持久性内存: 从系统软件的角度

陆游游 舒继武
清华大学

关键词：非易失性内存 内存存储 系统软件 文件系统

数据的存储和处理技术推动了计算机技术的快速发展。近二十年来，相比于计算密集型应用，数据密集型应用已经成为主流。高性能计算、大数据应用、物联网以及人工智能等领域均产生了海量数据，同时也对数据进行快速或实时的分析提出了要求。海量的数据存储和高效的数据处理需求对计算机存储系统提出了极大的挑战，内存存储与计算得到了学术界和工业界的广泛关注，比如提出了以 Spark 为代表的大数据处理框架以及以 SAP HANA 为代表的内存数据库等。

非易失性内存 (non-volatile memory) 近年来发展迅速。除了块粒度访问的闪存存储 (flash memory) 已经广泛应用于存储系统之外，字节粒度访问的非易失性内存技术也层出不穷，英特尔与美光公司研制的 3D XPoint 技术即将商用。字节粒度访问的非易失性内存可直接连接于内存总线上，被称为持久性内存 (persistent memory)。持久性内存以接近于动态随机存取存储器 (DRAM) 内存的速度提供数据的持久存储，为数据的实时处理提供了机会。

然而，持久性内存改变了传统易失性内存与持久性外存的结构。这一结构的改变对传统计算机系统软件的设计提出了挑战。如何基于持久性内存构建系统软件是近年来学术界关注的话题。

持久性内存的机遇

内存存储与计算既需要数据的大容量存储，也需要数据的高速处理。然而，在当前的计算机存储层次结构中，主存与外存之间的延迟差异远大于寄存器、CPU 缓存以及主存之间的延迟差异。例如，在一个计算机典型延迟中，寄存器的延迟约为 1 纳秒，CPU 缓存 (L2) 的延迟约为 10 纳秒，DRAM 主存的延迟约为 60 纳秒，然而外存磁盘的延迟约为 5 毫秒。相比于寄存器、CPU 缓存以及 DRAM 主存之间一个数量级的差异，DRAM 与磁盘之间的延迟差异高达 5 个数量级。由于这一差异，内存存储与计算的缓存不命中将极大地影响系统的性能。

然而，DRAM 和磁盘技术的发展均遇到瓶颈。

DRAM 的容量限制：DRAM 的容量限制不仅源于晶体管 - 电容式 (1T1C) 单元的密度受限，也源于难以扩展的动态刷新 (refresh) 操作。在 DRAM 单元中，电容需要足够大，以提供可靠读写；晶体管需要足够大以减少漏电，提高数据保存时间。在 DRAM 的芯片上，DRAM 单元需要在限定时间内完成刷新操作。当 DRAM 单元数量越来越多时，刷新操作所占比例越来越高。例如，当刷新一个 64Gb 的 DRAM 时，消耗了约 46% 的性能和 47% 的能耗[1]。因而，DRAM 在容量上难以扩展。

磁盘的性能限制：磁盘的 I/O 访问中存在盘片旋转和磁头定位两个机械式部件的操作。机械式部件的操作限制了磁盘性能的提升。近 20 年来，磁盘的带宽维持在略高于 100MB/s 的水平，延迟维持在数毫秒量级，几乎没有变化。正是机械部件的存在限制了磁盘的性能提升。

相比之下，持久性内存容量可扩展，且性能接近于 DRAM，是内存存储与计算中存储介质的有力

竞争者。

持久性内存的挑战

字节粒度访问的非易失性内存不但自身特点与DRAM和磁盘有很大差异，而且改变了传统计算机存储层次结构中易失性 - 持久性的边界，这两方面对构建持久性内存的系统软件均提出了新的挑战。

软硬件开销占比的变化

作为持久性存储，持久性内存与磁盘存在很大的差异，包括读写方式（内存总线与PCI总线读写）、读写粒度和读写性能等。与读写方式和读写粒度的差异对系统软件的操作方式带来的影响相比，读写性能的差异对持久性内存上的系统软件的效率要求带来的挑战更大。持久性内存的性能与DRAM内存接近，远高于磁盘和固态硬盘(Solid State Drive, SSD)的性能。加利福尼亚大学圣迭戈分校(UCSD)的报告[2]显示，在非易失性存储系统中，软件开销显著高于磁盘存储系统。在磁盘存储系统中，软件开销占比仅为0.3%。在持久性内存(DDR NVM)系统中，软件开销占比高达94.3%，如图1所示。因而，在持久性内存中，软件的高效设计是一个新的挑战。

