中国科学: 信息科学 2021 年 第 51 卷 第 6 期: 869–899
SCIENTIA SINICA Informationis
《中国科学》杂志社
SCIENCE CHINA PRESS
评述

非易失主存的系统软件研究进展

舒继武*, 陈游旻, 胡庆达, 陆游游

清华大学计算机科学与技术系, 北京 100084
* 通信作者. E-mail: shujw@tsinghua.edu.cn

收稿日期: 2019–06–21; 修回日期: 2019–09–04; 接受日期: 2019–09–18; 网络出版日期: 2021–05–13

国家重点研发计划 (批准号: 2018YFB1003301)、国家自然科学基金重点项目 (批准号: 61832011) 和广东省科技创新战略专项项目 (批准号: 2018B010109002) 资助

摘要 互联网和物联网规模的迅速扩张促使全球数据存储总量呈现爆炸式的增长, 导致数据系统从计算密集型向数据密集型方向发展. 如何构建可靠高效的数据存储系统, 成为大数据时代迫切需要解决的问题. 相比传统磁盘, 非易失主存具有性能高以及字节寻址等优点, 这些独特的优势为高效存储系统的构建提供了新的机遇. 然而, 传统存储系统的构建方式不适用于非易失主存, 无法发挥出非易失主存的性能优势, 并且容易造成一致性开销高、空间利用率低、编程安全性低等问题. 为此, 本文分析了基于非易失主存构建存储系统面临的挑战, 在系统软件层次分别综述了空间管理机制、新型编程模型、数据结构、文件系统和分布式存储系统等方面的研究进展, 并展望了基于非易失主存构建存储系统的未来研究方向.

关键词 非易失主存, 系统软件, 空间管理机制, 编程模型, 数据结构, 文件系统, 分布式系统

1 引言

互联网和物联网规模的迅速扩张促使全球数据存储总量呈现爆炸式的增长. 根据 IDC 公司的报告, 全球数据总量正以每两年翻一番的速度快速增长, 到 2020 年将达到 44 ZB 的规模 [1]. 此外, 大数据种类更加复杂多变, 超过 85% 以上的数据将以非结构化或者半结构化的形式存在 [2]. 因此, 数据密集型应用逐渐成为大数据时代的主流应用, 这对存储系统的性能提出了越来越高的要求. 然而, 随着多核处理器技术的快速发展, 外存存储性能的提升速度远低于处理器性能的提升速度 [3], 外存存储系统将难以满足日益增长的应用需求. 因此, 如何设计构建高效可靠的数据存储系统, 支撑应用层越来越高的性能需求, 成为了大数据时代迫切需要解决的问题.

近年来, 新型非易失存储器 (non-volatile memory, NVM) 以高集成度、低静态能耗、持久性和接近 DRAM 的性能等特性, 为存储系统的发展带来巨大的机遇, 吸引了研究人员的广泛关注. 目前主流

引用格式: 舒继武, 陈游旻, 胡庆达, 等. 非易失主存的系统软件研究进展. 中国科学: 信息科学, 2021, 51: 869–899, doi: 10.1360/SSI-2019-0128
Shu J W, Chen Y M, Hu Q D, et al. Development of system software on non-volatile main memory (in Chinese). Sci Sin Inform, 2021, 51: 869–899, doi: 10.1360/SSI-2019-0128

的非易失存储器包括相变存储器 (phase change memory, PCM)[4]、自旋矩存储器 (spin transfer torque RAM, STT-RAM)[5]、阻变存储器 (resistive RAM, RRAM)[6]、赛道存储器 (racetrack memory, RM)[7], 以及 Intel 公司和 Micron 公司联合开发了 3D XPoint 非易失存储技术[8]. 其中, 基于 3D XPoint 技术的 PCIe 接口设备已于 2017 年进入市场, 而基于 DIMM 接口的 Optane DC 持久性内存也于 2019 年 4 月刚刚发布. 因此, 可以利用这些非易失存储器构建高吞吐、低延迟的大内存存储系统. 随着非易失存储器技术的进一步发展与应用, 基于该技术构建持久性主存存储系统是满足应用日益增长的性能需求的有效方法. 然而, 相比于磁盘或者闪存等传统外存存储介质, 非易失主存的特性存在较大的差异, 如何构建持久性主存存储系统仍然面临诸多需要解决的难题.

(1) 软件栈开销高. 非易失主存将持久性数据读写访问延迟从毫秒级降至纳秒级, 而传统存储架构是针对外存设计的, 持久化路径上的软件栈开销较高, 难以发挥新型存储器件的性能优势.

(2) 一致性开销高. 非易失主存提供主存层次的数据持久性, 而处理器的片上缓存系统依然是易失性的, 系统故障可能导致非易失主存上的持久性数据处于不一致的中间状态, 而传统的一致性技术往往会导致过高的持久化延迟, 严重降低非易失主存系统的性能.

(3) 空间利用率低. 非易失主存的价格明显高于传统外存, 传统主存空间管理机制容易引入主存碎片, 主存碎片问题将显著降低非易失主存的空间利用率, 增加系统的成本.

