一个例子和伪代码。从一个对象的地址到一个OPointer的转换包括一个通过ARTB和AOTB的反向过程。

当一个持久性对象移动时，OPointer通过定位所需的变化来帮助实现MOA。在数据移动时唯一需要更新的是两个表，OTB和AOTB，用该NVRegion中对象的新偏移量更新它们的条目。由于OTB和AOTB都位于该NVRegion上，GC可以简单地作为GC过程的一部分进行修改。指向该对象的OPointer的值不需要改变；在移动之后，对它的解除引用仍然可以到达该对象。

OIDs是通过FPool管理的。FPool是一个迷你堆，包含一组新的持久化对象可能采用的ID。当一个新的持久化对象被创建，可以跨区域引用时，运行时从FPool中请求一个OID，将OID放入OTB和AOTB中，将OID与NVRegion中的对象的偏移量相联系。FPool总是返回最小的空闲OID，以尽量减少OTB中的碎片。如果FPool在ID请求时是空的，运行时通过将OID表的大小增加一倍来重新确定其大小，然后将新的索引添加到FPool中；AOTB也被重新确定大小。最小堆的使用确保了ID请求的对数计算复杂度；删除一个持久性对象会将其ID放回FPool中。

*3.2.3 限制。*OPointer的主要弱点是对OTB的频繁而缓慢的访问。它为每个从其他区域引用的对象分配了一个OID；因此，OTB的大小可能很大。每一个解除引用都需要访问OTB。OTB的大尺寸可能会导致许多缓存缺失，因此，解指也很慢。

3.3 提案二。多尺寸的GPointer（SGPointer）。

多尺寸GPointer是OPointer的一个变种。它通过将对象分组并在分组对象中共享组ID（GID）来减少OTB的大小。

在多尺寸GPointer中，一个对象组是一个连续的内存空间，充满了两种类型的内存块：（1）可以分配给对象并由内存管理部门管理的空闲块；（2）头在组中的分组对象。对象组具有以下属性。

- 一个对象组是数据移动的最小单位。
- 一个对象的跨度可以超过它所属的组的末端。
- 一个小组不能与另一个小组重叠。
- 小组的规模是预先确定的。

第一个属性确保在数据移动后，对象到组的起始地址的偏移量保持不变。有了这种设计，指针现在可以由三部分组成，RID GID Offset，其中RID是NVRegion ID，GID是对象的Group ID，Offset是对象到组的基本地址的偏移量。现在，OPointer中大的OTB和AOTB被小得多的GTB（组ID到组的基地址）和AGTB（反向GTB）所取代，导致更好的数据定位，因此，缓存性能。在数据移动时，唯一的更新是对GTB和AGTB中组的基础地址的更新。由于组中的数据偏移量保持不变，它们可以通过原来的GPointer进行检索。

第二个属性确保GPointer即使对大于预定的组大小的对象也能工作。图7(b)说明了一个256-B的组。该组中的最后一个对象跨越了边界。

第三个属性很容易理解。最后一个属性对于有效的组和对象管理是必要的。一个问题是应该使用什么尺寸。一个大的尺寸可以减少GTB和AGTB的大小。这可以提高缓存的性能，但会使组更容易被分割。(当一个对象在组的中间被释放时就会发生碎片化）。

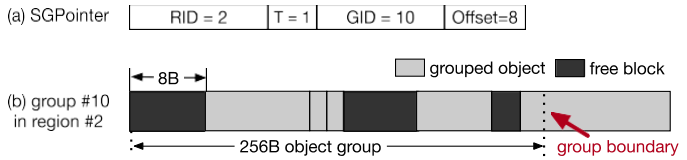


图7.SGPointer(a)和其相关的组(b)。SGPointer指向该组中的第一个对象。

多尺寸GPointer通过为一个对象组提供多种尺寸选择（在我们的实现中为1 B、256 B、4 KB和64 KB）来解决这个问题。为了实现这种灵活性，一个多尺寸的GPointer格式被设计为包含四个字段：前30位构成RID，后面两个字构成T字段。T可以是0-3，分别对应于我们实现中的1 B、256 B、4 KB和64 KB大小。剩下的位构成G和O字段，用于GID和组内偏移。

助理数据结构需要支持多尺寸设计。它为AGTB使用了一个排序的二进制树（与AOTB类似，它将区域内的偏移量映射到GID），并使用一些数据结构的四个副本，每个组类型一个。具体来说，GPointer使用四个GID表（GTB）和四个FPool（对组类型*i*表示为*GTBi*和*FPooli*，*i*=0、1、2、3）。此外，GPointer将一个NVRegion的四个GTB的基地址和该区域的基地址一起安装到RTB表中。GTBs存储了各组到该区域基底的偏移量；它们是跨程序共享的。NV区域的元数据根据数据结构数量的增加而扩展。作为一种优化，多尺寸GPointer使用一个AGTB而不是四个AGTB。它通过将GID填充到32位并将组类型与GID连接起来，将每个区域内的偏移量映射为由组类型和GID组成的值。第3.6节将解释如何选择适当的组，一个新创建的对象应该被放入。

3.4 建议三：多级GPointer（LGPointer）。

尽管多尺寸GPointer在一个对象可能进入的组中提供了一些灵活性，但它有一个刚性的设计。一个组的大小是固定的。如果一个大的组碰巧遭受了严重的碎片化，它就不能被分解成更小的组来缓解这个问题。

我们引入了多级GPointer来增强GPointer在组大小上的动态适应性。在这种设计中，在运行时，一个对象组可以被分割成多个较小的组，多个组可以合并成一个大组，所有这些对应用程序、程序员或用户都是透明的。

*3.4.1 指针和数据结构。*图8上面的方框显示了一个多级GPointer的结构。前32位定义了RID。接下来的32位被分为四个字段，P0到P3，它们是相关GTB的索引，接下来详细介绍。

多级GPointer中灵活的组大小的一个关键因素是GPointer的第一个（最重要的）位。我们称该位为*类型位*。P0是第一级GTB的索引，即GTB0。GTB0中的一个条目可能是两种类型之一。如果它的类型位是0，该条目是NVRegion中下一级GTB（GTB1）的偏移。如果该位是1，它是NVRegion中一个16MB的对象组的偏移量，指针P1_P2_P3的后缀构成该16MB对象组中的对象的偏移量。GTB1和GTB2的设计与GTB0相同，只是它们对应的对象组大小为64KB和256B，后缀分别为P2、P3和P3。图8给出了一个简单的说明。GTB3中的每个条目只能是一个1-B的对象组的偏移。(因为它是1 B的大小，所以不需要组内偏移)。

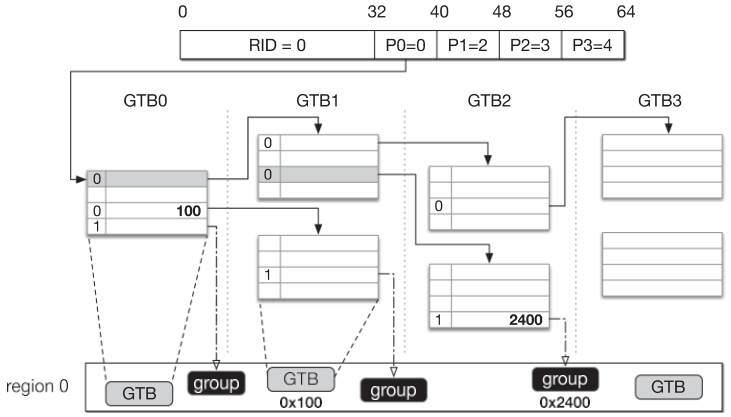


图8.多级GTBs（第3.4节）。

对于一个NVRegion来说，只有一个GTB0，但GTBi的数量是n_{i-1}（n_{i-1}是所有GTB(i-1)的条目总数；i = 1, 2, 3）。

GTB被保留在与其他数据混合的区域中，如图8的底部方框所示。运行时根据多级GPointer中的字段走过GTBs。

多级GPointer中其他辅助数据结构的使用与多级GPointer中类似。

所描述的多级指针的设计需要四种可能的组大小--1 B、256 B、64 KB和16 MB--对应于对象组的四个级别。一个对象组的完整ID是指针中各组ID字段的连接。例如，如果该组是第2级组，P0和P1的连接构成了它的ID。