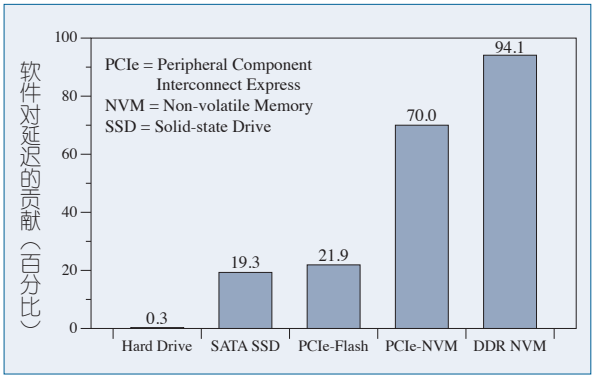

图1 存储系统中软件开销占比变化[2]

持久性内存系统软件的高效设计问题引出了一系列有趣的研究问题。硬件延迟的降低，操作系统自身的延迟不可忽略。在操作系统中，持久性内存I/O操作是否需要经过内核？系统调用和虚拟文件系统(VFS)层如何重新设计？文件系统的数据缓存如何管理？如何利用DRAM与NVM各自的相对优势？

易失性–持久性边界的变化

在传统存储层次结构中，寄存器、CPU缓存和DRAM主存均为易失性存储器，而外存为持久性存储器。在持久性内存存储系统中，寄存器和CPU缓存较大可能仍使用易失性存储器，而主存和外存采用持久性存储器。易失性 - 持久性边界从主存 - 外存边界上移至CPU缓存 - 主存边界[3]，如图2所示。然而，在主存 - 外存的边界上，操作系统对于DRAM主存中的缓存数据管理是白盒管理，可以通过软件形式对数据组织与写回进行灵活控制。在CPU缓存 - 主存边界上，CPU缓存的管理是硬件控制，对于系统软件是黑盒操作。系统软件难以灵活控制，而通过clflush等刷写命令对CPU缓存效率影响较大。易失性 - 持久性边界的变化对持久性存储系统中高效一致的数据写回也是新的挑战。

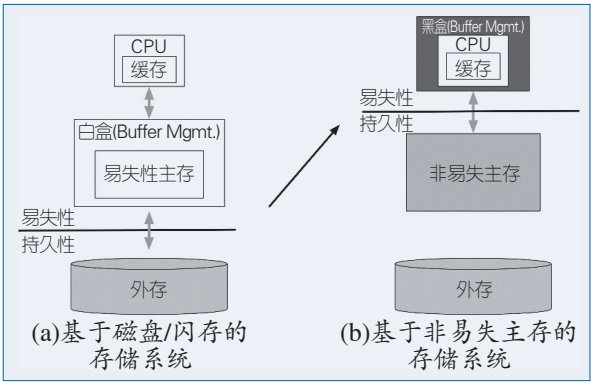

图2 易失性-持久性边界的变化[3]

由于易失性 - 持久性边界的变化，持久性内存系统的崩溃一致性(crash consistency)需要重新设计。如何设计高效的机制是新的难点。同时，这一边界的变化，使得程序中的数据可以直接持久性地存储在内存中，为程序中的堆数据提供了持久性保障“程序级别的数据持久保存与数据共享”这一新的编程模型，也成为持久性内存研究中的新内容。

持久性内存对操作系统的影响

针对持久性内存中系统软件的挑战，文件系统和内存管理需要新的设计以满足持久性内存对管理功能以及效率的需求。同时，持久性内存的出现促使了存储系统一致性管理的重新设计，新的编程模型也应运而生。

崩溃一致性

存储系统的一致性由并发控制和故障恢复两个部分保证。其中，故障恢复引入的一致性也被称为崩溃一致性。崩溃一致性要求系统遇到意外故障后，系统要么处于新的状态，要么处于旧的状态，不会出现中间状态。在传统的存储系统里面，崩溃一致性通过数据的版本有序写回的方式保证，比如文件系统日志 (journaling) 机制、数据库写前日志 (write-ahead logging)、影子页 (shadow paging) 等。在持久性内存中，数据缓存在 CPU 缓存中。由于 CPU 缓存为硬件控制，因而控制数据版本以及写回顺序性会极大地影响持久性内存系统的性能。针对崩溃一致性的研究可从软 / 硬件两方面优化持久性与顺序性。