(4) 编程安全性低. 主存层次的数据持久性导致编程中引入的程序错误变得更加难以解决, 例如野指针、多次释放同一个对象、内存泄露等问题, 即使系统重启也无法解决主存中存在的程序错误.

综上所述, 传统存储系统的构建方式不仅无法发挥出非易失主存的性能优势, 而且易于产生一致性开销过高、空间利用率过低、编程安全性过低等方面的问题. 目前, 基于非易失主存构建存储系统已成为学术界和工业界的热点研究问题. 本文首先介绍现有存储系统在非易失主存的系统软件设计上面临的独特挑战和需要解决的问题, 然后在此基础上从非易失主存的空间管理机制、新型编程模型、数据结构、文件系统和分布式存储系统等 5 个方面详细综述现有研究工作的进展, 最后展望未来的研究趋势和方向.

2 非易失主存系统软件设计的挑战及需要解决的问题

相比于传统磁盘或固态硬盘, 非易失内存表现出完全不同的硬件特性, 这促使我们重新设计新的系统软件, 以更好地利用其硬件属性. 本节将首先论述面向非易失内存的系统软件设计将面临的挑战, 然后指出系统软件设计过程中需要解决的问题.

2.1 非易失主存的挑战

现有的存储系统针对磁盘或者闪存的特性设计了大量的优化策略. 但是, 字节寻址的非易失主存提供主存层次的数据持久性, 可在主存层次构建持久性存储系统, 这对存储系统的构建方法提出了新的挑战, 具体表现在:

(1) 软件栈开销. 传统存储架构、软件及各模块都是针对磁盘或者闪存设计的, 难以发挥出新型存储器件的性能优势. 例如, 现有的存储系统除了维护易失性主存中的数据外, 还需要将关键数据以不同于主存对象的组织方式 (如文件格式) 写入到外存中, 从而确保在程序正常结束或系统突然断电时数据的持久性. 然而, 将数据从主存持久化到外存的过程会触发昂贵的软件栈延迟, 例如, 与外存设备进行页交换、在 (反) 序列化时更改数据格式, 以及触发臃肿的系统调用等操作均会造成严重的系统开销. 此外, 相比磁盘或者闪存, 非易失主存将持久性数据读写访问延迟从毫秒级降至纳秒级. 因此, 上

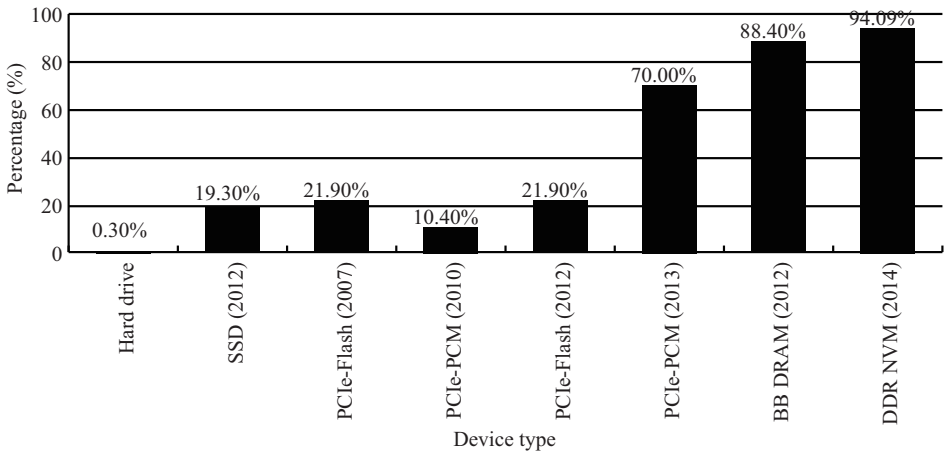


图 1 传统持久化路径的软件栈开销 [9]

Figure 1 Software stack overhead in the traditional persistence path [9]

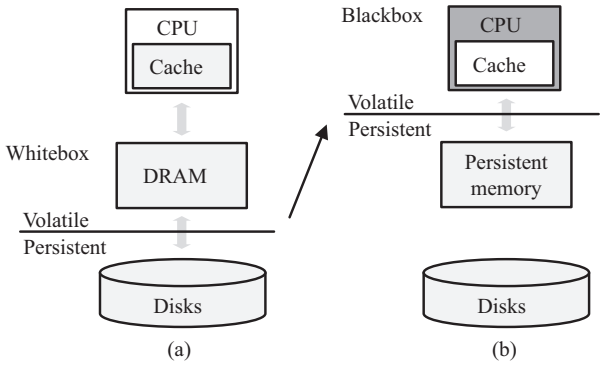


图 2 易失性 – 持久性边界的变化 [10]

Figure 2 The change of the volatile-persistent boundary [10]. (a) Disk-based storage system; (b) NVM-based storage system

述的软件开销占比还将进一步被放大.