*3.4.2 组的拆分和合并。*多级GPointer的一个吸引人的特性是，它允许轻松、有效地分割和合并对象组。组大小的动态可调整性为加强GTB访问中的位置性和组中的碎片之间的权衡提供了机会。

*分割。*即使在程序运行时，也可以随时采用分割操作。运行时首先定位要拆分的对象组的GTB条目（例如e）。然后，它准备一个下一级的GTB，其中有每个子组的偏移量被填充。它最后更新e，用新的GTB的区域内偏移量替换它，并将e的类型位设置为0。

图9说明了拆分0x01A0组的过程。首先，系统分配了一个新的三级表GTB2，如图中右侧所示。然后，它用区域内的偏移量填充该表，从0xABC开始，以0x100的步长增加，这相当于新组的大小，256B。

新的小组的GID从0x01A000到0x01A0FF；GID的最大共同前缀是大组的GID。

*组合并。*组合并将小组合并到一个大组中，以减少GID的数量。这个过程与组拆分相反。合并需要避免空间中的组冲突。考虑到图7中所示的对象组。该组的最后一个对象超过了组的边界。如果运行时将另一个组紧跟在所示组之后，该对象可能与新组中的另一个对象重叠，导致空间冲突。

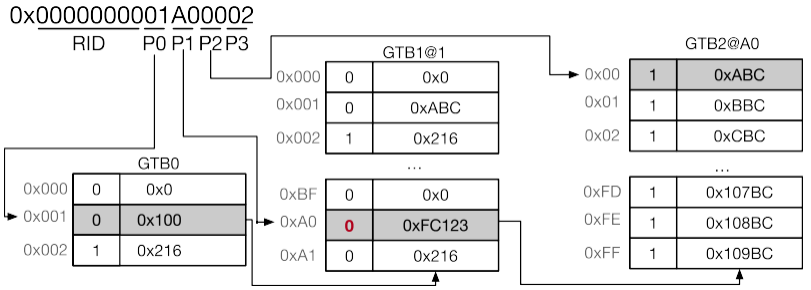


图9.将0x01A0组分割成ID从0x01A000到0x01A0FF的子组（3.4节）。

一组对象组*S*可以被合并，当且仅当它们满足以下条件。

- 它们的大小是一样的。
- 他们的ID形成一个连续的序列。
- *L*在二进制表述中是*S*中ID的最大公共前缀，其中*L*是*S*中所有组的大小之和的二进制表述，例如，组0x0010只能是组0x001000至0x0010FF的合并结果。
- 合并不会对*S*的任何两个小组造成冲突。

直观地说，这些条件确保*S*可以在更高层次上合并成一个单一的组。

3.5 硬件支持

这三种方法可以通过系统软件应用于现有硬件。有了额外的硬件支持，它们可以变得更有性能。这三种类型的指针都以使用指针重定向为中心，因此，它们都会受到重定向开销的影响。我们提出了一些硬件特性来加速指针的转指。主要的想法是在转指中避免基于软件的位操作，用专用的旁观缓冲器来缓存转换。对指令集的相应改变不超过先前的NVM硬件支持[55]，也就是为NVM访问增加指令*nvld*和*nvst*。

*对OPointer的硬件支持。*如图10所示，在这个设计中引入了一个区域-对象转换旁观缓冲器（ROTLB）。它将OPointers翻译成虚拟地址，然后将它们传递给TLB。ROTLB中的每个条目都是一个OPointer-虚拟地址对。如果一个OPointer与ROTLB中的任何条目都不匹配，翻译就会采取慢速路径❶在慢速路径中，硬件从OPointer派生出区域ID和对象❷ID，加载区域ID的地址。翻译表来自一个新的寄存器*ncr.0*，该寄存器存储每个进程RTB的地址，并找到区域的虚拟地址和OTB的虚拟地址（对象ID翻译）。❸从OTB（对象ID转换表）中加载区域内的偏移量，以及❹表），将区域地址和偏移量相加，产生对象的虚拟地址。然后，它将虚拟地址缓存在ROTLB中，并将其发送到TLB进行进一步操作。

ROTLB是分层次的。一个小的快速的L1 ROTLB被一个大的慢的L2 ROTLB所支持，两个ROTB都是设置关联的。(第4节提供了一个关于它们大小的敏感性研究。) RTB和OTB是通过虚拟地址访问的，允许操作系统在必要时交换OTB的内容。

ROTLB点击的延迟与TLB点击的延迟相同。一个L2 ROTLB缺失会对每个RTB和OTB产生一次内存访问。我们利用以前的一个名为POTB的设计[55]来缓存RTB，以进一步减少内存访问。

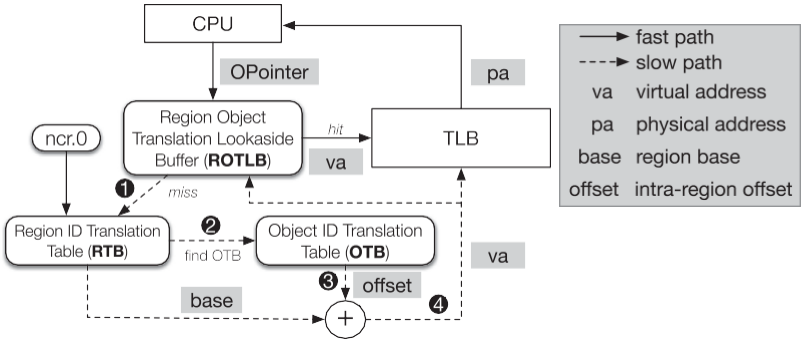


图10.区域-物体平移的旁观缓冲器。

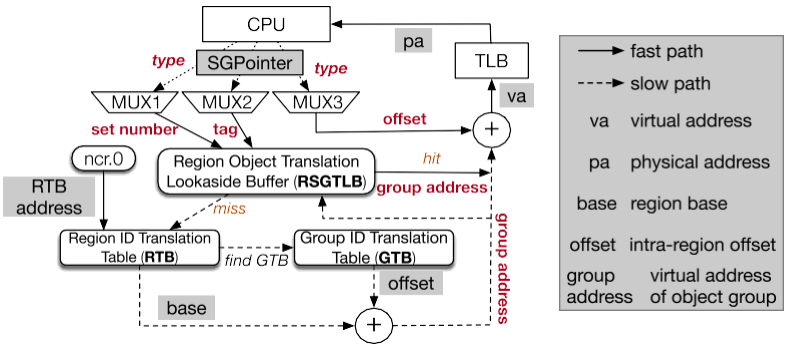


图11.区域多尺寸组翻译查找缓冲器。

*对SGPointer的硬件支持。*这个设计引入了一个区域多大小的组转换旁观缓冲器（RSGTLB），如图11所示。它将所有组大小的SGPointers翻译成对象组的虚拟地址。该设计引入的所有三个多路复用器将SGPointer的2位类型字段作为选择器输入。它们从指针中选择不同的位。具体来说，*MUX1*从GID字段中提取7个最小有效位，对于1-B组中的指针来说是0-6位，对于256-B组中的指针来说是8-14位。*MUX2*通过选择除组内偏移外的所有字段来生成用于搜索的标签。*MUX3*选择组内偏移字段。这些操作增加的延迟可以忽略不计，因为多路复用器的电路深度小于3。

设计的其他部分与OPointer的硬件支持类似（图11），只是GTB取代了OTB。因此，这两种设计有着相似的性能。

*LGPointer的硬件支持。*对于LGPointer，RSGTLB并不适用，因为组的大小是与指针解耦的。该设计使用了四个缓冲区，每个缓冲区用于四个级别的指针中的一个。在指针转换时，硬件同时在所有的缓冲区中搜索。如果搜索到任何缓冲区，它就把虚拟地址转发给TLB，否则就采取慢速路径。