以硬件方式优化顺序性：由于 CPU 缓存为硬件控制，如果在 CPU 缓存硬件中加入对一致性的支持，将会减少软件的开销。微软研究院早在其设计持久性内存文件系统 BPFS 时就提出了 Epoch 的硬件语义 [4]，即软件程序仅需发送 epoch 指令给 CPU 缓存，无须等待数据持久化返回；由 CPU 缓存硬件保证不同 epoch 之间的数据按序持久化。清华大学提出了 LOC(Loose-Ordering Consistency) 技术 [5]，将预测执行的概念引入数据持久化，放松事务持久化的顺序性要求，从而降低了顺序性带来的开销。密西根大学提出了 Strand Persistence 技术 [6]，允许在无依赖的程序片段中并发地执行顺序性约束，提高了并发性能。

以软件方式优化顺序性：传统的存储系统也多采用软件方式来弱化顺序性约束，以降低一致性开销，例如采用 Checksum、反向指针、计数等方式。在持久性内存存储中，Mnemosyne 采用了 TornBit 的做法 [7]，在每个数据块中保留一个比特位 torn bit，在写回时被置位，并与数据块原子性写回。由于在初始化时 torn bit 比特位被清空，可通过 torn bit 的置位状态判断多个数据块的原子性，而无须在写入路径上维护顺序，从而降低了一致性开销。

以硬件途径优化持久性：微软的研究人员提出全系统持久化 (Whole System Persistence, WSP) 技术 [8]，CPU 缓存采用非易失性存储器的方法，将存储层次结构中除寄存器之外的存储均设计为非易失性存储。寄存器的存储也可通过计算机中的剩余电量保证数据持久写回，以时刻保持系统的运行状态。这种技术虽然能防止意外掉电产生的崩溃一致性问题，但不能处理由程序 bug 带来的崩溃一致性等问题。而且将 CPU 缓存完全设计为非易失性存储的做法对于器件要求苛刻，短期内难以实现。Kiln[9] 仅在 CPU 末端缓存 (last-level cache) 上采用非易失性存储，通过末端非易失性缓存的使用降低了数据版本持久化的开销。

以软件方式优化持久性：清华大学提出的模糊持久化 (BPPM) 技术 [3]，允许未提交数据被写回持久性内存，通过日志数据组织方式使之可检测与可恢复；对于已有数据版本的已提交数据，允许其延迟持久化。通过有条件地放松数据在易失性与持久性状态上的转换，降低了一致性的开销。

近年来，尽管持久性内存的一致性技术涌现，然而尚未出现公认的简洁有效的方法，这一方向仍期待进一步的研究。

文件系统

在文件系统中，页缓存 (page cache) 是持久性内存文件系统中的一个主要性能开销。在持久性内存文件系统中，持久性数据直接放置于内存级别的存储器中，因而页缓存在内存级别的缓存是冗余的。为了避免页缓存的冗余数据拷贝问题，直写 (Direct Access, DAX) 方式被引入到传统文件系统中以提高其在持久性内存上的使用效率，如 Ext4-DAX。

字节访问粒度是持久性内存区别于持久性外存存储的新的访问特点。传统文件系统即使采用

DAX 技术仍然难以利用该特点。微软研究院设计并实现了面向字节粒度非易失性内存的持久性内存文件系统 BPFS[4]。BPFS 采用树状结构组织文件的数据与元数据，引入短路影子页 (short-circuit shadow paging) 方法提供高效的数据一致性。短路影子页利用 8 字节的原子更新特性以更新数据指针，从而避免在树结构的整个路径上进行数据拷贝，提高了效率。英特尔公司也设计并实现了持久性内存文件系统 PMFS[10]，同样也是为字节粒度访问设计的，从而避免了块粒度 I/O 中的数据拷贝与读改写 (read-modify-write) 开销。