图 1 [9] 对比了使用不同接口访问不同存储介质时软件栈开销在存储总开销中的占比. 如图所示, 在基于磁盘的传统存储系统中, 由于访问持久性数据的性能瓶颈在于硬盘的访问延迟, 所以软件栈开销占比仅为 0.30%; 而在基于 DIMM 接口的非易失主存存储系统中, 因为访问持久性数据的延迟大幅降低, 所以软件系统的开销比例大幅增加, 软件栈开销占比高达 94.1%. 因此, 基于非易失主存的存储系统需要设计更加高效的软件接口, 减少软件栈开销.

(2) 一致性开销. 非易失主存同时具备接近 DRAM 的性能和数据持久性, 消除了传统易失性主存和持久性外存的边界, 从易失性主存 + 持久性外存的 “两级结构” 变为了非易失性主存 “单级结构”. 这种结构上的颠覆性变化带来了一致性机制维护上的挑战.

如图 2 [10] 所示, 在基于外存的传统存储系统中, 易失性 – 持久性边界位于主存和外存之间; 而在基于非易失主存的存储系统中, 易失性 – 持久性边界位于处理器缓存和主存之间. 虽然非易失主存提供了主存层次的数据持久性, 然而处理器的片上存储系统 (例如处理器缓存) 依然是易失性的, 程序错误和电力故障都可能导致处理器缓存中的数据丢失, 而位于非易失主存的数据可能处于未完成的中间

表 1 不同存储器件的价格 [16]

Table 1 The price of different memory technologies [16]

Storage device	Price
HDD	< \$0.06/GB
SSD	< \$0.9/GB
Battery-backed DRAM	\$12/GB
PCM	\$2∼8/GB
3D-XPoint	\$2∼8/GB
STT-RAM	The highest

状态, 从而导致持久性数据在系统重启后出现不一致性的问题. 因此, 处理器缓存中的数据需要原子性地写回到非易失主存. 由于目前 64 位计算机只支持 8 字节数据的原子操作 [11], 系统设计者需要额外的机制保证数据的一致性 [10∼13]. 然而, 非易失主存往往存在读写不对称的特性, 写操作会带来更高的延迟和能耗, 因此, 额外引入的一致性机制往往会引入过高的持久化开销 [13].

此外, 在原有双层结构中, 传统存储系统通过软件的方式控制需要持久化的主存页, 系统性能主要由外存的数据组织及主存的数据管理决定, 而处理器缓存效率对存储系统影响较小. 基于持久性内存的单层存储体系架构下, 处理器缓存是由硬件控制的, 大多数现代处理器通过对主存写操作进行重排序来提高系统性能, 因此, 处理器缓存效率直接影响了内存级存储系统的性能. 然而, 这些优化机制可能打乱数据持久化到非易失主存的顺序, 导致持久性数据在系统故障时处于不一致的状态. 为了解决这个问题, 系统设计者需要通过硬件刷写指令 (如 clflush, clflushopt 等) [14] 保证数据持久化操作的顺序: 这些硬件刷写指令能确保将某个数据对应的处理器缓存行替换到非易失主存中. 然而, 这些刷写指令往往十分昂贵, 可产生高达 200 ns 的延迟 [15]. 因此, 如何确保处理器缓存中的数据及时有序地写回到非易失主存, 高效地保证存储系统数据的一致性, 成为系统设计者亟待解决的问题.

(3) 空间利用率. 随着非易失存储器技术的进一步成熟, 非易失主存不仅可以保证主存层次的数据持久性, 而且可以达到高于传统 DRAM 主存的容量. 然而, 非易失主存的价格依然远高于传统持久性存储器, 非易失主存的空间利用率直接决定了存储系统的成本.

如表 1 [16] 所示, 以 PCM/3D-XPoint/STT-RAM 为代表的非易失主存, 它们的价格远远高于磁盘和闪存. 然而, 传统的主存分配器容易产生主存碎片, 这主要包括两种不同类型的碎片 [13]: 第 1 种是内部碎片, 以 Intel 公司的 PMDK [12] 系统为例, 它将任意一个主存分配操作的大小与 64 B 的倍数对齐. 如果应用申请 65 B 的主存, PMDK 会分配给它 128 B, 这样就产生了 63 B 的内部碎片. 第 2 种是外部碎片, 假设应用释放了一系列不连续的 64 B 主存区域, 但是在这些区域周围的区域依然在使用中, 那么这些释放的区域依旧不能作为有效空间被分配给更大的数据分配操作. 当主存分配操作的大小频繁发生变化时, 外部碎片会变得十分严重 [17]. Stanford University 的研究人员发现主存碎片可能会占用超过 50% 的主存空间 [18].

与易失性主存不同的是, 非易失主存中的数据将被一直保存下去, 即使系统重启也无法消除已经存在的主存碎片, 所以, 碎片问题在非易失主存上将会变得更加严重. 如果缺乏有效的碎片消除机制, 这将会严重影响非易失主存的可用性和成本. 高级编程语言, 例如 Java 或者 C#, 尝试采用垃圾回收的策略减少主存碎片, 但是引入了较高的对象引用分析和程序中断成本 [19]. 此外, 非易失主存的容量和写延迟均高于基于 DRAM 的传统主存, 所以对象引用分析和程序中断成本将更加严重.