硬件开销。与之前为NVM增加地址转换的硬件建议类似[55]，这项工作中的硬件变化不需要修改缓存一致性。我们使用CACTI[10]来估计硬件面积开销。该设计的硬件开销包括几个逻辑电路和一个旁观缓冲器（ROTLB/RSGTLB/RLGTLB）。缓冲器的大小被设定为9KB，这样访问延迟就能满足目标配置（一

1KB L1的周期和8KB L2的七个周期）。面积成本是微不足道的，0.065*毫米²*，根据 CACTI。

3.6 特殊的复杂性

这一节讨论了三个特殊的复杂情况和我们的解决方案。前两个与SGPointer和LGPointer都有关。第三个是专门针对LGPointer的。

*分组。*对于这两种GPointers，一个复杂的问题是决定哪些对象应该被分组。主要考虑的是内存碎片。如前所述，对对象进行分组有助于减少助理数据结构的大小，但当某些对象在组内其他对象之前被释放时，可能会导致组内的内存碎片。GC不允许在一个组内打包对象，因为整个组是最小的移动单位。因此，对对象进行分组的原则是将具有相似寿命的对象分组。

之前有很多研究提出了预测对象寿命的方法[14-16]。以前的工作中常见的一个简单而有效的启发式方法是，在同一分配点创建的对象往往具有相似的寿命。我们的实现采用了这种分组策略。

*选择组的大小或级别。*对于SGPointers和LGPointers来说，当要创建一个新的组时，必须决定这个新的组应该有多大（或在什么级别）。我们通过解释新对象被分配时的整个过程来介绍我们的解决方案。

在分配一个新的对象时，分配器将检查坐在这个对象之前的组（称为候选组）是否有与它相同的寿命，如果有，它试图将该组ID分配给这个对象（通过设置其地址的一些位），如果该组仍然有足够的空间。如果空间不足，它就创建一个新的组（通过从FPool获得一个空闲的组ID），并将该新组ID分配给这个对象。注意，如果可能的话，新组被设置为比候选组高一级的大小；其理由是，候选组的填充表明，在该分配点可能会有更多的对象被创建。另一方面，如果候选组有不同的寿命，则为该对象创建一个最小尺寸的新组。另一个方案是用一个固定的组大小来代替自适应的组大小。第4.5节给出了这些方案的经验性比较。

*选择组的ID。*对于SGPointers，一个新创建的组可以使用任何自由的组ID。然而，对于LGPointers来说，情况就比较复杂。一个错误的GID选择可能会阻止两个组在未来的合并。考虑到两个相邻的16-B对象，每个都被放入一个1-B组。(如果这两个组的ID是0和1，它们的P3字段将分别等于0x0和0x1。虽然当它们是1-B组时，这样做很好，但当这两个组被合并到一个更高级别的组时，它们的P3字段将被视为更大的对象组中两个对象的偏移量，这将导致错误（第二个指针将指向第一个对象的中间而不是第二个对象）。因此，对于多级GPointers，当为一个新组挑选ID时，运行时确保新ID与候选组（例如*X*）的差距等于组*X*的实际跨度（即组*X*的*max（EndOfObjects, EndOfGroup）-StartOfGroup*）。

*尺寸限制。*NVPointer的格式对NVRegion的大小形成了一些限制，这对于先前的可重定位指针来说也是如此[19, 55, 56]。正如先前的工作[19]所建议的那样，为了增加灵活性，我们可以设计多种格式，对区域ID和其他字段采用不同的位数，这样在内存系统中，较大的NVRegions和较小的NVRegions可以并存。我们把这个扩展留给未来的工作。

表2.操作的计算复杂度

		OPointer	Multi-sized 多级ĂĂĂĂ
解除引用	2	2	2-5
初始化	O(1) - O(log *n*)	O(log *n*)	O(log *n*)
移除	O(log *n*)	O(log *n*)	O(log *n*)
分割	-	-	O(*m*)
合并	-	-	O(*m*)

解除引用的复杂性是以间接数来衡量的。

3.7 计算和空间复杂度

表2显示了所提技术的计算复杂性。在持久性指针上最常见的操作是指针解指。解除引用的开销是以基于表的重定向的数量来衡量的。多级GPointers可能有几个额外的重定向，这取决于对象组处于什么级别；然而，总体而言，这些技术具有类似的时间复杂性。在实践中，解引用的开销主要由访问表的缓存性能决定。

对于指针分配，OPointer的时间复杂度是$O(1)$，只需要在AOTB上进行一次哈希查询。对于GPointers，运行时必须从FPool请求一个新的OID。由于FPool是一个最小堆，时间复杂度是O（log n），n是FPool中ID的数量。对AGTB的访问也有一个时间复杂度O（$lo∂n$），因为AGTB是在一个排序的二叉树中实现的。对于多级GPointer，组的拆分和合并的时间复杂度为O（m），其中m是合并的GTB的大小。

RTB和ARTB占用的内存空间是很小的，因为它们各自只需要一个NVRegion的条目。对于OPointer，主要的空间开销来自OTB和AOTB。对于一个NVRegion上的n个NVObjects，OTB和AOTB的大小都是8nB。对于SGPointer和LGPointer，主要的空间开销是GTB；如果$∂$是组的大小，GTB的空间开销是OPointer中的1/$∂$到1/1，取决于对象大小。然而，与Java程序中的平均对象大小（149-744 B [9]）相比，所产生的对象大小增加是 。

4 评价

本节从五个方面评估了所提出的MOA解决方案的功效。第一个方面是合理性，也就是说，在数据移动的情况下，它们是否真的能够支持MOA。关于这个方面，所有的解决方案都没有差异。它们都提供了健全的MOA解决方案，通过观察任意数据移动注入基准时内存访问返回的值来验证。因此，我们把本节的讨论集中在其他四个方面，它们都与解决方案的效率有关。

- 时间成本。由于所有提议的解决方案都用间接引用取代直接引用，重定向引入了时间开销，这一点通过这个指标来衡量。
- 缓存性能。由于解决方案是通过一些中间环节来实现重定向的，这个指标衡量的是增加的空间使用对缓存的影响。
- 对NVM访问延迟的性能敏感性。
- 内存碎片化。这是对GPointers的两种变体所特有的。由于对象被放入组中，并且不允许在组内移动，当一些对象在组的中间被释放时就会发生内存碎片。这个指标衡量碎片化的严重程度。

表3.基准列表

	基准描述
SeqSequentially	访问一个有1.79亿个对象的数组；每个对象是8B，由一个指针指向。
Rand	类似于Seq，但指针是随机洗牌的。
ListAccess	1.34亿个节点的链接列表。每个节点是一个8-B对象。
BTree	在一个BTree中插入5000万个随机键，然后进行1亿次搜索查询。每个BTree节点有两个数组，用于整数键和指向其他节点的指针；最小度数为256。
HATTrie	将2300万个维基百科页面标题[5]插入到trie和数组哈希的组合中，然后是1亿次搜索查询。trie存储字符串的前缀，数组哈希存储字符串的后缀。哈希值在需求时被分割成 trie 节点和更小的哈希值。
ClauDB	一个现实世界中的Java内存键值存储数据库，用作LRU缓存，三个工作负载的失误率分别为1%、5%和10%。

4.1 方法论

基准和数据。我们用五个Java基准对一些重要的数据结构进行评估，如表3的上半部分所列。我们之所以选择这些基准，是因为它们表现出不同的内存访问模式，鉴于它们对内存访问的关注，这对全面了解各种解决方案的性能至关重要。此外，我们将该技术应用于一个真实世界的应用，一个10,440行的键值存储ClauDB[3]，这将在第4.6节中详述。真实世界的应用预计会表现出比微观基准更复杂的内存访问模式，因为它有[4]种类型的任务。这提高了对所提技术的评估。

请注意，对NVM的编程支持对Java来说还是初步的；英特尔的持久性Java集合（PCJ）是目前最发达的支持[6]。PCJ实际上是通过JNI重用英特尔C库（PMDK），而不是提供为Java定制的支持。因此，使用它的程序因跨语言函数调用而遭受非常大的执行时间开销，使得来自MOA指针的开销看起来微不足道（在所有情况下低于1%）。因此，我们实现了一个原型的Java NVM程序性能测量框架。该框架将程序中的NVM访问替换为本地实现，而不是JNI函数调用，因此，避免了JNI带来的人为的跨语言开销。

