基于 Redis 的混合内存池设计与实现

摘要

以 Redis 为代表的新型内存数据库在大数据处理领域以其高效性取得了广泛的关注，并且取得了巨大的商业价值。但是，由于其对 DRAM 内存开销的巨大，Redis 数据库会随着数据量的增长受到应用的限制。为了解决这个难题，一种有效的模型 —— 非易失性内存-DRAM 混合内存架构 —— 孕育而生。 这种架构结合了新型非易失性内存以及 DRAM 内存两者的优势与潜力，开始受到了广泛的关注。

先前的解决办法主要有两种，第一种方式是将新型非易失性内存作为系统的主存储器，DRAM 内存作为上层次的 Cache。第二种方式是将 Redis 全部移植到新型非易失性内存中。这两种办法的缺点在于没有充分发挥非易失性内存与 DRAM 的内存，使得 Redis 持久化性能并没有得到提升甚至降低了 Redis 数据库的综合性能。

这篇文章首先提出了多级数据存储架构以及四级数据流动模型，然后以此为理论依据提出了使用混合内存池的 Redis 问题的解决方案。我们使用工作在 Memory 模式的 Intel® Optane™ DC 非易失性内存作为系统的主存储器，使用工作在 App-Direct 模式的 Intel® Optane™ DC 非易失性内存作为 Redis 数据库的持久化存储器。

实验结果表明，我们的工作相比于使用 SSD 为持久化介质的数据库，在 RDB 持久化方面，I/O 性能提升 45.4%。 AOF 重写方面，I/O 性能提升 44.7%，开启 RDB&AOF 混合持久化之后，综合性能提升 11.05%。由于使用了新型非易失性内存来替代大量的 DRAM 内存，使得成本下降了 14.4%。

关键词: 混合内存 存储器架构 Redis 持久化

目录

图片目录

1 引言

新型内存数据库在大数据处理领域以其高效性取得了广泛的关注， 并且取得了巨大的商业价值。 由 Redis 社区有效维护的 Redis 就是一种典型的内存型数据库 [1]， 它支持字符串、哈希、列表、集合、有序集合以及其他的数据模型。这种数据库在互联网以及云计算领域得到了广泛的应用。但是，由于其对 DRAM 内存开销的巨大，Redis 数据库会随着数据量的增长受到应用的限制。

为了解决这个难题， 正在兴起的持久化内存（NVM）技术已经展开了积极而又活跃的研究。比如 Phase Change Memory (PCM) [9]，STT-RAM [10]， recetrack memory [11] 以及 3D Xpoint [12]。NVM 内存可以有效的降低内存的成本开销。 一方面，与传统的 DRAM 内存相比， NVM 在持久性、 存储能力方面有着显著优势。 另一方面，与普通的固态硬盘（SSD）相比， NVM 有着可字节寻址性、更快速的访问速度以及更低的访问延迟。但是，受到 NVM 在读写速度有差异以及有限次写寿命的影响，NVM 内存目前为止还不能完全取代 DRAM 在计算机系统中主存储器的地位[13]。因此，一种有效的模型 —— NVM-DRAM 混合内存架构 —— 孕育而生。这种架构结合了 NVM 内存以及 DRAM 内存两者的优势与潜力，开始受到了广泛的关注。

目前为止，这种混合架构主要有两种应用方式。 第一种方式就是将 NVM 内存作为主存储器， 并且将 DRAM 内存作为上层的 Cache 加速器。 这种方案主要是看中了 NVM 在存储能力高、相同容量价格便宜的潜力，因此也成为了最为主流的应用方式。第二种方式是将 NVM 作为一种块设备来存储数据，这种方案主要是看中了 NVM 的可持久性以及高速访问性能的特点。这种方案在一些缓存类应用有着相应的研究。比如，Xianzhang Chen 等人使用 NVM 作为 KV 数据库的存储层并且提出了一种内存 KV 数据库的混合架构 [7]。那篇文章在 NVM 中的特定位置引入了数据库元数据来加速架构的启动和关闭速度。但是，这种方案的弊端在于元数据的引入限制了 NVM 在其他领域的应用。

幸运的是，随着 Intel® Optane™ DC 持久化内存 [14] 以及持久化内存编程组件（Persistent Memory Development Kit）[2] 的发展，现在变得越来越容易去使用 NVM 来作为应用存储层的一部分或者使用 NVM 作为计算机系统主存储器。与此同时，一些内存管理工具，比如 ipmctl, ndctl 的发布使得 NVM 可以像计算机系统中的其他设备一样方便管理。另外，由于持久化内存编程组件使用了更高阶的应用程序接口而不是底层的 CPU 指令，开发者会非常轻松的在 NVM 内存上面进行编程。

这篇文章的主要目的就是要在不损失 Redis 整体性能的情况之下降低 DRAM 内存的成本花费并且提高 Redis 数据库的持久化速度。与此同时，我们将尝试提出一种多层次数据存储解决方案，这种解决方案也可能应用到其他问题中来提升整体数据处理性能。这篇文章的主要工作可以归结如下：

（1） 我们首先会基于 NVSL 实验室 [3] 的研究来概述 Intel® Optane™ DC 持久化内存的一些基本性能指标，并且尝试给出一些使用的建议。

（2） 我们提出了全新的多级数据存储架构解决方案，这种方案既可以解决这篇文章的主要目标——提升 Redis 持久化性能，也可以应用到其他领域之中去。

（3） 我们基于 Redis-5.0.8 版本进行修改，并且测试改进之后的一些基本性能指标，并且给出后续工作的一些建议

接下来的文章将会按照如下方式进行组织。在第二部分，文章将会继续讲解 Intel Optane DC 持久化内存、持久化内存编程组件、Redis 持久化的原理、混合内存架构研究现状、混合内存架构综述以及这篇文章的灵感。在第三部分， 文章将会基于 NVSL 实验室 [3]的研究来概述 Intel Optane DC 持久化内存基本性能指标以及给出使用意见。在第四部分，本文提出多层次的数据存储解决方案以及基于持久化内存的 Redis 方案的细节实施。全文的实验结果和数据处理将会放在第五部分讲述。第六部分会总结全文。

2 背景与灵感

2.1 Intel® Optane™ DC 持久化内存

Intel® Optane™ DC 持久化内存[14]是一种全新的非易失性 DIMM 内存。与传统的 DRAM DIMM 内存一样，这种内存可以接在计算机主板的内存总线上。但是，Intel® Optane™ DC 持久化内存与 DRAM 内存相比，可以支持持久化操作。同时，这种新型内存的单位存储容量也很高，单 DIMM 条可以达到 128G。

在传统的计算机系统之中，使用 DRAM 内存直接访问持久化介质（通常是硬盘）。即使是使用全新的高速固态硬盘，响应延时还是相差几个数量级。另外，传统的 DRAM 内存单位存储成本与硬盘相比也相差几个数量级，同时数据在断电之后无法保留——这也是限制了新型内存数据库发展的主要原因。但是，Intel® Optane™ DC 持久化内存开始成为搭建上述两级存储器之间的桥梁：在 DRAM 内存与块存储设备之间加入了大容量的持久化内存层。这样的做法既可以解决 DRAM 断电之后数据丢失的难题，同时以较低的成本提供了十分出色的性能[3]。

Intel® Optane™ DC 持久化内存有两种工作模式：第一种工作模式是 Memory 模式。在这种模式下面，持久化内存将会做为计算机的主存储器来使用，同时将原有的 DRAM 内存上升以及作为 CPU Cache。这种模式的优势在于使用十分简单，通过简单的配置就可以在减少 DRAM 的情况之下得到原来系统的出色性能。第二种工作模式是 App-Direct 模式。在这种模式中工作的持久化内存将会作为 DRAM 的下级存储器使用。目前主要有三种方案来访问在 App-Direct 模式下工作的持久化内存，第一种是标准原生设备接口来访问，这种方式的缺点开发难度大且不易进行优化；第二种是通过持久化内存可感知文件系统（persistent memory aware file system）来访问，这种访问方式最为简单，通过两级文件系统的映射关系，可以把持久化内存当做普通的块设备来处理。但是数据将会在多级文件系统之间流动，这将会大大降低持久化内存的读写效率。第三种方案是通过内存映射文件（memory mapped file）来访问，这种方案通过 MMU 映射直接访问持久化内存，绕开了持久化内存驱动程序（NVDIMM driver）实现起来高效便捷[15]。

2.2 持久化内存编程组件

持久化内存编程组件 （Persistent Memory Development Kit）[2]是由存储与网络工业协会 （Storage Network Industry Association）主导开发的通用持久化内存编程工具。持久化内存编程组件包含了一系列方便 NVM 的应用开发的函数库和工具。目前支持在 Window 以及 Linux 操作系统上进行开发、运行以及调试，但是持久化内存编程组件一个明显的缺点就是提供的函数接口还不够完善，对于一些特定的应用来讲仍然有改进的空间。