(4) 编程安全性. 由于非易失主存在内存层次提供了数据持久性的保证, 大量传统程序错误在非

表 2 非易失主存上的指针错误 [20]

Table 2 The pointer errors on non-volatile main memory [20]

Pointer types	Yes/No
Persistent pointer → volatile data	No
Volatile pointer → persistent data	Yes
Intra-heap	Yes
Inter-heap	No

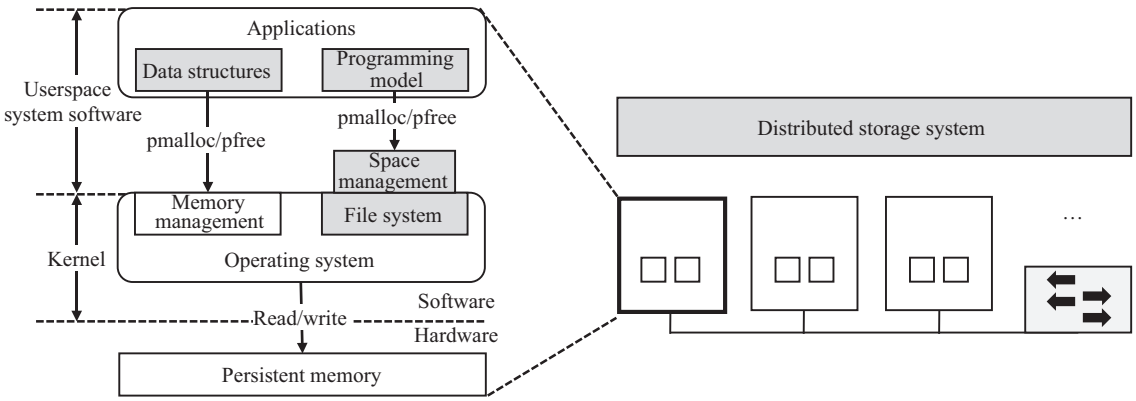


图 3 非易失主存的系统软件层次

Figure 3 The system software level for non-volatile main memory

易失主存上会导致严重的问题. 例如, 多次释放同一个对象、主存泄露等. 更糟糕的是, 任何一个系统崩溃会将这些程序错误永久性地保存下来, 在系统重启后依然可能导致系统出现崩溃, 因而这些程序错误变得更加危险.

此外, 非易失主存还可能引入一些新的指针错误. 表 2 [20] 说明了非易失主存上可能存在的指针错误类型: (1) 当应用程序使用持久性指针指向易失性数据时, 系统重启后易失性数据会丢失, 这可能导致持久性指针指向一个未知的数据, 产生野指针访问的错误; (2) 当应用程序使用某个持久性堆结构中的指针指向另一个持久性堆结构的数据时, 系统重启后另一个持久性堆结构的地址可能会发生变化, 这同样可能导致野指针访问的错误. 因此, 非易失主存需要设计保证安全性的编程模型和保障机制, 有效避免程序员编程过程中引入的软件错误.

2.2 非易失主存的系统软件层次需要解决的问题

非易失主存的特性对现有存储系统的构建带来巨大的挑战. 近年来, 基于非易失主存的系统软件研究得到学术界和工业界的广泛关注. 如图 3 所示, 在系统软件的设计上, 研究者们尝试从以下 5 个方面解决非易失主存带来的挑战.

(1) 非易失主存的空间管理, 主要解决 3 个问题: (a) 支持上层应用以用户态的主存接口直接访问持久性数据, 减小传统持久化路径上的软件栈开销, 发挥出非易失主存良好的性能; (b) 保证持久性主存分配/释放操作的一致性, 避免出现主存泄露和多次分配/释放同一个地址等问题; (c) 设计主存碎片处理机制, 提高非易失主存的空间利用率, 降低系统的成本.

(2) 非易失主存的编程模型, 主要解决 3 个问题: (a) 提供自管理的编程接口, 保证程序执行过程中程序数据的持久性和一致性, 减小软件栈开销; (b) 设计高效的事务机制, 减小日志机制引入的一致

性开销; (c) 设计可靠的编程模型和保障机制, 减少用户编程过程中可能引入的编程错误.

(3) 非易失主存的数据结构, 主要解决一致性开销过高的问题. 索引数据结构是影响存储系统性能的关键模块, 然而持久性索引结构需要保证操作的一致性, 过高的一致性开销会严重降低系统的性能, 这对索引结构的设计提出了新的挑战.

(4) 非易失主存的文件系统, 主要用于降低内核软件栈引入的开销. 为屏蔽不同文件系统的差异性, 内核引入了虚拟文件系统 (virtual file system, VFS) 对文件系统进行统一抽象, 并提供了标准化的访问接口. 但是, VFS 维护的元数据缓存、数据页缓存, 以及厚重的软件层严重影响系统性能.