所有的基准都有一个单区域和四区域的版本。我们假设整个GC管理的堆在NVM上，其余的（如堆栈）在DRAM上。数据集的大小与区域的数量成正比，数据以轮流的方式分配到NVRegions中。Seq和Rand分别占用了1.5GB和2GB的内存。

机器配置。我们运行真实系统实验（针对我们所有的软件方案）以及仿真实验（针对我们所有的硬件方案），配置见表4的上半部分。我们使用真实系统来评估所有的软件实现，使用处理器仿真模型来评估硬件实现。真实系统是一个英特尔i7-6700K CPU系统。我们的实现利用一条指令`bextr`来有效地从一个字中提取 数量的连续比特。我们使用gcc 8.2编译器并将参数`-mbmi`设置为

[3]https://github.com/tonivade/claudb。
[4]https://dumps.wikimedia.org/enwiki/20190301/enwiki-20190301-all-titles.gz。

表4.真实机器和模拟配置

真正的机器平台（用于评估软件解决方案）。	
处理器。 TLB。 缓存。	英特尔i7-6700K，四核，3.4 GHz（涡轮增压4 GHz）。 L1：4路，64条；L2：4路，1,536条 L1：8路，32KB；L2：4路，256KB；L3：16路，8MB
仿真模型（用于评估基于硬件的解决方案）。	
CPU。 TLB。 缓存。 内存。	4 Ghz；ROB：352条；加载队列。128个条目。存储队列。72个条目；分支预测器：双模态L1：4路，64个条目，1个周期；L2：4路，512个条目，8个周期；页表行走。100个周期 L1：8路，64KB，5周期；L2：8路，256KB，10周期；L3：8路，2MB，20周期 DRAM。CAS 12.5 ns (50周期), RCD 12.5 ns, RP 12.5 ns；NVM：75 ns（300次）。
ROTLB/ RSGTLB/ RLGTLB	L1：4路，64条，1个周期；L2：4路，512条，1个周期。 试，8个周期；表走。ROTLB 120次，RS- GTB 110次；RLGTLB 150次

启用编译器对该指令的支持。硬件运行Ubuntu操作系统18.04.2 LTS，带有4.15.0内核。我们加载了内核模块PMEM驱动，并将16GB作为持久性内存。我们使用英特尔PMDK的低级别API来管理NVM区域。我们使用openjdk 1.8.0_242和G1 GC来处理Java程序。G1 GC是一个用C++实现的生成性GC。

为了评估我们的架构支持，我们使用了一个基于跟踪的模拟器Champsim[1]，其参数显示在表4的底部。Champsim的准确性在最近的工作中得到了验证[32, 61, 62]。该模拟器模拟了一个失序的处理器和TLB、高速缓存和内存子系统的详细操作。页表行走延迟被建模为一个固定的100个周期的TLB失误惩罚。我们还模拟了具有固定延迟的OTB/GTB行走。我们为每个程序模拟了10亿条指令的执行。

在实际机器上测量Java程序的性能时，我们在测量稳定执行时间之前先进行预热，以避免Java运行时的非确定性行为。在我们的实验中，我们重复收集了10次稳定执行时间（在3次预热运行之后）；观察到了边际变化，并报告了平均性能。

4.2 执行时间的开销

为了进行性能比较，我们使用先前提出的可重定位的NVPointer（本文称为*RPointer*）[55]作为基线。一个RPointer由RID Offset组成，其中RID是一个NVRegion的ID，Offset是该区域中对象的偏移量。这个方案支持所有NVRegion的重新定位，但不支持NVR- gion内GC触发的数据移动。因此，它不支持MOA指针。通过与这个基线的比较，我们衡量了支持MOA所带来的额外开销。我们选择RPointer，因为它为在可重定位的NVRegions中检索对象提供了最先进的性能。

图12报告了我们的三种MOA指针在没有硬件（（a）和（b））和有硬件支持（（c））的情况下

的时间开销。在实验中，我们将LGPointer限制在不大于64KB的组中使用。否则，低级表很少被使用，LGPointer将显示出与RPointer类似的性能。

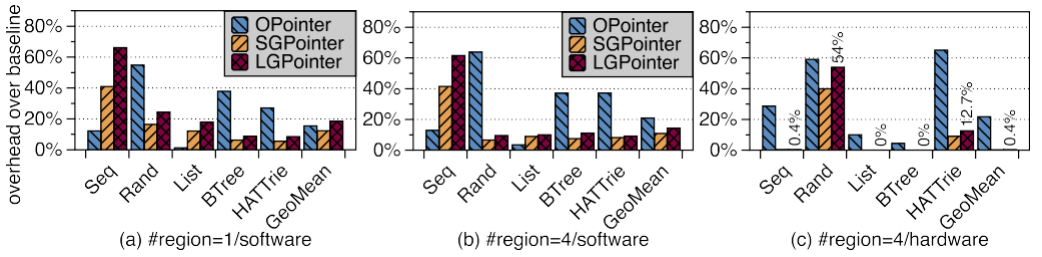


图12.与RPointer[55]相比，MOA在软件（a和b）和硬件支持下（c）的执行时间开销。

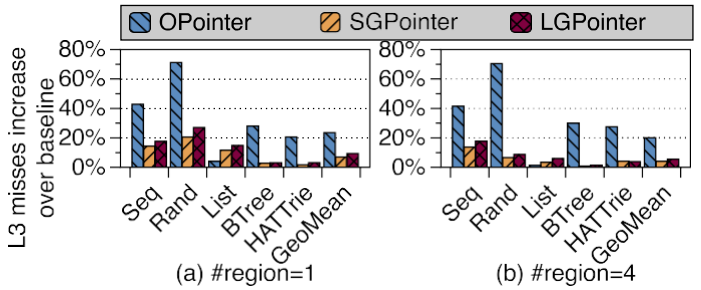


图13.相对于RPointer，软件MOA方案在真实机器上增加的L3缓存错过次数。

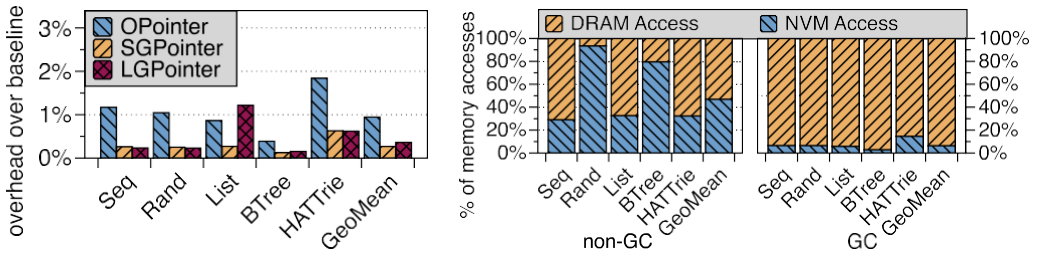
图12的(a)和(b)部分显示，有一个或四个NV区域的结果相似，平均产生14%到21%的开销。与OPointer相比，SGPointer大大降低了开销，特别是对于Rand、BTree和HATTrie，这要归功于分组对象。然而，在Seq和List上，OPointer的表现比SGPointer和LGPointer更好。对于Seq来说，由于定期的数据访问，因此，数据定位，主要的开销来自于MOA指针的额外操作，而不是缓存丢失；OPointer的额外操作最少。对于List来说，节点是随机放置在内存中的，并且是按照节点创建的顺序连锁起来的。因此，对节点的访问是随机的，而对助理数据结构的访问是按顺序进行的。因此，OPointer表现得更好，因为OTB上的缓存缺失被缓慢的数据访问所掩盖，而且它的位操作比SGPointer和LGPointer的少。

图12(c)显示，硬件支持是非常有效的。它将SGPointer中的MOA点的成本降低到平均开销的0.3%（或LGPointer的0.4%）。由于硬件支持利用了空间或时间定位，但Rand产生了一个完全随机的访问路径，它代表了硬件性能的病理情况，特别是对OPointer，它有一个大的OTB。

图13报告了每个基准的L3缓存失误的增加（通过PAPI[51]收集）。图中显示，SGPointer和LGPointer实现了比OPointer小得多的增长，表明分组的有效性。

4.3 垃圾收集间接费用

本小节详细介绍了在我们的硬件支持下，MOA指针对JVM中使用的G1 GC产生的开销。在GC执行过程中，内存访问被排放到内存跟踪中。



(a) Time overhead on GC (w/ hardware extension). (b) Memory access breakdowns by GC and application (non-GC).