持久化内存编程组件建立在 NVM 内存直接访问机制（Direct Access）基础之上， 也就是上一部分所讲的第三种 NVM 访问方案，即通过使用操作系统底层的内存映射文件（memory mapped file）来访问。因此，使用持久化内存编程组件进行编程之前，需要了解内存映射文件的编程模型。

持久化内存编程组件目前包含的函数库如下：

libpmem：提供最底层持久化内存支持方案

libpmemobj：提供事务对象存储，提供内存分配、事务机制以及持久化内存编程的通用组件

libpmemblk：提供定长块存储，其中每一个数据块支持原子性更新操作

libpmemlog：提供日志存储方案

libpmempool: 提供内存池管理以及诊断方案

libvmemcache：是一个可嵌入和轻质的内存缓存解决方案，它旨在通过有效且可扩展的内存映射存储来充分应用大容量的内存

在这一系列的函数库之中，主要的持久化工具有 libpmem、libpmemobj、libpmemlog 以及 libpmemblk，其中这四个函数库的关系为：libpmem 为最底层的存储库，剩下的三个函数库是在 libpmem 库之上进行开发的，并且屏蔽了如多种类型的刷新操作，开发难度相对较低，但是相应的是程序运行效率会进一步的下降。另外值得一提的是，只有 libpmem 函数库支持 Intel Optane™ DC 持久化内存以字符设备的方案进行访问，即 devdax 模式访问。devdax 访问会比其他模式更加高效[30]。

2.3 Redis 持久化

Redis 数据库是典型的内存数据库，与传统的关系数据库不同之处在于 Redis 数据结构全部为非持久性内存数据结构。这样的特点会导致 Redis 数据库退出之后所有的数据并不能够保存下来，因此 Redis 需要设计一套持久化机制来让库中所有的数据保存在存储新介质之中。

Redis 提供了不同级别的持久化方式：RDB 持久化以及 AOF 持久化[16]。

RDB 持久化方式是在通过数据库进行快照的方式进行存储。用户既可以通过指定时间来自动执行 RDB 持久化，也可以通过 SAVE 或者 BGSAVE 命令来手动开启 RDB 持久化。RDB 持久化的大致原理是扫描所有数据库的每一个键值对，并且将所有的非线性数据结构转化为线性结构存储在 RDB 文件之中。并且对于较短的值存储在压缩列表中，对于较长的值先执行 LZB 压缩算法[26]，通过压缩之后进行存储。RDB 持久化形成了一个非常紧凑的全量存储文件，所有的数据已经通过了高度的压缩，因此在网络的传输之中将会大大的节省了带宽，很方便通过加密的方式在数据中心之间传输，非常适用于灾难恢复。但是 RDB 持久化执行期间对于系统资源的消耗巨大，而且不能通过设置检查点的方案实现增量转储。

AOF 持久化方式是通过数据库生成追加写操作的方式来实现数据库持久化。AOF 持久化的执行由系统自动完成。数据库的每一条修改操作都会追加在 AOF 缓冲区之中，用户通过设置不同的刷新策略：无 fsync、每秒 fsync、每次 fsync 来控制 AOF 缓冲区的刷新频率[17]。AOF 持久化相比 RDB 持久化提供了更高的可靠性与稳定性，但是开启 AOF 持久化将会降低 Redis 系统的整体性能。AOF 文件格式简单，但是缺点是同样数据量的 RDB 文件与 AOF 文件相比，后者显得庞大很多，因此 AOF 持久化支持 BGREWRITEAOF 操作，通过该操作可以实现 AOF 文件的瘦身。

在 Redis-5.0.8 版本中，支持 RDB-AOF 混合持久化。开启了混合持久化，将会充分发挥两种持久化方案的优势，规避两者的缺点，因此这种方案在持久化的应用中有着更为广泛的应用。

2.4 混合内存架构研究现状分析

混合内存架构问题是当前研究的重点问题。DRAM 内存访存速度快，读写均匀而且基本不会造成 DRAM 存储颗粒的损耗，缺点在于其高昂的物料成本以及能耗成本。因此在计算机系统中，DRAM 的容量不会很大。与此同时，NVM 内存具有较低的物料成本，因此可以在计算机系统中有数倍容量空间。但相应的问题在于访存速度相比 DRAM 有所下降，而且读写不均衡，写操作会对其存储颗粒造成损耗。因此，结合两者的优势，避开二者的劣势来使用混合内存具有极高的现实意义。

Zhiyong Zhang 等人研究了混合存储架构在减少操作系统日志开销方面的应用。他们提出了 JAPM（Journaling-Aware Page Management) 规则，JAPM 规则将不频繁的数据放在 NVM 内存中来减少日志系统的负担，将高频率的数据存放在 DRAM 中以提升写性能以及混合内存的寿命 [4]。 Duo Liu 等人研究了在以 PCM 为 NVM 内存的混合内存的嵌入式系统中的磨损均衡算法，他们提出了基于特定应用程序的磨损均衡技术——叫做 Curling-PCM 技术。这种技术的基本原理是将 PCM 中的热区域周期性的移动，每次只移动一部分。通过这种办法可以有效的缓解 PCM 芯片的写带宽资源紧张的局面 [5]。Hyunchul Seok 等人提出了基于混合内存的页面 Cache 算法。这种算法可以预测页表命中区域和将要替换出去的页面，以此来通过绑定写的方式访问 DRAM。通过这种算法，可以将写入次数降低最大 52.9%，节约 19.9% 的电能 [6]。Xianzhang Chen 等人为内存 KV 数据库设计了一种混合内存架构——UDORN (Unified Database on Raw NVM)。这种架构在 NVM 区域设计元数据来对数据库进行定位 [7]。Eunji Lee 等人设计了一种缓存策略 UBJ（Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory），这种策略规定了日志的多种状态并为此构造了有限自动机，并且将不常被引用的日志替换进 NVM 中，这种策略与传统的 Linux ext4 [18]文件系统相比，I/O 性能提升了 76% [8]。

2.5 灵感

前面我们提到 Redis 对于主存储器的开销巨大。因此通过 DRAM-NVM 混合架构来降低商业成本的开销。目前商业中的应用方案是开启 Intel® Optane™ DC 非易失性内存的 Memory 模式[19]，通过减少实际系统中 DRAM 内存的方式来降低运营成本。这种解决方案的优势在于实现起来非常简单，但是并没有充分发挥新型非易失性内存的可持久性特点，并且在一些使用场景中造成持久化操作效率的降低。另外，学术界的一种研究解决方案是通过在新型非易失性内存上重构 Redis 数据库，将所有的易失性数据结构转化为非易失性数据结构，并且使用元数据来定位 Redis 数据库在内存中的真正位置。这种方案的确可以大大降低 Redis 数据库运营成本，但是重构数据库耗时时间长、NVM 读写不均衡导致严重拖慢

Redis 的运行效率等缺点使得这种解决方式得不偿失。

进一步的，我们调研了 Redis 数据库的实际应用场景。阿里巴巴集团曾经面向不同类型企业给出了基于 Redis 的云存储解决方案 [20]。在视频直播应用领域，使用基于 Redis 数据库的动态扩容升级技术。比如一个主播开启了直播之后，观众的数量并不是一成不变的，而是在前期有一个增长过程，这种增长速度有时候是爆炸性的增长。因此在每达到一个观看数量级之后增加云服务数量来提高直播承载能力。每扩容一个节点，会开启一个从 Redis 服务器，并且通过接收主 Redis 服务器持久化文件来恢复备份。这种方案的缺陷在于每一次新增节点都会执行一次 RDB 持久化算法，降低了 Redis 数据库整体的吞吐率。另外 I/O 开销也是一个不可忽视的影响因素。但是如果在 NVM 上构造一个内存文件系统，并且在文件系统上存储最新版本的 RDB 文件，这样在动态扩容过程中，最快传输速度可以达到 DRAM 内存的写速度。这将会是一个巨大的提升。另一方面，网络游戏服务领域中，Redis 数据库的作用也非常重要。当一款网络游戏非常火爆的时候，网络游戏提供商通常选择划定多个游戏服务区来满足用户需求，但是不同服务区之间的数据时无法共通的。这样做的原因在于多个游戏服务区之间进行同行的代价非常巨大。跨服通信需要通过经过备份存储层来写入/读取数据，一次通信需要将一份数据至少写三遍。但是如果建立了基于新型持久化内存的 Redis 数据持久化服务器，可以实现一份数据写一遍，大大提高了跨服通信的效率。