(5) 非易失主存的分布式存储系统, 主要解决传统分布式系统软件性能低下的问题. 传统分布式系统软件栈厚重, 数据冗余拷贝现象严重, 协议复杂, 且通用的分布式架构在新硬件下难以扩展.

3 非易失主存的空间管理

非易失主存空间管理主要涉及软件接口设计、分配/释放操作的一致性和主存碎片处理机制等 3 方面的研究工作.

3.1 软件接口设计

现有的研究工作主要通过两种方式将非易失主存集成到传统的计算机系统中: 字节寻址的文件系统 [21] 和持久性堆结构 [10~13, 20, 22~26]. 字节寻址的文件系统通过文件目录树组织非易失主存空间, 支持通过 POSIX 接口访问非易失主存中的持久性数据. 这种方法可以继承来自文件系统的所有功能, 例如分层的命名系统、完善的共享保护机制和可扩展的元数据结构等. 然而, 文件系统会引入昂贵的软件栈开销, 包括系统调用、块接口访问开销等. 此外, 文件系统是针对块设备设计的, 难以高效地利用处理器缓存和页表转换缓冲区. 因此, 直接使用文件系统的方式会引入较高的软件栈开销, 难以发挥出非易失主存性能的优势, 并且严重限制主存访问操作的灵活性, 例如指针操作等.

为了降低软件栈开销, 研究人员设计了持久性堆结构. 相比文件系统, 持久性堆结构直接将非易失主存设备映射到进程地址空间中, 消除了持久性数据在块设备和虚拟主存地址之间的序列化开销. 在传统易失性主存中, 应用程序通过 malloc/free 编程接口动态分配主存, 然后通过 CPU load/store 指令在用户态访问主存数据. 在非易失主存中, 持久性堆结构同样为程序员提供了诸如 pmalloc/pfree 的用户态主存分配接口, 为上层应用程序提供对持久性数据的直接访问功能. 然而, 与传统易失性主存不同, 除了提供主存分配和释放功能, 非易失主存的空间管理还承载着其他功能.

(1) 命名系统和权限管理机制. 非易失主存提供主存层次的数据持久性, 这要求系统即使在重启后, 也可以通过特定的命名规则定位到对应的持久性数据. 此外, 用户可以对持久性数据进行权限管理, 控制持久性数据的共享范围.

(2) 易失性/持久性切换机制. 非易失主存具有高于 DRAM 的集成度, 解决了主存容量扩展性有限的问题. 因此, 非易失主存不仅可以作为持久性存储用于存放持久性数据, 而且当易失性主存容量不足时可以作为易失性主存用于存放易失性数据.

3.1.1 命名系统和权限管理机制

现有的研究工作主要通过以下两个方面提供命名系统和共享保护机制.

(1) 基于原生堆的持久性堆结构. 汉阳大学设计了基于原生堆的持久性堆结构 HEAPO [25]. 它为持久性堆结构预留了极大的虚拟地址空间 (例如 32 TB), 并将非易失主存设备映射到这块虚拟地址空

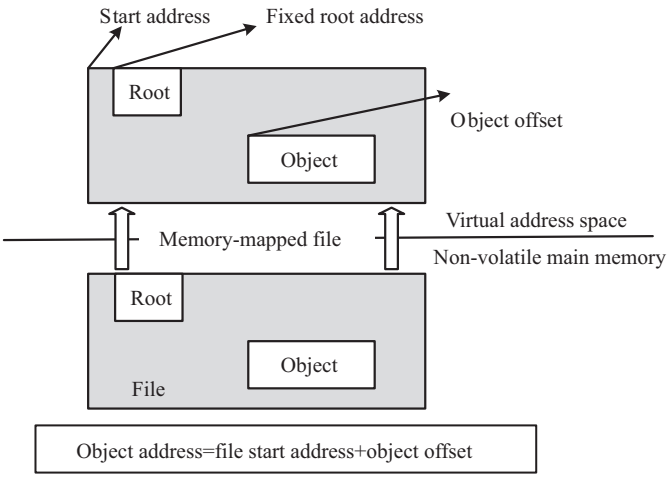


图 4 PMDK 的组织结构

Figure 4 The architecture of PMDK

间中. 它利用现有的 Linux 内存管理算法 (伙伴算法) 管理非易失主存空间. 对于每一个分配的对象, 它按照页粒度 (4 KB) 进行分配, 并将其绑定在某个固定的进程虚拟地址上.

HEAPO 为每个对象分配了一个对象名, 并在内核态利用字典树维护命名空间, 在用户态利用 Hash 表维护字典树的缓存, 从而实现了命名系统. 同样, HEAPO 为每个对象配置了访问权限控制: 任何一个进程在访问一个对象时, 都需要通过命名系统找到对象的位置, 并检查是否有权限访问这个对象.