图14.GC上的开销。

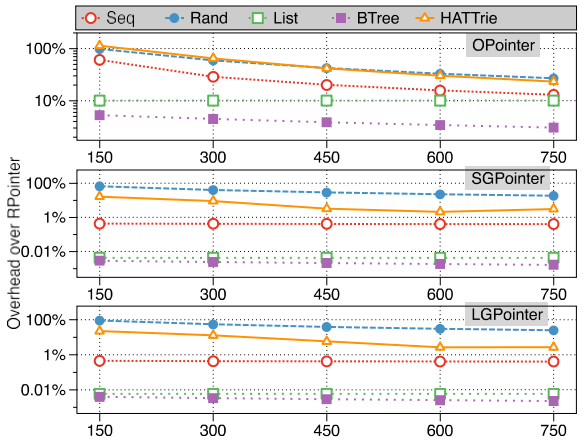


图15.NVM访问延迟（周期）对OPointer、SGPointer和LGPointer的执行时间开销的影响，有硬件扩展。

这将成为模拟器的输入。在每个基准的执行过程中，通过调用*System.gc()*，GC被提示运行10次；被销毁的对象的比例从10%增加到100%。表3显示了销毁前的对象总数。图14(a)报告了在RPointer上执行GC的执行时间的开销。该图显示了几乎可以忽略的开销（SGPointer和LGPointer的开销小于1%，OPointer的开销略高），低于基准执行。图14(b)显示，与非GC执行相比，GC产生的NVM访问量要小得多；因此，它受MOA指针开销的影响要小得多。

4.4 敏感性研究

图15报告了当我们改变NVM访问延迟时硬件MOA的开销。该图显示，SGPointer和LGPointer的开销仍然很低（在大多数情况下<10%），，对NVM的延迟不敏感，即使延迟是DRAM的5倍。

4.5 内存碎片化

虽然分组有助于GPointers减少内存消耗，但它也可能增加内存碎片。为了研究这种影响，我们进行了一个实验来测量内存碎片。

表5.固定组大小（1KB，8KB）、自适应大小和最佳大小（OPT）的内存碎片化情况

记忆释放模式参数。	记忆。分裂			#群体	
	小组的规模				
纳米	1 KB	8 KB	自适应	适应性强	OPT
11	1	1	0	8,388,608	8,388,608
3131	0.10	1	0.71	1,262,800	2,706,004
6464	0	0	0.33	589,824	262,144
150100	0.02	0.30	0.19	536,871	738,199
随机一半	0.87	1	0.81	2,593,267	6,709,760

越低越好；范围是[0,1]。在1KB和8KB的情况下，#组分别为4,194,304和524,288。

我们研究了两种固定的组大小（1KB，8KB）和第3.6节中描述的自适应大小的使用。

该实验借用了一个名为Fragger[45]的合成基准，它通过首先用200B或更大的小对象填充内存，然后释放一些，并通过分配尽可能多的大对象来测量碎片整理的质量，从而实现碎片整理的最大化。我们遵循第一步，除了按照GPointers的要求，给每个创建的对象分配一个组ID。然后，我们通过在每$n+m$个对象的末尾重复释放m个对象来扩展基准（默认的Fragger是我们方法的一个特例，$n=m=1$）。在最后一步，我们不分配大的对象，而是模仿GC，把没有活对象的组移到一起，形成一个大的自由内存空间（注意，GC移动组，但不移动组内的对象）。该内存将由一些自由空间组成，每个空间都被一些活的对象所包围。让H是最大的自由空间的大小，让A是总的自由空间。一个用来衡量内存碎片化的指标是

$$分裂=\frac{1H}{A}.$$

在我们的实验中，我们将内存空间设置为4GB，小对象的大小为256B。表5显示了使用固定组大小（1KB，8KB）或自适应大小时，GPointers的碎片和相应的组数。作为参考，我们还在OPT栏中显示了使碎片最小化的组数，该组数是基于对创建对象的位置和寿命的充分了解而获得的。不同的n和m值创造了不同的内存释放模式。我们进一步增加了另一种释放对象的模式，即随机释放所有对象的一半。它在表的底部表示为 "RandomHalf"。

因为GC不会在组内移动对象，所以较大的组往往会遭受更严重的碎片化。这解释了8KB比1KB有更大的碎片。另一方面，较小的1KB组需要比使用8KB组多8倍的对象组（因此，OID缓存性能更差）。"自适应 "方法在两者之间进行了权衡。当n和m的值恰好使空位与固定大小的组的边界完全一致时（例如，$n = m = 64$），固定大小的组方案工作得很好。然而，在一般情况下，自适应方案工作得更好，因为它的自适应方法是有意识地对跨越组的边界的对象 （第3.6节）。

4.6 真实世界的应用。ClauDB

评估使用了ClauDB的三个工作负载，它们分别产生1%、5%和10%的失误率。每个工作负载都是SET和GET操作的混合；性能的测量是在

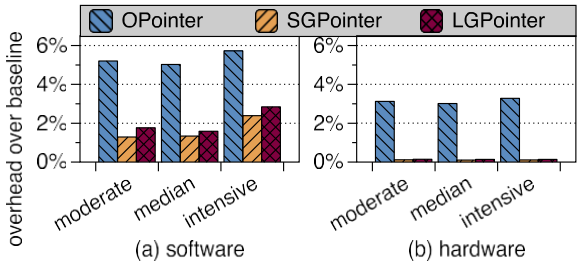


图16.ClauDB与软件和硬件MOA解决方案的性能开销；越低越好。对于硬件MOA，NVM的访问延迟是600个周期，是DRAM的3倍。

每秒的GET数量。我们评估了MOA so-lutions引起的性能下降。

图16显示了三种工作负载的结果。所有的软件MOA解决方案在*密集型*工作负载上的性能下降都略多，因为有更多的持久性对象创建和解构。SGPointer的性能略优于LGPointer，而两者都保持了97%的per- formance。OPointer因其大型助理数据结构而产生了5%的退化。所有的硬件MOA解决方案在所有的工作负载中都保留了99%的基线性能，而OPointer产生的开销大约是其他两个解决方案的两倍。

5 相关的工作

自愈屏障[22]是一种GC技术，用于解决非阻塞性GC中的数据移动。它为在GC期间移动的所有对象创建了一个转发表，这样，尚未更新的指针可以通过转发表找到这些对象。一旦所有的指针被更新，转发表就被删除。这种技术对于这项工作中的问题是不够的。由于GC不知道来自当前这个应用视图之外的区域的指针，它将无法删除转发表中的条目。该表将不断增长，面临OPointer所面临的问题。

最近有几项关于位置无关指针的研究[7, 12, 13, 19, 41, 55, 56, 68, 70]，如Intel PMDK[7]和Twizzlers[12]。然而，所考虑的情况只是整个NV区域的起始地址的变化。当对象被移动到NV区域内的不同位置时，他们的解决方案并没有保留对象的可寻址性，因为他们假设对象对其NV区域的起始地址的偏移量保持不变。例如，在Wang等人[55]以前的工作中，作者将区域ID和对象在该区域的偏移量存储在一个指针中，使用硬件来加速向虚拟地址的转换。PMDK[7]和Twizzler[12]采用相同的指针格式。这些方法在有GC的情况下会失效：如果对象在该区域被GC移动，指针会变得无效。相比之下，这项工作提出了一个新的概念，即移动无关寻址，通过重新设计指针结构和翻译机制来实现该属性，并进一步开发了四种新的方法，缓解了移动无关寻址的空间和时间成本。