在上述调研之后，我们基本确定了之后多级数据存储模型。

3 Intel® Optane™ DC 持久化内存基本性能概述

位于加利福尼亚大学圣迭戈分校的非易失性系统实验室（Non-Volatile System Laboratory） 在 2019 年 8 月份完成了对 Intel® Optane™ DC 持久化内存（以下简称 Optane DC 持久化内存）的测试工作 [3]。这一部分，我们将会基于 NVSL 的研究成果进行概述。

Intel® OptaneTM DC 持久化内存和 DRAM 相比较，最重要的区别在于前者有着更长的延迟以及更低的带宽并且存取性能是不对称的。

在延迟方面，Optane DC 持久化内存真实的随机读延迟为 305ns，相比之下，DRAM 的访问延迟为 81ns。但是 Optane DC 持久化内存的顺序读延迟为 169ns。这表明 Optane DC 持久化内存内部存在一定的缓冲区域。

在带宽方面，Optane DC 持久化内存要低于 DRAM，尤其是写性能差距更加明显。

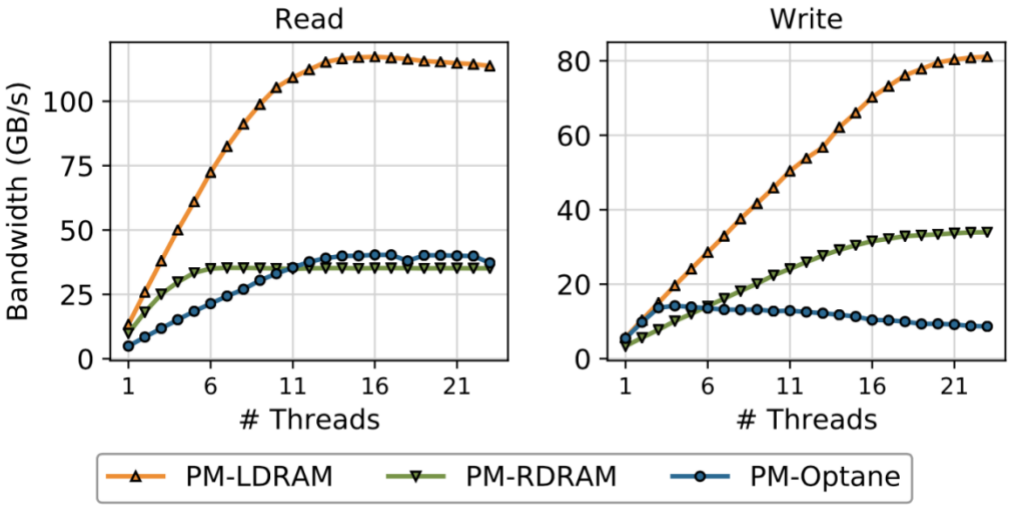

图 3-1　内存顺序访问带宽对比（图来源［3］）

图 3-1 描绘了分别从 1 到 23 个线程访问 6 个 DRAM 或者 Optane DC 持久化内存模块的顺序读写带宽状况。其中 PM-LDRAM 表示的是本地的 DRAM 访问，PM-RDRAM 表示的是 NUMA 结点中临近的 DRAM 访问情况。Optane DC 持久化内存的顺序读取带宽的峰值为 39.4GB/s（左图所示），顺序写入带宽在 4 个线程的时候达到最大为 6.6GB/s（右图所示）。对于单线程的方式访问 Optane DC 持久化内存，最大顺序读取速度为 6.6GB/s，　最大顺序写入带宽为 2.3GB/s。

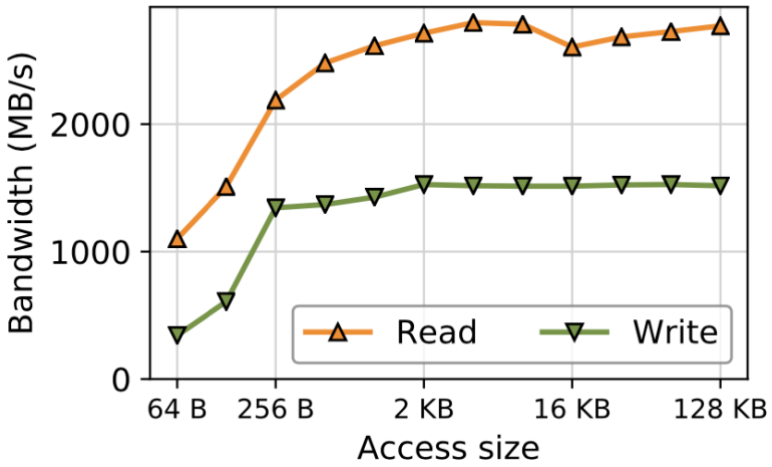

图 3-2　Optane DC 非易失性内存随机读写性能（来源［3］）

图 3-2 描绘的是以单线程的方式以随机读写不同大小块的方式访问单模块的 Optane DC 持久化内存的带宽情况。图中可以看到，随着存取块在 64B 到 256B 之间的访问带宽会快速的增长，并且随机写带宽在 1.5GB/s 的时候到达峰值，随机读带宽在 2.8GB/s 到达峰值。

基于上述的基本性能状况，我们给出如下的使用建议：

1. 使用 15 个线程左右的顺序读会使得带宽最大，与此同时，顺序写最好维持在 4 个线程

2. 随机读取的内容大小在 10KB 左右最佳，随机写入的内容可以先放入缓冲区中，等到 2KB 左右统一刷入 NVM 中。

4 方法论

这一部分是整篇文章的核心部分。在这一部分里面，文章首先会讲述多级数据存储架构的提出背景，然后依据背景与挑战来提出全新的多级数据存储架构，最后在多级数据存储架构思想的指导下，实现 Redis 到 NVM 的迁移已经实现的细节问题。

4.1 传统数据存储架构现状分析

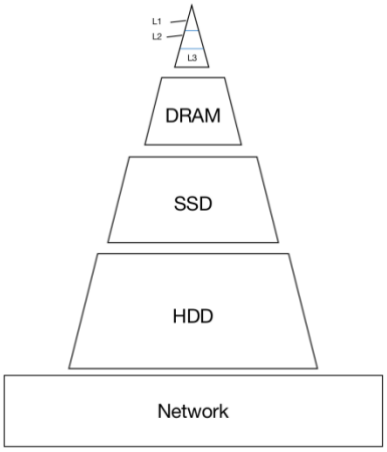

图 4-1 金字塔形的数据存储架构

常用的数据存储架构是金字塔形的数据存储架构，最上面一级是 CPU L1 Cache， 最下面一级是网络以及云存储硬盘。这样设计的存储依据是按照存储器访问性能高低与单位存储的价格成本来制定的。这种结构有着一个明显的分界线，那就是 DRAM 内存与硬盘之间的分界线。这条分界线并非是人为故意加上去的，而是存储领域发展多年的结果。

DRAM 内存与硬盘的分界线一方面是易失性存储介质与非易失性存储介质的分界线，但另一方面显示出了在数据存储金字塔中分界线两侧存储器的巨大差异。DRAM 内存的相应速度一般在 50ns 级别 [23]，但是即使最高效的硬盘的响应时间也在 ms 级别。中间的差距在$10^5—10^6$ 数量级。这样的差距将会直接拖慢整体系统的性能。另外，持久化与易失性的界限也非常明显，这就意味着 DRAM 中驻留的重要数据必须存储的硬盘中作为备份，这样的做法也会带来存储介质的额外开销。

除了数据金字塔分界线两侧存储器的差异越来越大，计算机系统整体数据处理能力的大跃升也对数据存储架构带来了巨大的挑战。CPU 单核心的处理能力已经开始进入瓶颈期，CPU 核心频率越高，能耗将会以指数级别增长，同时制程工艺也开始触及天花板。因此，多核处理器甚至众核处理器时代已经来临[21]。比如 AMD 公司最新发布的第三代线程撕裂者 CPU 核心数已经达到了 64 颗 [24]，这样的数据处理能力使得 DRAM 也有些捉襟见肘。与此同时，深度学习技术 [22] 在计算机中的广泛应用，图形处理器（GPU）也开始密切融入到计算机系统中。

比如 NVIDIA 公司发布的 GeForce RTX 2080Ti 拥有 4352 个 CUDA 核心，浮点运算能力达到了 13.4 Tflops [25]。因此，在拥有 GPU 的计算机系统中，GPU 的存储结构与金字塔形数据存储结构显得格格不入，一方面，GPU 显存开始制定自己的标准体系，但另一方面，GPU 数据的装载与返回还是离不开 DRAM 与 GDRAM 之间通过系统总线通信。显然这种设计方案与计算机存储一体化的设计思路分道扬镳。在这样的背景下，金字塔形存储架构需要得到一定的调整与改进。