(2) 基于主存映射文件的持久性堆结构. 虽然基于原生堆的持久性堆结构支持命名系统和权限管理机制, 但是 4 KB 大小的对象分配粒度可能导致较低的空间利用率, 并且每个新分配的对象都会引发较高的内核态操作开销. 目前的文件系统, 例如 ext4 [27] 和 PMFS [21], 支持应用对持久性数据的直接访问 (direct access, DAX), 即避免了将持久性数据拷贝到 DRAM 中的额外开销. 应用通过 mmap 接口, 将对应的非易失主存区域以文件的方式映射到某个特定的虚拟主存地址, 然后通过用户态的主存访问接口访问数据, 文件系统并不会对持久性堆结构的性能产生太大的影响. 在系统重启后, 用户同样将文件通过 mmap 接口映射到进程地址空间中, 就可以继续执行正常的操作. 通过上述方式, 非易失主存利用文件系统提供命名系统和权限控制机制, 简化了持久性堆结构的设计. 目前, 大量系统都采用了这种方式 [10~13, 20~26].

持久性内存开发库 (persistent memory development kit, PMDK) [12] 是由 Intel 公司设计的新型编程模型 (之前被命名为 NVML). 它利用基于主存映射文件的持久性堆结构, 使程序能够在用户态访问非易失主存的持久性数据. 图 4 描述了 PMDK 的空间组织结构, 它通过主存映射文件的方式, 将一块非易失主存映射到进程的某块虚拟地址空间, 然后在用户态对这块主存空间进行管理. 上层应用通过 PMDK 的分配接口申请相应大小的持久性对象, 并用持久性指针指向分配到的对象. 因为系统在重启后可能会将相同的非易失主存文件映射到不同的虚拟地址, 所以 PMDK 中的持久性指针需要两个变量: 所在文件的 ID, 通过它可以找到文件映射到虚拟地址空间的起始地址; 对象在文件中的偏移量. 此外, PMDK 在每个文件上都预留了一个固定地址的根指针. 基于上述信息, 系统在重启后可以通过固定地址的根指针, 找到其他合法对象所在的位置.

3.1.2 易失性/持久性切换机制

当非易失主存作为持久性存储设备时, 不论是文件系统还是持久性堆结构, 目前的研究工作都需要依赖虚拟文件系统 (VFS) 将非易失主存映射到进程的地址空间, 上层应用可以通过主存访问接口访问非易失主存上的数据. 然而, 当易失性主存的容量不足时, 非易失主存无法直接作为易失性主存使用. 现有的工作缺乏将非易失主存切换成不同主存类型的功能, 从而导致当易失性主存容量不足时主存数据频繁被交换到外存中, 严重影响应用程序的性能. 此外, 因为文件系统的元数据 (inode 或者 superblock) 是针对块设备设计的. 即使系统利用 VFS 构建持久性堆结构满足易失性应用的容量需求, VFS 也会因为较高的处理器缓存和页表转换检测缓冲区的缺失率带来较高的软件开销 [28].

为了解决上述问题, Georgia Institute of Technology 提出了 pVM [28]. 传统易失性主存是由虚拟内存 (virtual memory, VM) 管理的. pVM 通过修改 VM, 将非易失主存抽象成为一个非一致性存储器访问架构 (non uniform memory access architecture, NUMA) 节点, 实现了应用透明的自动内存容量扩展功能. 因为 pVM 无需文件系统的支持, 所以它保留了 VM 子系统在处理器缓存和页表转换检测缓存中良好的命中率, 提高了系统的性能.

3.2 分配/释放操作的一致性

非易失主存的空间管理主要包含分配和释放操作. 在系统执行主存分配/释放操作的过程中, 系统错误 (例如断电或者程序崩溃) 可能导致操作处于不一致的非法状态, 从而导致系统重启后出现主存泄露或者野指针访问等错误.

下文以分配操作为例, 说明分配操作可能导致的一致性问题. 分配操作主要包含两个持久化操作: 修改并持久化分配器的元数据, 用于表示这块主存区域已经被分配; 上层应用使用持久性指针指向分配到的主存区域, 从而在系统重启或崩溃后避免这块区域丢失. 非易失主存的分配操作需要保证上述两个持久化操作的原子性, 否则会导致内部错误和外部错误 [22]. 内部错误指的是分配器元数据的错误, 它可能导致同一个主存区域被多次分配或者某块主存区域泄漏. 外部错误指的是上层应用没有使用持久化指针指向分配到的数据区域, 导致这块区域泄漏或者出现野指针错误. 因此, 非易失主存的空间管理机制需要保证分配/释放操作的一致性. 值得注意的是, 数据本身的一致性将由上层应用来保证, 这将在第 4 节中详细说明.

为了解决这个问题, 很多研究工作提出了基于事务机制的主存分配接口, 例如 PMDK [12], NVM DIRECT [26], Mnemosyne [11] 和 NV-Heaps [20] 等. 它们利用事务机制保证两个持久化操作的原子性, 从而确保分配/释放操作的一致性. 然而, 事务机制往往利用 redo/undo 日志机制保证事务的一致性. 日志机制需要在修改某个数据之前, 先将新/旧数据持久化到日志中, 这会带来写放大问题, 引入了昂贵的持久化开销. 因此, 如何降低主存分配/释放操作的一致性开销, 成为了一个重要的研究问题.