然而，非易失性内存的读写延迟与 DRAM 内存的访问延迟有区别。非易失性内存的读写性能呈现不对称的特点，其读性能与 DRAM 接近，但写性能落后于 DRAM。清华大学的 HiNFS[11] 认为，直写方式并不是持久性内存文件系统最合理的方式，而传统缓存模式也存在优化空间。HiNFS 将直写 DAX 方式与传统的缓存方式相结合，对持久性内存文件系统中的读写操作以及不同持久化需求的写操作进行区分，提供了精细化的数据 I/O 控制，有效地提升了文件系统的性能。

NOVA[12] 持久性内存文件系统将日志 (log-structured) 结构引入到持久性内存文件系统。NOVA 采用了多日志的方式支持并发的元数据写操作，同时传统日志式结构中数据随机读的性能问题在持久性内存上也得到了有效缓解，从而达到较好的性能。

是否绕开操作系统内核以避免系统调用与虚拟文件系统层开销也是文件系统中一直讨论的问题。Aerie[13] 是一个用户态实现的内存文件系统。为了支持文件元数据的管理及数据共享，Aerie 采用一个单独的用户态进程作为可信任的第三方，实现元数据管理以及并发控制。用户态文件系统的缺点在于难以保证文件系统的读写安全，恶意进程可以随意读写文件数据。清华大学提出的 KucoFS，采用了用户态与内核态协同设计的方法，以内核态方式管理元数据，但将元数据操作通过卸载、授权 (lease) 等方式交由用户态辅助执行，既实现了内核态的数据保护，也提供了用户态灵活高效的数据读写。

内存管理

在持久性内存系统中，内存的碎片问题和内存分配一致性问题也显得更为严重。在传统易失性主存中，内存碎片在系统重启后即消失；在持久性内存中，内存数据在系统重启后仍然保存，因而碎片问题日积月累，更为严重。内存分配和释放时的空间管理元数据应能够在系统故障后恢复正确状态，因而需要提供崩溃一致性。

针对内存分配和释放的空间管理元数据一致性问题，Makalu[14] 依据数据结构中对象可达性区分空间管理元数据中的关键元数据和辅助元数据，通过延迟辅助元数据的持久化，降低空间分配时的开销。针对内存碎片问题，LSNVMM[15] 采用了日志式结构对持久性内存空间进行管理，通过垃圾回收可以减少碎片数量，从而缓解碎片问题。

持久性堆

在程序中，堆数据是执行过程中产生的数据。传统程序在进行数据持久化或数据共享时，需要将数据以文件的形式存储于文件系统中。在持久性内存中，倘若需要持久化或共享的数据可以被持久化索引数据结构索引住，这些数据就可以直接以内存数据结构的形式保存在持久性内存中，而无须通过文件系统的读写操作。这一类工作也被称为新型编程模型。

NV-Heaps[16] 和 Mnemosyne[7] 是其中较早出现的两项研究工作。NV-Heaps 是一个持久性对象存储系统，数据对象直接存储于持久性内存上，同时映射到程序堆空间上。为保证掉电后数据可达，NV-Heaps 检查指针类型，不允许出现持久性内存指针指向易失性内存对象的情形，同时也提供了内存分配的一致性。Mnemosyne 具有类似的功能，将持久性内存空间导出用于持久性对象的分配，并保证其一致性。Heapo[17] 基于原生堆的结构提供持久性堆的管理。英特尔公司也提供了 PMDK[18]（原称为 NVML）的用户态程序库，用于导出持久性内存空间，

以方便持久性内存的分配与管理。

尽管在持久性内存上有持久性堆与文件系统等不同系统构建形式的讨论，然而其中有很多相同的技术。持久性堆与传统的程序堆有较大的不同，为了实现数据的持久化存储与数据共享，持久性堆的数据对象需要采用持久化的索引结构，同样需要考虑持久性内存空间的分配与释放，以及内存空间上的数据组织与布局等问题。这些技术与文件系统、内存管理之间的技术是相通的。

持久性内存对分布式存储系统的影响

在分布式存储系统中，分布式的软件管理模块通常堆叠于本地存储系统之上，数据访问的路径冗长。因而，在持久性内存上，分布式存储系统的软件效率问题尤为严重。

清华大学将 GlusterFS 分布式文件系统分别布署于两种集群上：一种是以磁盘和千兆以太网构建的传统分布式集群，另一种是以内存和 56Gbps 的 Infiniband 网络构建的高速分布式内存集群。如图 3 所示，在 1KB 同步写的延迟测试中发现，尽管延迟从传统集群的 18 毫秒降低为内存集群的 324 微秒，但是软件开销占比从传统集群的 2% 急剧上升为内存集群的 99.7%。在 1MB 写的带宽测试中发现，传统集群中软件能够达到硬件裸带宽的 94%，而内存集群中软件仅能达到硬件裸带宽的 6.9%。因而，现有的分布式存储系统并不能有效地发挥出持久性内存等硬件的性能优势。