4.2 多级数据存储架构与四级数据流动模型

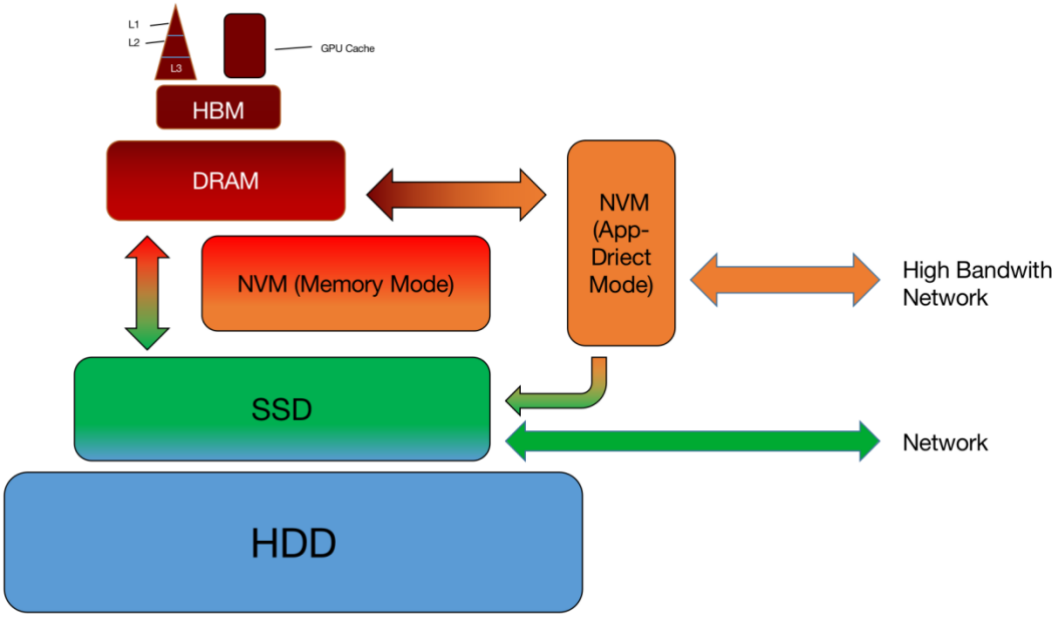

图 4-2 多级数据存储架构及其数据流动示意图

如图 4-2 所示，多级数据存储架构将重新整合 GPU 的数据存储架构，并且通过 NVM 来缩小 DRAM 与硬盘之间访问时延的差距，并且通过 NVM 的工作模式来模糊易失性与非易失性存储器之间的界限。

首先，为了解决多核心数据处理带宽首先的问题，我们使用 HBM （High Bandwidth Memory) [27] 作为 CPU 与 GPU 数据处理的统一接口，HBM 内存相比 DDR 内存拥有极高的带宽速度，在目前主流的 GPU 中有一定的应用。但最大的问题是在当前的计算机系统中，HBM 内存只是用来存储 GPU 的关键数据，因此在 GPU 闲置的时候，HBM 的存储区域不能够加以利用。另外，HBM 内存数据的装载目前是通过计算机总线与 DRAM 通信，这样的做法只会加剧系统总线资源的紧张场面。但是，在我们设计的方案中，通过将 HBM 与 DRAM 一样接入 DIMM 插槽中，直接将 HBM 交付由操作系统管理，实现接近 L3 Cache 的数据通信速度。这样做的好处在于大幅度释放了系统总线的压力，同时提高计算机系统数据的整体利用效率。

接下来多级数据存储架构相比金字塔形数据存储架构最大的改进之处在于对分界线两侧进行模糊化处理。利用 NVM 的重要特点，我们使用 NVM 的工作模式进行平滑过渡。首先使用工作在 Memory 模式下面的 NVM 作为整个系统的主存储器，这样的处理将会大大的降低传统架构中 DRAM 内存做主存储器所带

来的巨大成本。另外，使用工作在 App-Driect 模式下面的 NVM 来实现数据持久化功能的最高速存储器。这样的模糊化处理，将数据持久化的响应时间由原来的毫秒数量级降至微秒数量级。因此，操作系统进行重要数据持久化的时候，直接调用 NVM，可以去掉原有的进程阻塞操作。

然后，我们设计多级数据存储架构时，将计算机系统的所有数据全部进行了分类。参考冷——热数据模型以及受到了火山岩浆流动现象的启发，我们在多级数据存储架构中也划分了数据的流动模式。在 DRAM 层以上的存储系统中，数据的处理速度一般在 10Gbps 以上，这样的数据称之为热数据。当热数据开始被操作系统替换出去的时候，数据的流动也相应的开始了。当数据进入到 NVM 中，数据再次被调用的频率也开始下降。这时候的数据称之为温数据，温数据属于中间过渡的数据，这类数据有可能会重新变为热数据，也可能不在被系统引用。因此，温数据的处理也需要进行相关的调整。在这里，我们使用工作在 App-direct 模式下面的 NVM 和磁盘同时做温数据冷却化处理。当应用程序认定当前的温数据今后重新成为热数据的概率极低并且需要进行持久化，可以直接调用文件系统接口，将数据直接持久化到固态硬盘中。但是，当前温数据之后的转化状态无法确定且需要持久化的时候，将会以极快的速度将数据直接存入 NVM 中。之后，数据的流向也会按照数据使用频率成为热数据或者冷数据。

进一步的，我们对冷数据也进行了扩展。计算机系统中，部分的冷数据可能在特定的时间重新装载到内存中。但是也存在一些冷数据可能成为了备份数据，在相当长的一段时间内不再会被使用，这样的数据成为冰数据。这种数据可以流向更加廉价的持久性存储器中，比如机械硬盘或者磁带。

最后的改进是将金字塔形数据存储架构中网络 I/O 的地位进行了调整。随着网络技术的发展，许多高速网络协议已经变得成熟起来，甚至一些万兆网络可以达到与 DRAM 相同的处理速度，因此，计算机系统中的热数据可以直接在网络中进行流动。将高速网络直接接入 NVM 中，实现热数据跨系统的通信。同时，兼顾传统广域网数据通信，我们在 SSD 层也并入了网络传输。

4.3 实施

按照多级数据存储架构的思想下，这一部分我们将会对 Redis 的最新版本 Redis-5.0.8 进行迁移开发。

首先要解决的是 Redis 数据的划分问题。之前提到，Redis 在 NVM 上迁移有两种解决方案。第一种是在商业模式中，开启 Intel® Optane™ DC 持久化内存的 Memory 工作模式并将 Redis 数据库运行在此模式下。这样的解决方案在一定情况下可能会造成持久化程序效率下降，同时并没有发挥出 NVM 的持久化优势。第二种解决方案将 Redis 全部迁移到 NVM 中，通过使用元数据的方案来定位程序在 NVM 中的地位，这样的解决办法虽然降低来 DRAM 的开销，但是随之而来的就是 Redis 整体性能的下降。因此，上述两种解决方案出现的问题在于没有对数据进行分类。

按照 Redis 数据库的产品定位，Redis 数据库运行时装载的数据时热数据，被再次访问的概率极高。但是直接使用 DRAM 装载热数据的成本极高，因此我们使用 NVM-DRAM 混合内存架构来装载 Redis 数据库产生的热数据。在 Redis 开启持久化之后，产生的持久化主要有两种用途。第一种用途是用来形成数据库备

份文件，在数据库开机或者数据库故障恢复的时候才有可能使用到。第二种用途是用来尽心 Redis 数据库动态扩容操作，这样的持久化文件会重新转化为热数据。因此，Redis 数据库经过持久化的文件属于典型的温数据，这种类型的数据按照多级数据存储架构，应该放置在工作在 App-Direct 模式下面的 NVM 中。

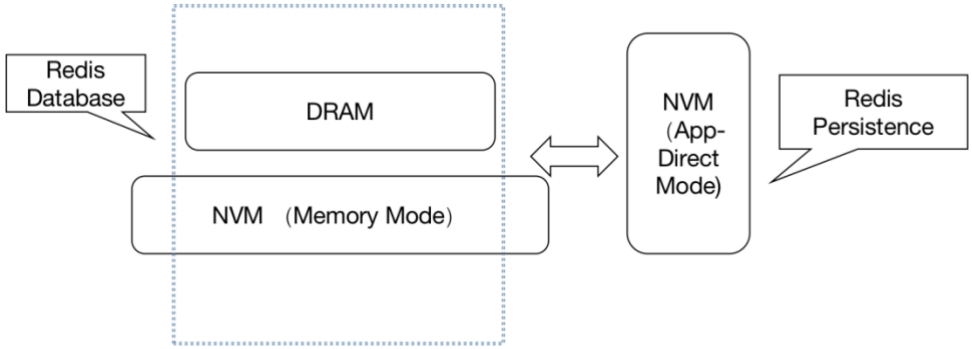

图 4-3 Redis 迁移解决方案

因此，我们提出的 Redis 迁移解决方案为：一、将 Redis 数据库的程序主体迁移至在处于 Memory 工作模式的 NVM 中。二、将 Redis 数据库的所有持久化操作迁移至处于 App-Direct 模式下面的 NVM 中。