Hasso Plattner Institute 设计了 nvm_malloc 分配器 [24]. 它将分配过程分为 3 个步骤: 预留内存、初始化和激活发布. 它将前两个步骤从事务过程中分离, 从而降低了分配操作的事务开销. 然而, nvm_malloc 使用了较为复杂的主存分配接口, 与传统主存分配器差异较大, 并且依然依赖事务机制保证分配操作的一致性, 产生较高的一致性开销.

惠普公司设计了 Makalu 分配器 [22]. 如图 5 所示, 它基于一个重要的假设: 当堆结构处于一致性的状态时, 所有已分配的主存区域在系统重启后可以通过一组已知的持久性根对象可达, 其他不可达的主存区域均为未分配的区域. 基于该假设, 它将分配器元数据分为关键元数据和辅助元数据. 关键元数据代表系统崩溃后无法恢复的元数据, 例如上层应用的持久性指针和主存超级块的切割信息; 而

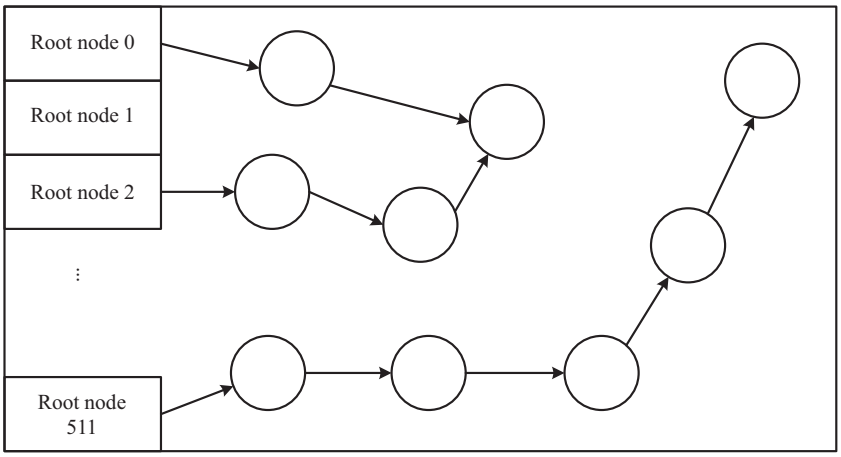


图 5 合法对象从根节点可达

Figure 5 The legal objects are reachable from the root nodes

辅助元数据可以通过合法的关键元数据进行恢复, 例如分配器元数据中用于记录哪些区域被分配的位图信息.

为了减小分配操作的一致性开销. Makalu 只将关键元数据维持在非易失主存中, 而将辅助元数据维持在易失性主存中, 并在系统重启时对辅助元数据进行恢复. 当系统执行分配/释放操作时, 它只需要确保上层应用修改持久性指针操作的原子性, 而不需要确保修改分配元数据的原子性, 从而避免了分配操作的事务开销. 当系统重启时, Makalu 通过存放在固定地址的根对象集合, 离线扫描整个非易失主存, 从而找到所有已分配的主存区域, 回收尚未分配的主存区域, 有效地避免了主存泄漏问题. 然而, 虽然 Makalu 降低了分配/释放操作的一致性开销, 但是它需要在系统重启时扫描整个非易失主存, 不可避免地引入了较高的恢复延迟.

为了降低分配操作一致性开销的同时提高系统恢复的性能, Technische Universität Dresden 设计了 PAllocator 分配器 [23]. 对于小于 16 KB 的主存分配操作, PAllocator 在非易失主存的固定位置预留了恢复指针, 用于记录上层应用的持久性指针地址, 从而在更新分配器元数据时无需考虑持久性指针更新操作的原子性, 减少分配操作的事务开销. 在系统恢复时, PAllocator 只需要扫描恢复指针, 就可以快速恢复这部分的分配器元数据. 对于大于 16 KB 的主存分配操作, PAllocator 利用基于混合主存的 FPTree [29], 降低系统恢复的开销. FPTree 通过只维护部分关键元数据的一致性, 降低了更新元数据的一致性开销. 当系统重启时, PAllocator 只需要扫描很小的非易失主存空间, 得到合法的关键元数据就可以重建 FPTree, 从而有效地降低了系统的恢复开销.

3.3 主存碎片处理机制

与易失性主存不同, 非易失主存的数据即使在系统关机后也会被保留下来, 而主存碎片也同样被保留下来. 如果系统缺乏一种有效的主存碎片处理机制, 主存碎片会持续积累, 严重降低非易失主存的空间利用率.