Chakrabarti等人[18]提到，NVRegions应该在故障时被垃圾收集，并且
设想了基本用途，但没有给出设计方案。NV-Heaps[23]使用基于参考的GC来检查可达性。Makalu[11]探讨了故障后的恢复时间。Cohen等人[24]提到，由于碎片化或对象大小的变化，数据迁移是必要的；他们使用一个完全离线复制的GC，扫描与根对象相连的对象。该方法仅在两个区域同时加载时支持跨区域指针，当区域与

跨区域指针基本上可以被看作是一个单一的实体。DwarfGC[36]是专门为碰撞一致性设计的，没有考虑跨区域指针。

一个相关的问题是使用共享内存时的GC。使用共享内存的程序是合作进程。GC是通过协调同步和对参与进程的整体控制来实现的。例如，XMem[57]在移动共享对象之前同步所有程序，Skyway[44]在服务器之间迁移共享对象，并使用一个集中的服务器来跟踪所有对象。其他系统要么使用非移动的GC[8, 58]，要么使用全系统GC[47]。相比之下，一个NVM对象可能被许多其他NV区域的对象所指向，而使用该对象的程序的GC可能不知道或没有对其他区域的这些指针的访问权限。因此，合作解决方案并不适用。

之前还有许多关于NVM编程的研究。他们针对其他方面，包括容错[23, 31, 53, 69]，编程模型[26, 39, 40, 71]，算法[52]，安全[65, 66]，等等。他们做出了重要的贡献，但没有解决这项工作中的问题。

AutoPersist[48]和GCPersist[63]实现了对象在DRAM和NVM之间的移动，以实现自动数据持久化，但没有考虑其他应用程序创建的其他区域的跨区域指针。Espresso[64]为Java提供了一个崩溃一致性的堆，但没有解决本文研究的可寻址性问题。

目前英特尔的Java对NVM的支持（PCJ[6]）是不完整的。它将持久性对象作为不被垃圾收集的堆外数据进行管理，通过C库PMDK提供持久性和可寻址性。其他项目要么有类似的限制[5]，要么是研究项目，没有向公众提供可运行的框架[48, 64]。

将DRAM与NVM[20, 21, 38]结合起来形成一个扁平的主存储器，是一种最先进的架构，它结合了持久性和性能。在最终更新NVM之前，DRAM被用来缓存耐用数据。在更新NVM中的持久数据方面，混合DRAM/NVM架构面临着与纯NVM存储器相同的可寻址性问题。只要亲们采用第2节中介绍的访问模型，这项工作中提出的解决方案对混合架构也是有效的。

指针分析[29, 30, 54, 60]从源代码中推断出一个指针指的是哪个变量。它有可能通过将一些指针检测工作卸载到静态代码分析中来增强拟议的技术。然而，由于内存的模糊性、别名和其他代码的复杂性，一般程序很难精确地进行指针分析。当运行时系统可以移动对象时，该分析也是具有挑战性的。

在其他工作中[17, 37]，定向表被用于其他目的，如崩溃一致性保证和快速数据持久性。例如，HOOP[17]采用了一种就地更新（OOP）的方法，它将持久对象的更新放在另一个NVM位置--OOP区域。当程序访问原始对象的地址时，HOOP通过一个重定向表将访问重定向到OOP区域。OTB中的重定向表的使用是不同的。首先，HOOP采用了物理到物理地址的映射，而OTB采用了OID指针到虚拟地址的映射--这在GC引起的移动中是必要的。因此，HOOP将重定向表放在TLB之后。然而，OTB必须在CPU流水线中把它放在TLB之前。其次，HOOP可以通过将OOP区域的就地更新应用到原始数据上，透明地从重定向表中删除条目。因此，它可以保持重定向表的小尺寸。其他的原地外更新工作[37]也采用了类似的优化。另一方面，OTB不允许这种用户程序透明的操作，因为它只能在被指向的持久对象被解构或GC保证将所有的OID指针刷成虚拟地址时，才能删除OID指针的OTB条目，这很难实现，如图1所示。因此，OTB比HOOP中的重定向表大得多，面临着高效地址转换的独特挑战。

所提出的设计中的一些想法从翻译旁观缓冲器（TLB）[50, 67]中获得了灵感。然而，他们解决了一个新的问题，即NVM上的GC。所提出的设计与TLB有根本的不同：它们以不同大小的对象的粒度来翻译地址，而不是以固定大小的内存页来翻译。这种差异导致了效率和正确性方面的特殊复杂情况。例如，一个对象可能跨越组的边界，如图7所示。因此，我们提出了一套优化方案来处理这些复杂的问题，包括为OPointer提供一个直接的in-dexed表，为LGPointer提供分组和拆分机制，以及第3.6节中描述的其他考虑。

6 讨论

*移动和非移动的GC。*虽然非移动的GC效率高，实现起来也比较简单，但它们并不能缓解内存碎片化。在NVM上，内存碎片变得更加普遍和严重，因为持久对象的寿命预计会很长。任何销毁、改变大小和创建持久数据的行为都可能增加内存碎片，随着时间的推移，内存碎片会越来越严重。过去关于基于磁盘的文件系统的经验[4]，如对现实工作负载的研究[25]，表明文件系统很容易随着时间的推移变得严重碎片化。NVM上的持久对象具有与文件相似的寿命，但却要比文件更频繁地更新，因此，为NVM移动GC的重要性。

*相对于普通程序，GC的成本。*先前的研究[16, 33, 48, 63]表明，GC对NVM程序产生了1.01%到34.8%的运行时间开销。一般来说，硬件加速的GC，如P-Inspect[33]，会产生边际的开销。本工作中提出的技术会产生0.4%（硬件解决方案）或21%（软件解决方案）的性能开销，如第4节所示。

*写入持久性。*建议的技术给NVM带来的额外写入流量可以忽略不计。他们只在持久的对象创建、移动和销毁时更新表1所列的数据结构，这比对象更新的频率低得多，也便宜得多。例如，在SPECjvm 2008[35]中，这些操作对每个对象产生24字节的写入流量，远远小于303.2字节的平均对象大小。比这更多数量级的对象更新。

7 结论

本文指出了在GC触发的数据移动情况下NVM的数据可寻址性问题。它提出了MOA的概念，并开发了在NVRegion内定位所需更新的解决方案，以保持NVM对象的完全可寻址性，即使该对象被GC移动到NVRegion的不同位置。我们提出了纯软件和基于硬件的解决方案。实验表明，MOA可以通过多尺寸GPointer和多级GPointer有效实现。

鸣谢

我们感谢匿名审稿人的反馈。本材料中表达的任何意见、发现和结论或建议都是作者的观点，不一定反映国家科学基金会的观点。

参考文献

[1] 2017.第二届卡奇替换锦标赛。2022年2月10日，检索到https://crc2.ece.tamu.edu/。
[2] 2021.Java中的数据清洗工具。2022年2月10日，检索到https://openrefine.org/。
[3] 2021.Java中的数据可视化工具。2022年2月10日，检索到https://www.tableau.com/。
[4] 2021.文件系统碎片化。2022年2月10日，检索到https://en.wikipedia.org/wiki/File_system_ fragmentation。
[5] 2021.管理的数据结构。2022年2月10日，检索到https://github.com/HewlettPackard/mds。

[6] 2021.Persistent Collections for Java.2022年2月10日，从https://software.intel.com/en-us/articles/java- support-for-intel-optane-dc-persistent-memory检索。

[7] 2021.持久性内存开发工具包。2022年2月10日，检索到https://pmem.io/pmdk/。

[8] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit.2015.ThreadScan。自动和标度，能够进行内存回收。在*第27届ACM算法和架构中的并行性研讨会（SPAA'15）论文集*。Association for Computing Machinery, New York, NY, USA, 123-132.

[9] David F. Bacon, Perry Cheng, and V. T. Rajan.2003.在 metronome中控制碎片和空间消耗，一个Java的实时垃圾收集器。In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES'03)*, (San Diego, California, USA).ACM New York, NY, USA, 81-92.