借助于 Intel® Optane™ DC NVM 的 Mix 模式，我们可以同时开启 Memory 模式以及 App-Direct 模式。关于如何开启并使用 Memory 模式的细节这里将不再赘述，这部分主要关注基于 App-Direct 模式下面 Redis 数据库的持久化迁移工作。

首先解决的第一个问题是如何访问 NVM。在上文分析的基础之上，主要有三种可选择方案。第一种方案是通过 DAX 文件系统来访问 NVM，这种方案的优点是使用简单，通过一定的配置之后，直接将 NVM 当做普通块设备来访问，但是效率的损失也是最显著的。第二种方案是通过持久化内存编程组件的高层次 API 来访问，这种访问方式需要对 Redis 持久化部分代码进行修改，使用相应的编程模型设计。但是这种访问方式中，持久化内存编程组件的代价高，因此对效率也有一定的影响。第三种访问方式同样通过使用持久化内存编程组件来完成，但是这里使用低层次的函数接口。这样的访问方式会以接近原生 NVM 的访问速度来完成读写操作，效率非常高。但是相应的开发工作较多，而且通常把 NVM 当做字符设备来处理，增加了开发难度。

为了提高 Redis 持久化的效率，我们使用了第三种方案来进行开发。开发主要有三大模块需要进行迁移：RDB 持久化迁移、AOF 持久化迁移以及 BGREWRITEAOF 模块迁移。为了完成迁移，我们大致浏览了 Redis 数据库的源码，并且着重研究了上述三个模块相应的源程序。

RDB 持久化是一种全量转储的持久化机制，在 Redis 中开启了 RDB 持久化之后，每隔一段时间系统将会调用相关的接口来备份整个数据库的数据。相关实现位于 rdb.c 的文件中，大致的操作如下。当系统出发 RDB 持久化条件或者执行 RDB 持久化相关命令之后，首先会调用 rdbSave 函数，这个函数主要完成 RDB 子程序的初始化工作。然后调用 RDB 持久化的核心函数 rdbSaveRio。这个函数将会遍历整个数据库，将所有的底层实现数据结构全部转化为线性结构，并且调

用 LZF 压缩算法对生成的线性结构进行压缩，最后统一调用 Redis 的 RIO 层来进行文件的相关操作。因此我们的改进主要在 RIO 层添加了有关 NVM 调用的接口。并且在 rdbSave 函数的时候连接到 NVM 中，返回一个指向该区域的指针，之后的操作将会转化为对指针的读写问题。这里处理的一个关键是什么时候将相关的数据刷入 NVM 中，我们需要权衡安全性与效率之间的关系。在 libpmem 函数库中，提供了一个函数 pmem_memcpy_nodrain( ) 可以实现我们所要求的目的，最后一次持久化过程中使用 pmem_drain( )函数确保将所有的内容成功持久化到 NVM 中。这样的做法可以有效避免频繁的随机写带来的性能下降问题，并且随着数据库规模的扩大，整体性能提升也会愈加明显。

AOF 持久化是一种动态的增量存储。相比于 RDB 持久化过程，AOF 持久化可以是数据库更加的安全有效，但是随之而来会使得数据库的性能有所下降。因此，如果将 AOF 文件存储在 NVM 中，会使得文件 I/O 操作的代价大幅度降低。AOF 持久化机制与 RDB 持久化有着很大的不同，AOF 文件相当于关系数据库的 Redo 日志，存储着用户的每一条操作。用户每当输入一条修改命令之后，这条命令会以追加的方式存入 AOF 缓冲区中，并且随着 serverCron 函数的循环调用，按照用户的设置方式在特定的时间内将 AOF 缓冲区中的数据写入到磁盘中。因此，我们的做法是直接将 AOF 缓冲区放入 NVM 的内存映射区域中，每一次命令的追加将会在内存映射区域内完成，并且按照用户的策略在特定时间内调用 pmem_persist( ) 函数将 AOF 缓存区刷入 NVM 中。这样的策略使得原有的两次内存写操作以及一次 I/O 操作变为一次内存写操作和一次 NVM 写操作。这种范式对 AOF 策略将会非常的友好。但是随着数据库工作时间的增长，AOF 持久化机制的缺陷也越来越严重，那就是 AOF 文件将会变得非常臃肿。Redis 数据库为此提供了一种机制——BGREWRITEAOF 机制 [31]，即通过重写 AOF 文件的方式来完成 AOF 文件的“瘦身”。这种机制的底层实现与 RDB 持久化有着相似的方式，通过遍历整个数据库的所有键值对，并且以逆过程形成所有键值对的添加操作。这样产生的新文件就是重写的 AOF 文件。为此，我们的做法参考了 RDB 持久化与 AOF 持久化的迁移工作，借助之前在 RIO 层实现的 NVM 的访问接口，我们通过相同的方式可以实现 BGREWRITEAOF 机制在 NVM 上的迁移工作，实现细节与 RDB 持久化相近。但是值得一提的是，AOF 重写的进程与 Redis 主进程是两个进程的关系，所以相关的变量传递需要借助进程之间的通信。

借助于持久化内存编程组件，我们成功的完成了 Redis 数据库持久化的迁移工作，Redis 的数据流由原来的热数据——冷数据模型相应的变为了热数据——温数据——冷数据——冰数据的四级流动模型。另外，当真实系统中即使不存在 NVM，我们经过迁移的数据库也可以正常的运行。接下来的部分，将会通过实验来对我们的工作进行检验。

5 实验

5.1 实验设置

接下来的所有实验全部是基于 linux-4.18 内核的环境平台搭建。实验平台的 DRAM 容量为 384GB， Intel® Optane™ DC 持久化内存大小为 1.5TB。我们的额实验将会基于上述的改进解决方案进行测试。测试有 RDB 持久化中 I/O 操作的代价测试、AOF Rewrite 操作中 I/O 代价测试、开启 RDB&AOF 混合持久化的综合性能测试、数据库规模压力测试共四大部分测试，测试的第三方工具为 Redis 官方提供的 Redis Benchmark 工具 [28]。另外，我们也有自己开发的数据生成器来测试一些特定的功能。测试环境中，我们为了控制好实验平台环境，确保没有其他大型程序的干扰。

5.2 RDB 持久化性能测试

RDB 持久化是 Redis 数据库频繁需要使用 I/O 操作的地方，因此 RDB 持久化中一个不可忽视的开销就是 I/O 开销。因此，这一小节我们会集中对 RDB 持久化 I/O 进行对比测试。实验中的参照对象有使用挂载着 XFS 文件系统的高性能固态硬盘为持久化介质的 Redis 数据库。另外，我们也将以文件系统方式访问 NVM 为持久化介质的 Redis 数据库作为实验的参照对象。其中，我们使用的文件系统有两种，一种是 EXT4 文件系统，另一种是 XFS 文件系统[29]。其中 EXT4 文件系统支持直接访问（DAX）模式，但是在 Linux 4.18 中，XFS 还不支持直接访问模式。

数据库中数据的生成使用我们自己开发的数据生成器。数据生成器的工作原理是产生大小为 512MB 的超大键值对。其中为了防止 Redis 数据库中数据压缩算法对本次实验的影响，我们构造键值对时采用随机算法并以均匀分布在 26 个含大小写英文字母以及数字中生成字符，这样产生的字节对大小就是 RDB 文件中相应内容的大小。另外，为了统计 I/O 操作，我们使用了控制变量法的原则。对于每一组实验，我们首先关闭 RDB 持久化中所有有关 I/O 操作并执行算法，测出算法执行时间。这一部分时间包括了 CPU、网络等开销在内的所有系统开销。接下来开启 I/O 操作编译项目，重复上述测试得到第二个算法执行时间。然后两个算法执行时间相减可以得出 I/O 的代价。