Octopus[19] 是一个面向非易失性内存和 RDMA 重新设计的分布式持久内存文件系统。它通过 RDMA 将分布式持久性内存构建成分布式持久内存池，减少数据在不同软件层次上的内存数据拷贝。针对高速网络与存储 I/O 中的中心化服务节点的处理瓶颈问题，Octopus 提出了客户端主动的 I/O 数据流，以网络代价换取服务器处理效率，将访问负载均摊至客户端，从而提高整体效率。基于 RDMA 的原语特性，Octopus 还提出了新的远程过程调用 (Remote Procedure Call, RPC) 原语以及分布式一致性协议。Octopus 有效发挥了硬件的性能，达到硬件裸带宽的 88% 以上，能大幅提升大数据处理的性能。

HotPot[20] 也是新提出的分布式持久内存框架，并在内核级别提供了持久命名机制与空间管理机制，向上层软件提供了透明的内存访问接口。

未来研究展望

持久性内存在学术界已经得到广泛的关注，近

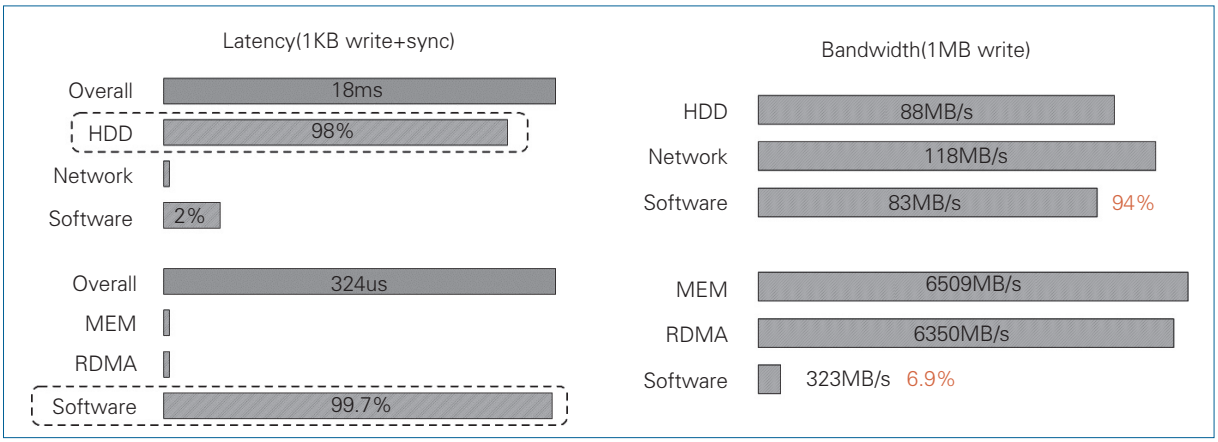

图3 GlusterFS在磁盘/以太网环境与内存/RDMA[1]环境下延迟与带宽的比较[19]

[1] RDMA，Remote Direct Memory Access，远程直接内存访问。

年来相关的研究论文也越来越多，然而以怎样的方式去用持久性内存、如何用好持久性内存等问题仍然需要更多的研究。随着研究的推进，我们认为持久性内存可能将在以下几个方面出现较大的变革。

1. 软硬件结合更紧密：与传统外存储相比，持久性内存提供了极高的数据持久化性能。在传统存储系统中，由于外存储较慢，软件处理开销几乎可以忽略；而在持久性内存中，软件处理开销尤为显著。进一步，在持久性内存系统中，CPU 若过多地参与到数据处理中，将显著影响应用性能。近年来 FPGA 以及新型硬件加速器的快速发展对 CPU 数据功能的卸载提供了新的机会。利用硬件的比较优势并实施软硬件协同的设计，将是持久性内存系统设计中一个重要的研究方向。