[10] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas.2017.CACTI 7：创新片外存储器中互连探索的新工具。*ACM Transactions on Architecture and Code Optimization* 14, 2 (2017), 14.

[11] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J.Boehm.2016.Makalu:非易失性内存的快速可恢复分配。在*2016年ACM SIGPLAN面向对象编程、系统、语言和应用国际会议（OOPSLA'16）的论文集中*，（荷兰，阿姆斯特丹）。Association for Computing Machinery, New York, NY, USA, 677-694.

[12] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller.2020.Twizzler: 一个以数据为中心的 非易失性内存的操作系统。在*USENIX年度技术会议（USENIX ATC'20）*上。USENIX协会，美国，65-80。

[13] Daniel Bittman, Peter Alvaro, and Ethan L. Miller.2019.一个持久的问题：管理NVM中的指针。In *Proceed- ings of the 10th Workshop on Programming Languages and Operating Systems*, (Huntsville, ON, Canada).Association for Computing Machinery, New York, NY, USA, 30-37.

[14] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss.2001.为java预装。In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, (Tampa Bay, FL, USA) 。Association for Computing Machinery, New York, NY, USA, 342-352.

[15] 罗德里戈-布鲁诺和保罗-费雷拉。2017.POLM2。针对热点大数据应用的对象寿命感知内存管理的自动剖析 。在*第18届ACM/IFIP/USENIX中间件会议（Middleware'17）论文集*，（内华达州拉斯维加斯）。Association for Computing Machinery, New York, NY, USA, 147-160.

[16] Rodrigo Bruno, Duarte Patricio, Jose Simao, Luis Veiga, and Paulo Ferreira.2019.用于 延迟敏感的大数据应用的运行时对象寿命分析器。在*第14届EuroSys会议（EuroSys'19）的论文集中*。计算机协会，美国纽约，纽约，16页。

[17] 蔡淼，Chance C. Coats，和黄健。2020.HOOP：非 易失性存储器的高效硬件辅助原位更新。在*ACM/IEEE第47届计算机结构年度国际研讨会（ISCA'20）上*。IEE, 584- 596.

[18] Dhruva R. Chakrabarti, Hans-J.Boehm, and Kumud Bhandari.2014.Atlas。利用锁实现非易失性内存的一致性。*在ACM面向对象编程系统语言国际会议论文集 & Applications（OOPSLA'14）*，（美国俄勒冈州波特兰）。Association for Computing Machinery, New York, NY, USA, 433-452.

[19] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu.2017.对非易失性存储器的位置依赖的有效支持。在第*50届IEEE/ACM国际研讨会论文集《MICRO'17》中*，（马萨诸塞州剑桥市）。Association for Computing Machinery, New York, NY, USA, 191-203.

[20] 陈仁海，邵子力，刘多，冯志勇，和李涛。2019.为大数据工作负载实现高效的基于nvdimm的异构存储层次管理。在第*52届IEEE/ACM国际年会 微架构研讨会（MICRO'19）上*，（美国俄亥俄州哥伦布市）。计算机械协会，美国纽约，849-860。

[21] Renhai Chen, Yi Wang, Jingtong Hu, Duo Liu, Zili Shao, and Yong Guan.2016. vFlash：用于优化移动设备I/O性能的虚拟化闪存。*IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 7 (2016), 1203-1214.

[22] Cliff Click, Gil Tene, and Michael Wolf.2005.The pauseless GC algorithm.在*第一届ACM/USENIX虚拟执行环境国际会议（VEE'05）论文集*。(Chicago, IL, USA)。计算协会 Machinery, New York, NY, USA, 46-56.

[23] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson.2011.NV-Heaps:使用下一代非易失性存储器使持久性对象快速而安全。在*编程语言和操作系统架构支持国际会议（ASPLOS XVI）论文集中*，（美国加州纽波特海滩）。Association for Computing Machinery, New York, NY, USA, 105-118.

[24] Nachshon Cohen, David T. Aksun, and James R. Larus.2018.面向对象的非易失性存储器恢复。*Pro- ceedings of the ACM on Programming Languages* 2, OOPSLA（2018），1-22。

[25] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, et al. 2017。如何分割你的文件系统。*登录Usenix Mag.*42, 2 (2017). https://www.usenix.org/publications/login/summer2017/conway.

[26] Alan Dearle, Graham N. C. Kirby, and Ron Morrison.2009.重新审视正交持久性。In *International Confer- ence on Object Databases*, Vol. 5936, Springer Berlin Heidelberg, Berlin, Heidelberg, 1-22.

[27] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch.2018.无同步区域的持久性。*在ACM SIGPLAN第39届程序语言设计与实现会议（PLDI'18）论文集中*，（美国宾夕法尼亚州费城）。Association for Computing Machinery, New York, NY, USA, 46-61.

[28] 韩磊和张宇。2015.多任务学习中的学习树结构。In *Proceedings of the 21st ACM SIGKDD In- ternational Conference on Knowledge Discovery and Data Mining*.(澳大利亚新南威尔士州悉尼)。Association for Computing Machinery, New York, NY, USA, 397-406.

[29] Michael Hind.2001.指针分析：我们还没有解决这个问题吗？In *Proceedings of the 2001 ACM SIGPLAN- SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*.（Snowbird, Utah, USA）。Asso- ciation for Computing Machinery, New York, NY, USA, 54-61.

[30] Ming-Yu Hung, Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq-Kuen Lee.2012.支持SSA形式的proba-bilistic指针分析.*IEEE Transactions on Parallel and Distributed Systems* 23, 12 (2012), 2366-2379.

[31] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli.2016.通过JUSTDO日志进行故障原子持久性内存更新。在*第21届编程语言和 操作系统架构支持国际会议（ASPLOS'16）论文集中*，（美国乔治亚州亚特兰大）。Association for Computing Machinery, New York, NY, USA, 427-442.

[32] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson.2017.Kill the program counter:重构处理器缓存层次中的程序行为。In *Proceedings of the 22nd In- ternational Conference on Architectural Support for Programming Languages and Operating Systems*,（中国西安）。*SIGARCH Comput.Archit.新闻*45, 1, 737-749.

[33] Apostolos Kokolis, Thomas Shull, Jian Huang, and Josep Torrellas.2020.P-INSPECT：对pro- grammable非易失性存储器框架的架构支持。在*第53届IEEE/ACM国际微架构研讨会（MICRO'20）*上，（美国科罗拉多州科泉市）。计算机协会，美国纽约，纽约，美国，509-524。

[34] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch.2017.语言层面的持久性。在*ACM/IEEE第44届计算机架构年度国际研讨会（ISCA'17）*上，（加拿大安大略省多伦多）。Association for Computing Machinery, New York, NY, USA, 481-493.

[35] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger.2017.关于DaCapo、DaCapo Scala和SPECjvm2008的内存和垃圾收集行为的全面Java基准研究。在*第八届ACM/SPEC国际性能工程会议（ICPE'17）的论文集*，（意大利拉奎拉）。计算机械协会，美国纽约，3-14。

[36] Heting Li和Mingyu Wu.2018.DwarfGC：用于云计算的NVM中的空间效率和崩溃一致的垃圾收集器。In *IEEE Symposium on Service-Oriented System Engineering (SOSE'18)*.192-197.

[37] 刘孟星，张明兴，陈康，钱学海，吴永伟，郑伟民，任静蕾。2017.DudeTM：用持久性内存的解耦构建持久性事务。*ACM SIGPLAN通告*52，4（2017），329-343。

[38] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang.2014.NVM Duet。统一的工作存储器和持久性存储架构。*ACM SIGARCH计算机架构新闻*42，1（2014），455-470。

[39] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris.2017.Persistent memcached。将遗留代码带入可字节寻址的持久性内存。在 *第9届USENIX存储和文件系统热门话题研讨会（HotStorage'17）上*。USENIX协会，加利福尼亚州圣克拉拉市，4。

[40] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum.2017.迈向实用的默认多核记录/重放。在*第22届 编程语言和操作系统的架构支持国际会议（ASPLOS'17）论文集*，（中国西安）。Association for Computing Machinery, New York, NY, USA, 693-708.