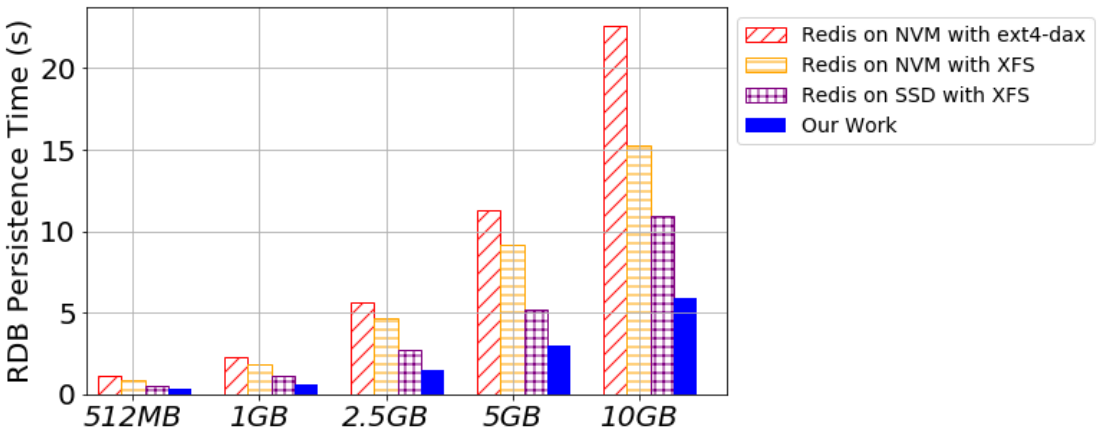

图 5-1　RDB 持久化性能对比图

如图 5-1 所示，我们统计了 RDB 持久化测试中 I/O 时间代价。其中持久化文件的规模分别为 512MB、1GB、2.5GB、5GB 以及 10GB。纵轴的单位为秒，其中柱形图的高度越低代表性能越好。首先最为明显可以看到的是，我们的工作相比于其他几种较为主流的 Redis 持久化方案来讲，取得了十分有效的提升。即使是搭载了以高效率著称的 XFS 文件系统的 SSD，也只有我们方案性能的一半左右。

但是，这张图也可以显现出 NVM 的一些不足之处。为了提升 Intel® Optane™ DC 持久化内存的兼容性，开发人员在 NVM 的驱动中加入了持久化内存可感知文件系统，使得 NVM 可以被操作系统识别为块设备来访问。但是这种访问方式在单线程程序中根本没法发挥优势，性能反而不如固态硬盘。因此，要想使用好 NVM，还需要使用原生的方式或者是使用持久化内存编程组件来发挥其最大的优势。

在我们的测试中发现，我们的解决方案中 RDB 持久化的 I/O 速度达到了 1.7GB/s，这个速度其实已经接近了单线程访问 NVM 的性能上限。这样的速度已经与 DRAM 内存单线程访问带宽相互匹配，有效的降低了 RDB 算法的 I/O 开销。

5.3 AOF 重写操作性能测试

前面已经谈到，Redis 数据库为了降低 AOF 持久化产生的 AOF 文件冗余的问题，引入了 AOF 重写算法对 AOF 文件做瘦身操作。但是 AOF 重写算法对系统综合性能的影响极其巨大，尤其是随着数据库规模的增长，AOF 重写将是数据库中不可忽视的事件。因此，Redis 数据库强制将 AOF 重写操作放置在后台子进程来运行，但是 I/O 操作仍然会加剧系统总线的紧张状况。这一小节，我们将会测试 AOF 重写的 I/O 耗时情况。由于 AOF 重写操作在代码层面与 RDB 持久化非常的相近。因此，我们的总体测试方法与 RDB 持久化性能测试中使用的方法基本一样。唯一不同之处在于起始时间段的测量。我们以子进程创建为起始时间，并且在父进程从管道接收子进程完成信号为终止时间，在上述时间内，我们认为是 AOF 重写算法的有效执行时间。

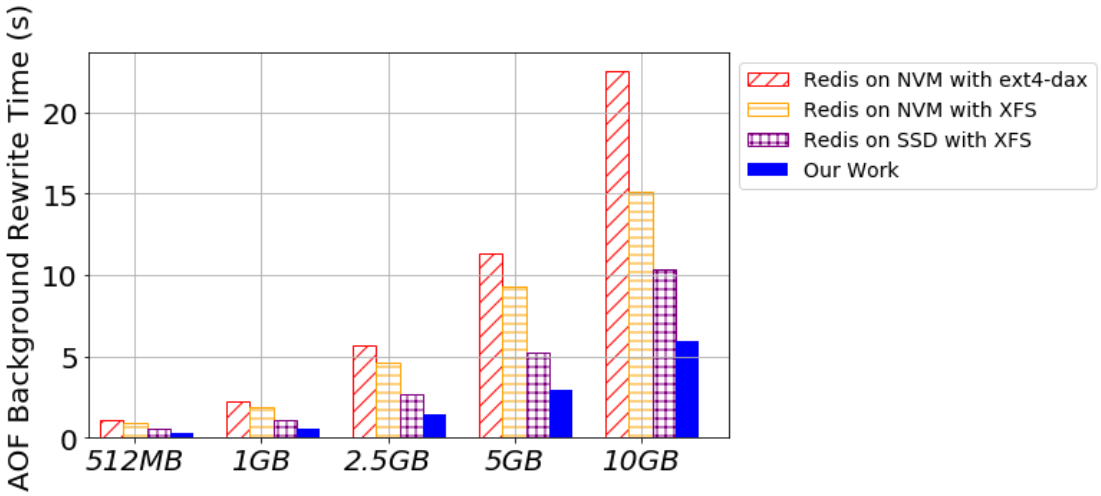

图 5-2　AOF 后台重写性能对比图

如图 5-2 所示，AOF 后台重写中，相同数据规模相同策略的 I/O 消耗时间与 RDB 持久化中 I/O 消耗时间相差无几。因此，RDB 持久化中 I/O 有关的结论也适应于 AOF 后台重写算法。

另一方面，5.3 中的实验也作为对照验证了实验 5.2 中对于 I/O 操作估计的正确性。即使是有着不同的 CPU 代价，但是通过实验 5.2 的方式可以正确的排除外界变量对实验的干扰。另外，我们的工作的确降低了 I/O 操作的开销，并且这种优势会随着 Redis 数据库规模的增长愈发的明显。

5.4 开启 RDB&AOF 混合持久化综合性能测试

在实际的 Redis 数据库使用场景中，配置策略并不会像上述两个实验使用场景那么理想化。有许多外界因素会对 Redis 数据库的综合性能产生影响。因此，在这一小节我们将会模拟 Redis 的高频请求，并且对相应高频请求的 Redis 做相应的评估测试。这一部分的实验使用 Redis 官方提供的基准测试工具 Redis-Benchmark 做相应的配置与分析。

Redis-Benchmark 基准测试工具是 Redis 数据库测试最常用的测试工具，通过模拟多个客户端发出不同的请求，计算出当前 Redis 数据库的峰值处理速度。在本实验中，我们模拟了 50 个客户端的请求状况，并且将请求数目从最低 6 万行提升至 600 万行，最终对上述数据计算平均数，这样得出的数据能够更加真实的反映出 Redis 数据库的实际使用场景。

无论是固态硬盘还是 NVM，最大的一个缺陷是读写不均衡。数据的读取速度一般远大于写速度，因此测试读操作的性能研究价值不大。这里，我们主要测试 Redis 数据库中的 SET 、INCR 、LPUSH、RPUSH 、SADD 以及 HSET 共六组写操作。为了得出更加准确的综合性能分析，我们分别执行上述六类操作 10000 次、50000 次、100000 次、500000 次以及 1000000 次，然后对得出的结果进行平均最终得到了高频请求写的平均性能指标。