2. 系统软件的变革：持久性内存驱动计算机系统软件的变革，从基础的数据结构到内存与文件系统管理，再到系统调用、操作系统内核设计、以及分布式系统设计，都将出现新的变革。持久性存储介质的以字节粒度访问、读写不对称、磨损等新特性与传统存储介质有较大差异，这些新特性对系统软件设计提出了新要求，并将推动这些新要求被实现。

3. 新的编程模式与应用：持久性内存的快速发展一方面是因为硬件技术的发展，另一方面也是因为应用对数据访问苛刻的性能要求。以图计算为代表的内存计算应用呈现较多随机的字节访问模式，这对持久性内存更为友好，同时也需要对持久性内存的可扩展性和数据一致性等方面进一步支持。此外，如何结合持久性内存来优化或设计数据处理框架也有待进一步研究。 ■

陆游游

CCF 专业会员。清华大学助理教授。主要研究方向为存储系统、分布式系统和计算机体系结构等。

luyouyou@tsinghua.edu.cn

舒继武

CCF 会士。清华大学教授。主要研究方向为网络 / 云 / 大数据存储系统、基于非易失存储器件的存储系统与技术、存储安全与可靠性和并行 / 分布式处理技术等。

shujw@tsinghua.edu.cn

参考文献

[1] Liu J, Jaiyen B, Veras R, et al. RAIDR: Retention-aware intelligent DRAM refresh[C]//*Proceedings of the 39th Annual International Symposium on Computer Architecture* (ISCA). IEEE, 2012: 1-12.

[2] Swanson S, Caulfield A M. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage[J]. *Computer*, 2013, 46(8): 52-59.

[3] Lu Y, Shu J, Sun L. Blurred persistence: Efficient transactions in persistent memory[J]. *ACM Transactions on Storage* (*TOS*), 2016, 12(1):1-29.

[4] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory[C]//*Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (SOSP). ACM, 2009: 133-146.

[5] Lu Y, Shu J, Sun L, et al. Loose-ordering consistency for persistent memory[C]//*Proceedings of the IEEE 32nd International Conference on Computer Design* (ICCD). IEEE, 2014.

[6] Pelley S, Chen P M, Wenisch T F. Memory persistency[C]//*Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture* (ISCA). 2014: 265-276.

[7] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight persistent memory[C]//Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2011: 91-104.

[8] Narayanan D, Hodson O. Whole-system persistence[C]// Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2012: 401-410.

[9] Zhao J, Li S, Yoon D H, et al. Kiln: Closing the performance gap between systems with and without persistence support[C]//Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). ACM, 2013: 421-432.

[10]Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory[C]//Proceedings of

the Ninth European Conference on Computer Systems (EuroSys). ACM, 2014:1-15.

[11]Ou J, Shu J, Lu Y. A high performance file system for non-volatile main memory[C]//Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016: 12.

[12]Xu J, Swanson S. Nova: A log-structured file system for hybrid volatile/non-volatile main memories[C]//14th USENIX Conference on File and Storage Technologies (FAST 16). 2016: 323-338.

[13]Volos H, Nalli S, Panneerselvam S, et al. Aerie: Flexible file-system interfaces to storage-class memory[C]// Proceedings of the Ninth European Conference on Computer Systems (EuroSys ' 14). ACM, 2014: 1-14.

[14]Bhandari K, Chakrabarti D R, Boehm H J. Makalu: Fast recoverable allocation of non-volatile memory[C]// Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems , languages, and applications (OOPSLA' 16). 2016: 677-694.

[15]Hu Q, Ren J, Badam A, et al. Log-structured non-volatile main memory[C]//Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC). 2017:1-15.

[16]Coburn J, Caulfield A M, Akel A, et al. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories[C]//Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, 2011: 105-118.

[17]Hwang T, Jung J, Won Y. Heapo: Heap-based persistent object store[J]. ACM Transactions on Storage (TOS), 2015, 11(1): 1-21.

[18]Intel Corporation. Persistent Memory Programming[EB/OL].(2018-10-22)[2018-11-28]. https://pmem.io/pmdk.

[19]Lu Y, Shu J, Chen Y, Li T. Octopus: An RDMA-enabled distributed persistent memory file system[C]// 2017 USENIX Annual Technical Conference (USENIX ATC 17). 2017, July 12-14, Santa Clara, CA.

[20]Shan Y, Tsai S Y, Zhang Y. Distributed shared persistent memory[C]//Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017: 323-337.