[41] Amirsaman Memaripour和Steven Swanson。2018.Breeze。用户级访问非易失性主存储器，用于 遗留软件。In *IEEE 36th International Conference on Computer Design (ICCD'18)*.IEEE，413-422。

[42] 米泽宇，李定基，杨子涵，王欣然，和陈海波。2019.Skybridge:用于微内核的快速和安全的进程间通信。在*第14届EuroSys会议（Eurosys'19）论文集*，（德国德累斯顿）。Association for Computing Machinery, New York, NY,

USA, 1-15.

[43] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton.2017.用WHISPER对持久性内存的使用进行分析。在*第22届 编程语言和操作系统的架构支持国际会议（ASPLOS'17）论文集*，（中国西安）。Association for Computing Machinery, New York, NY, USA, 135-148.

[44] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu.2018.Skyway:在分布式大数据系统中连接管理堆。在*第23届编程语言和操作系统的建筑 支持国际会议（ASPLOS'18）的论文集*。Association for Computing Machinery, New York, NY, USA, 56-69.

[45] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek.2010.Schism:分片容忍的实时垃圾收集。*在第31届ACM SIGPLAN编程语言会议论文集 Design and Implementation (PLDI'10)*, (Toronto, Ontario, Canada).Association for Computing Machinery, New York, NY, USA, 146-159.

[46] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda.2017.为非易失性主存储器编程是 hard。在*第八届亚太系统研讨会（APSys'17）论文集*，（印度孟买）。Association for Comput- ing Machinery, New York, NY, USA, Article 13, 8 pages.

[47] 任宇昕，Gabriel Parmer，Teo Georgiev，和Gedare Bloom。2016.CBufs:高效的全系统内存管理 ment和共享。在*ACM SIGPLAN内存管理国际研讨会（ISMM'16）论文集中*，（美国加州圣巴巴拉）。Association for Computing Machinery, New York, NY, USA, 68-77.

[48] 托马斯-舒尔，黄健，和何塞普-托雷利亚斯。2019.AutoPersist:基于 reachability的易于使用的Java NVM框架。In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementa- tion (PLDI'19)*, (Phoenix, AZ, USA)。Association for Computing Machinery, New York, NY, USA, 316-332.

[49] 严素欣。2019.持久性记忆。抽象、抽象、再抽象。*IEEE Micro* 39, 1 (2019), 65-66.

[50] George Taylor, Peter Davies, and Michael Farmwald.1990.TLB切片--一种低成本的高速地址转换 机制。In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, (Seattle, Washington, USA).Association for Computing Machinery, New York, NY, USA, 355-363.

[51] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra.2010.用PAPI-C收集性能数据。在*2009年高性能计算的工具*。Springer, 157-173.

[52] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell.2011.用于非易失性字节寻址存储器的一致和持久的 数据结构。In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, (San Jose, California).美国USENIX协会，美国，5-5。

[53] Haris Volos, Andres Jaan Tack, and Michael M. Swift.2011.Mnemosyne: Lightweight persistent memory.In *Proceed-第16届编程语言和操作系统架构支持国际会议（ASPLOS XVI）的报告*，（美国加州纽波特海滩）。计算机协会，纽约，美国，91-104。

[54] 王绍忠，余林亚，贺立安，黄元申，李仁权。2021.OpenCL 2.0程序的基于指针的发散分析.*ACM Transactions on Parallel Computing* 8, 4, Article 20 (2021), 23页。

[55] 王天聪，Sakthikumaran Sambasivam，Yan Solihin，和James Tuck。2017.硬件支持的持久性 对象地址转换。在第*50届IEEE/ACM国际微架构研讨会（MICRO'17）论文集中*，（马萨诸塞州剑桥）。计算机械协会，美国纽约，800-812。

[56] 王天聪，Sakthikumaran Sambasivam，和James Tuck。2018.硬件支持的per-sistent对象的权限检查，以提高性能和可编程性。在 *ACM/IEEE第45届计算机体系结构年度国际研讨会（ISCA'18）*上，（加利福尼亚州洛杉矶）。IEEE出版社，466-478。

[57] Michal Wegiel和Chandra Krintz。2008.XMem:类型安全、透明、用于跨运行时通信和协调的共享内存。*在 ACM SIGPLAN第29届编程语言设计与实现会议（PLDI'08）论文集*，（美国亚利桑那州图森市）。Association for Computing Machinery, New York, NY, USA, 327-338.

[58] Michal Wegiel和Chandra Krintz。2010.跨语言、类型安全和透明的对象共享，用于同地管理的运行时间。在 *ACM面向对象编程系统网络和应用国际会议（OOPSLA'10）论文集*，计算机协会，美国纽约，223-240。

[59] Paul R. Wilson.1992.单处理器垃圾收集技术。在 *内存管理国际研讨会上*(*Lecture Notes in Computer Science*), Vol. 637.1-42.

[60] Robert P. Wilson and Monica S. Lam.1995.C程序的高效上下文敏感指针分析。In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, Vol. 30, (La Jolla, California, USA)。计算机械协会，美国纽约，1。

[61] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin.2019.没有片外元数据的时序预取。In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*,（Columbus, OH, USA）, Association for Computing Machinery, New York, NY, USA, 996-1008。

[62] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin.2019.高效的元数据管理，用于不规则数据预取。在*ACM/IEEE第46届计算机架构年度国际研讨会（ISCA'19）*上，（亚利桑那州凤凰城）。Association for Computing Machinery, New York, NY, USA, 1-13.

[63] 吴明宇，陈海波，朱浩，臧炳玉，关海兵。2020.GCPersist:一个高效的GC辅助的懒惰持久性框架，用于NVM上有弹性的Java应用。In *Proceedings of the 16th ACM SIGPLAN/SIGOPS In- ternational Conference on Virtual Execution Environments（VEE'20）*, Association for Computing Machinery, New York, NY, USA, 1-14.

[64] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan.2018.Espresso：用非易失性内存酝酿Java的更多非易失性。在*第23届国际编程语言和操作系统的架构支持会议（ASPLOS'18）上，*。Association for Computing Machinery, New York, NY, USA, 70-83.

[65] 徐远超，严溶欣，和沈西鹏。2020.MERR：通过有效的内存暴露减少和随机化提高持久性内存对象的安全性。在*第25届国际编程语言和操作系统建筑支持会议*（瑞士洛桑）*论文集（ASPLOS '20）中*。计算机械协会，纽约，NY，987-1000。

[66] 徐远超，叶陈成，严溶欣，和沈锡鹏。2020.基于硬件的域虚拟化，用于持久性内存对象的内部 进程隔离。在*ACM/IEEE 第 47 届 计 算 机 体 系 结 构 年 度 国 际 研 讨 会 （ ISCA'20 ） 上*。 IEEE Press, 680-692. https://doi.org/10.1109/ISCA45697.2020.00062

[67] 叶晨曦，徐远超，沈西鹏，廖晓飞，金海，和闫索利欣。2021.基于硬件地址的，以键值存储为中心的加速。在*IEEE高性能计算机架构国际研讨会（HPCA'21）上*。IEEE出版社，736-748。

[68] 叶晨曦，徐远超，沈西鹏，廖晓飞，金海，和闫索利欣。2021.在 非易失性存储器上支持遗留库。一种用户透明的方法。在*ACM/IEEE第48届计算机体系结构年度国际研讨会（ISCA'21）上*。IEEE, 443-455.

[69] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez.2011.FREE-p:保护非易失性存储器，防止硬性和软性错误。在*IEEE第17届高性能计算机架构国际研讨会（HPCA'17）上*。IEEE出版社，466-477。

[70] 张璐和Steven Swanson。2019.Pangolin: A fault-tolerant persistent memory programming library.In *USENIX Annual Technical Conference (USENIX ATC'19)*.897-912.

[71] 张明哲，林景天，姚欣，和王祖利。2018.SIMPO:使用NVRAM进行可靠的大数据计算的可扩展内存中持久性对象框架。*ACM Transactions on Architecture and Code Optimization* 15, 1, Article 7 (March 2018), 28 pages.

2021年8月收到；2022年1月修订；2022年1月接受