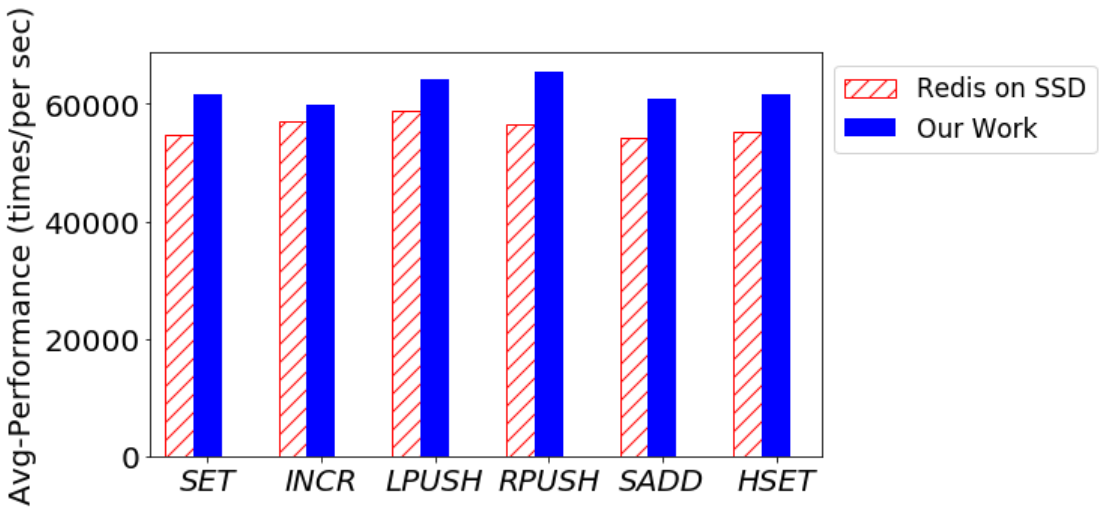

图 5-3　Redis 开启 RDB&AOF 混合持久化综合性能测试图

如图 5-3 所示，开启了 RDB & AOF 混合持久化之后，测试出的性能就是经过优化与迁移之后项目的最终性能提升情况。图中明显可以看出，Redis 数据库中最消耗资源的写请求都得到了一定的提升。其中 SET 请求性能提升了 12.7%，INCR 请求性能提升了 4.9%，LPUSH 请求性能提升了 9.1%，RPUSH 请求性能提升了 15.8%，SADD 请求性能提升了 12.1%，HSET 提升了 11.7%。综合性能提升 11.05%。总体来讲，整个项目的综合性能提升十分的明显，主要原因是通过优化 I/O 的处理操作，将热数据-冷数据模型转化为我们之前提出的四级模型之后，数据的存储有了进一步的提升。

上述的综合性能中，检验了我们项目中设计持久化部分的改动。其中 RDB 自动持久化策略为 1. 每 900 秒发生 1 次修改 2. 每 300 秒发生 10 次修改 3. 每 60 秒发生 1 次修改。满足以上三个条件的任意一条，都会执行 RDB 持久化操作。 AOF 本身为自动执行持久化操作，并且数据每隔一秒就会刷入存储介质中，但是随着数据量的增加，Redis 数据库将会执行 AOF 重写操作。在实验中发现，当客户端请求量超过 50 万行之后通常就会在日志中记载一次 AOF 重写操作。因此，这一小节实验的性能提升是综合了各种持久化策略的最终结果。同时也间接证明了 I/O 操作会对 Redis 数据库造成不可忽视的负面影响。

5.5 数据库规模压力测试

在我们的解决方案中，处理对数据库持久化的优化之外，同时也对 DRAM 的成本开销进行优化。限制 Redis 数据库使用的一个主要原因是 DRAM 成本价格过高，这对于商业中 Redis 的使用十分不利。我们之前的解决方案是使用 NVM 作为计算机系统的主存储器，并且使用一部分的 DRAM 来做上层次的 Cache 提供加速。

在这一小节的实验中，我们需要使用大量的数据来装载数据库，同时测试 Redis 数据库性能的变化情况。数据库的装载首先由之前的数据生成器来完成，之后我们用 Redis-Benchmark 压力测试工具来进行性能测试。

实验所用测试平台的 DRAM 大小为 384GB，Intel® Optane™ DC 持久化内存

的大小为 1.5TB，并且全部开启为 Memory 模式。因此，在操作系统层面显示出来的主存储器大小为 1.5TB。我们将会测试数据库规模为 100GB、200GB、300GB、384GB、500GB、800GB 以及 1024GB 的性能变化情况

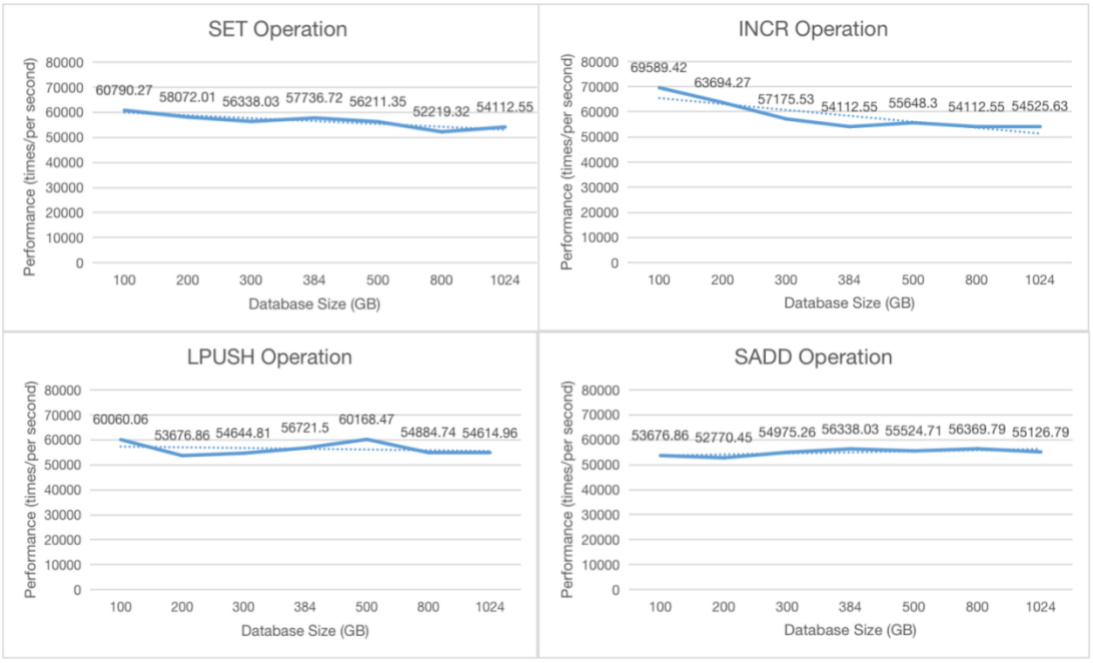

图 5-4　写操作性能测试图

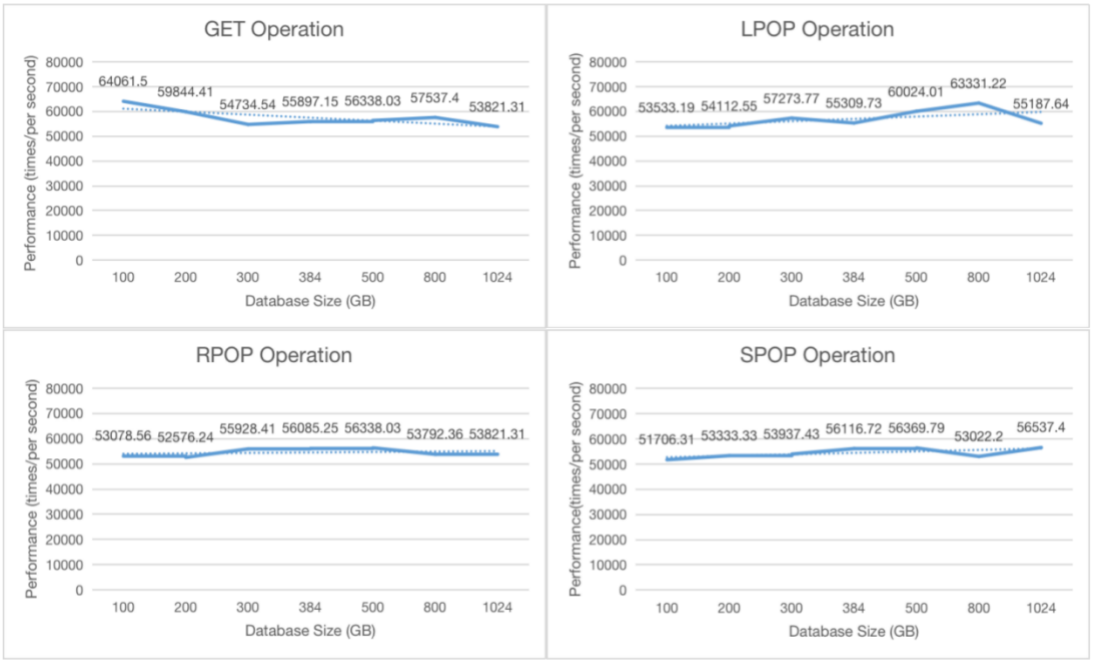

图 5-5　读操作性能测试图

图 5-4 中，我们首先测试了 SET、INCR、LPUSH 以及 SADD 共四组写操作。在图 5-5 中，我们测试了 GET、LPOP、RPOP 以及 SPOP 共四组读操作。实验数据总体表明，随着数据库规模的增长，性能并没有显著地变化。尤其是当数据库规模大于 384GB 之后，总体上数据库的性能并没有受到太大的影响。但是，从细节上来分析，写操作相比于读操作仍然在会随着数据库规模的增长发生下降。 造成这种下降的原因是 NVM 读写方面不均衡。值得一提的是，在我们搭建的实验平台中，Redis 数据库性能瓶颈早已不在是 NVM 的访存速度，而是网络相关的延迟、CPU 单核心频率等其它因素造成 Redis 数据库基本操作的波动。

在商业应用中，将 Intel® Optane™ DC 持久化内存作为系统的主存并使用部分 DRAM 作为 Cache 已经成为了主流的 Redis 数据库降低成本的解决方案。在这一小节的实验中，我们使用了 384GB 的 DRAM 以及 NVM 的配合后，成功的在维持 Redis 数据库效率基本不变的前提下运行了数据规模为 1TB 的 Redis 数据库。因此这种解决办法的的确确能够降低因为 Redis 数据库规模增长而带来的 DRAM 成本开销问题。

目前，相同容量的 Intel® Optane™ DC 持久化内存的价格为 DRAM 内存的 60%左右。我们的实验中，在保持 Redis 性能基本不变的情况之下，成本最高降低了 14.4%。

比较遗憾的一点是，受限于我们的工作量与时间限制，我们没有来得及去做实验验证要想使得 Redis 数据库最优经济化的 DRAM 与 Intel® Optane™ DC 持久化内存配比问题。不过总体而言，我们可以通过少量的 DRAM 以及相应代替的 Intel® Optane™ DC 持久化内存方案大幅度降低 Redis 数据库的运行成本。

6. 结论

随着社会信息化程度的提高，海量数据的处理与存储必然是一项极具挑战性的技术。以 Redis 数据库为代表的内存型数据库已经广泛运用在各种常见情形中。比如社交软件的发帖点赞数量、热搜排行等场景下都有着这类数据库的身影。尽管在一方面， 内存数据库受限于数据规模的增长，但是另一方面随着 NVM 的发展已经混合内存架构技术的成熟，内存空间紧张的问题也会迎刃而解。

与此同时，由于存储器技术的日益革新，传统金字塔形数据存储架构也相应的受到了挑战。一方面是处理器与 DRAM 内存之间的差距越来越大，另一方面，易失性存储器与传统非易失性存储器之间的差距也越来越大。我们提出的多级数据存储架构融入了新型存储器，并且对传统的金字塔形数据存储架构进行了一定程度的改进。改进之后，我们的工作可以缩小处理器与存储器性能之间差距，同时模糊了非易失性存储器与易失性存储器之间性能上的差距。另外，在多级数据存储架构的基础上提出了热数据——温数据——冷数据——冰数据的四级数据流动模型。在多级数据存储架构以及四级数据流动模型的基础之上，我们实现了本文的核心项目——基于混合内存架构的 Redis 数据库。其中，我们的工作相比于使用 SSD 为持久化介质的数据库，在 RDB 持久化方面，I/O 性能提升 45.4%。 AOF 重写方面，I/O 性能提升 44.7%，开启 RDB&AOF 混合持久化之后，综合性能提升 11.05%。并且使用 NVM 来替代大量的 DRAM 内存，使得成本下降了 14.4%。

混合内存技术目前还在处于起步阶段，还有很多技术需要去研究，很多应用场景需要去挖掘。另外，持久化内存编程组件在功能上并不是很完善。因此，随着越来越多的开发者进入到持久化内存开发社区之后，NVM 将会得到极大的应用。

Redis 数据库是以高效率著称的内存数据库，因此，如何优化 Redis 数据库并提高其性能是一个长久的话题。我们的工作主要以优化 Redis 持久化中的 I/O 性能为主。其实 Redis 数据库还有其他的优化之处，比如降低网络延迟，提升 CPU 执行效率等。

在未来，NVM 及其开发领域的工作主要集中在：

（1） 现有内存应用程序的移植工作。通过使用持久化内存编程组件对现有应用程序的移植，可以提升内存应用程序的综合性能。

（2） 混合内存架构的进一步优化。由于 NVM 读写性能不均衡，并且写有损耗问题，因此如何降低 NVM 的写次数将是一个非常重要的问题。

（3） 要想使 NVM 完全融入到传统的计算机系统中，还需要进一步的将持久化内存编程简单化。因此，开发者需要去适配各种编程语言、简化编程模型以及得到操作系统原生支持等工作，只有构建了 NVM 的开发生态，才能够促进 NVM 的发展。

参考文献

[1]https://redis.io/

[2]https://pmem.io/

[3]Izraelevitz J, Yang J, Zhang L, et al. Basic performance measurements of the intel optane DC persistent memory module[J]. arXiv preprint arXiv:1903.05714, 2019.

[4]Zhang Z, Ju L, Jia Z. Unified DRAM and NVM hybrid buffer cache architecture for reducing journaling overhead[C]//2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2016: 942-947.

[5]Liu D, Wang T, Wang Y, et al. Curling-PCM: Application-specific wear leveling for phase change memory based embedded systems[C]//2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2013: 279-284.

[6]Seok H, Park Y, Park K W, et al. Efficient page caching algorithm with prediction and migration for a hybrid main memory[J]. ACM SIGAPP Applied Computing Review, 2011, 11(4): 38-48.

[7]Chen X , Sha H M , Abdullah A , et al. UDORN: A design framework of persistent in-memory key-value database for NVM[C]// 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA). IEEE, 2017.

[8]Lee E, Bahn H, Noh S H. Unioning of the buffer cache and journaling layers with non-volatile memory[C]//Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13). 2013: 73-80.

[9]Zhou P , Zhao B , Yang J , et al. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology[J]. Acm Sigarch Computer Architecture News, 2009, 37(3):14.

[10]Kültürsay E, Kandemir M, Sivasubramaniam A, et al. Evaluating STT-RAM as an energy-efficient main memory alternative[C]//2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2013: 256-267.

[11]Parkin S S P, Hayashi M, Thomas L. Magnetic domain-wall racetrack memory[J]. Science, 2008, 320(5873): 190-194.

[12]Hady F T, Foong A, Veal B, et al. Platform storage performance with 3D

XPoint technology[J]. Proceedings of the IEEE, 2017, 105(9): 1822-1833.

[13]Zhang Z, Ju L, Jia Z. Reducing Journaling Overhead with Hybrid Buffer Cache[C]//International Conference on Algorithms and Architectures for Parallel Processing. Springer, Cham, 2015: 462-475.

[14]https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[15]https://software.intel.com/en-us/articles/quick-start-guide-configure-intel-optane-dc-persistent-memory-on-linux

[16]Carlson J L. Redis in action[M]. Manning Publications Co., 2013.

[17]Xu M, Xu X, Xu J, et al. A forensic analysis method for Redis database based on RDB and AOF File[J]. Journal of Computers, 2014, 9(11): 2538-2544.

[18]Cao M, Bhattacharya S, Ts'o T. Ext4: The Next Generation of Ext2/3 Filesystem[C]//LSF. 2007.

[19]https://redislabs.com/press/eharmony-selects-redis-labs-for-unmatched-performance/

[20]https://promotion.aliyun.com/ntms/act/redishybridstorage.html?spm=5176.54432.1380373.5.31701cf2mmZhPr&aly_as=zjMxYistk

[21]McClanahan C. History and evolution of gpu architecture[J]. A Survey Paper, 2010, 9.

[22]Goodfellow I, Bengio Y, Courville A. Deep learning[M]. MIT press, 2016.

[23]Mori Y, Yamada R, Tsukada S, et al. Dynamic random access memories and method for testing performance of the same: U.S. Patent 7,450,458[P]. 2008-11-11.

[24]https://www.amd.com/zh-hans/products/cpu/amd-ryzen-threadripper-3990x

[25]https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/

[26]Mohammad Banikazemi. LZB: Data Compression with Bounded References[C]// Data Compression Conference. IEEE Computer Society, 2009.

[27]Jun H, Cho J, Lee K, et al. Hbm (high bandwidth memory) dram technology and architecture[C]//2017 IEEE International Memory Workshop (IMW). IEEE, 2017: 1-4.

[28]https://github.com/antirez/redis

[29]Sweeney A, Doucette D, Hu W, et al. Scalability in the XFS File System[C]//USENIX Annual Technical Conference. 1996, 15.

[30]https://zedware.github.io/PMDK/

[31]Shoolman Y, Steinberg Y, Gottlieb Y, et al. Methods, systems, and media for controlling append-only file rewrites: U.S. Patent Application 15/639,965[P]. 2019-1-3.

附录——源代码

地址: https://github.com/bruceruc/Graduation-Project