为了减少碎片, PMDK [12] 针对不同大小的主存分配操作采用了不同的分配策略. 对于小于 256 KB 的分配操作, 它采用分离适配策略: 使用 35 种不同尺寸的分配类, 将每个 256 KB 的超级块切割成多个更小的尺寸为 8 字节倍数的主存块, 满足一定区间的主存分配操作. 虽然细粒度的分离适配策略在一定程度上减少了小于 256 KB 的分配操作所产生的主存碎片, 但是大于 256 KB 的分配操作依然易

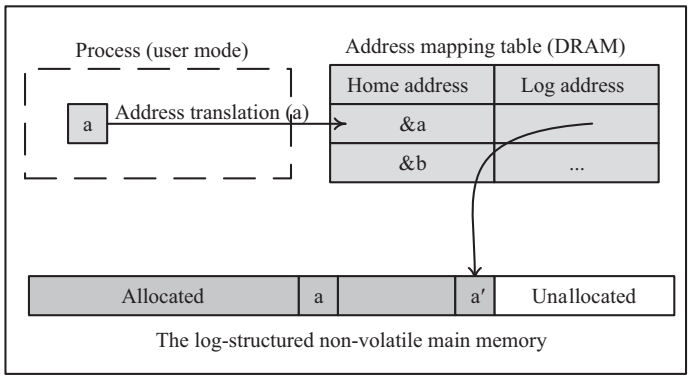


图 6　基于日志结构的非易失主存

Figure 6　The log-structured non-volatile main memory

于引入较高的主存碎片.

为了解决这个问题, PAllocator [23] 设计了去碎片化机制: 当主存分配器的连续空间无法响应某个分配操作时, 它首先定位到目前最大的空闲区域, 然后通过文件系统的 fallocate 函数合并非易失主存的空洞区域, 从而减少主存碎片. 然而, PAllocator 只能消除页粒度的主存碎片, 无法消除更细粒度的主存碎片.

为了消除更细粒度的主存碎片, 清华大学提出了 LSNVMM [13]. 如图 6 所示, LSNVMM 将整个非易失主存组织成一个日志结构. 对于所有分配操作, 它将新数据直接添加到日志末尾, 而不是将主存超级块切割成固定大小的主存块, 从而消除了大部分的内部碎片. 此外, 它通过将合法数据进行迁移, 回收未被使用的主存空间, 将其组织成更大的连续区域, 达到了消除外部碎片的目的. 并且, LSNVMM 的碎片清理过程不需要中断整个系统的正常运行, 对整个系统的性能影响较低.

3.4　小结

目前主流的持久性堆结构借助于文件系统, 有效地解决了命名系统和权限控制的问题. 其次, 研究人员通过修改操作系统的虚拟主存管理, 实现了非易失主存作为易失性主存的自动容量扩展功能. 再次, 研究人员通过减少需要保证一致性的关键元数据, 在保证可接受的系统恢复延迟前提下, 降低了分配/释放操作的一致性开销. 最后, 研究人员通过细粒度的主存块切割策略或者基于日志结构的非易失主存组织结构, 提高了主存的空间利用率.

4　非易失主存的编程模型

非易失内存提供了单层的数据存储架构, 应用程序可以直接在内存级实现数据的持久性存储, 这种新的存储架构避免了传统存储系统中将内存格式的数据序列化存储到外存的过程. 因此, 系统软件需要提供新的编程模型, 帮助程序员或开发者像管理易失性内存一样轻松地管理非易失内存, 同时提供全面的功能支持.

4.1　编程接口设计

在传统存储系统中, 易失性主存中的数据需要转换成序列化的格式后才能写回到外存中. 而在字节寻址的非易失主存中, 数据可以直接持久化在主存中, 无需进行格式转换. 非易失主存改变了持久

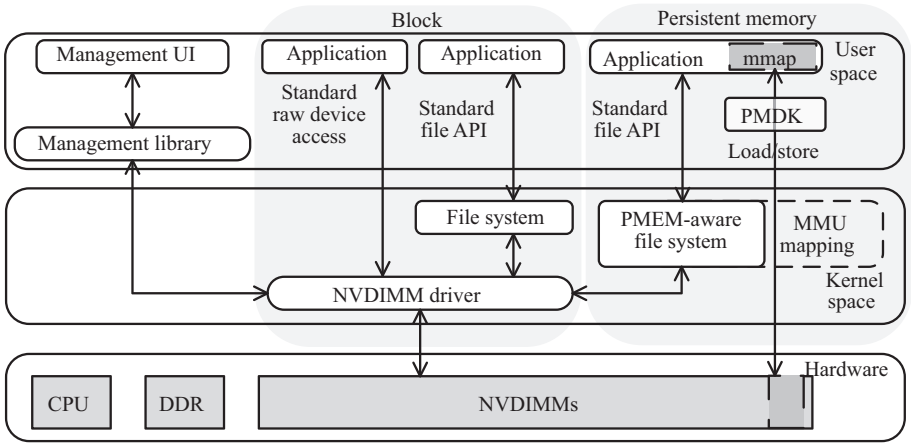


图 7 SINA 编程模型概述

Figure 7 SINA programming model

性数据的存储层次, 上层应用以访问主存的方式访问非易失主存上的持久性数据. 当数据从处理器缓存写回到非易失主存上时, 数据就已经持久化. 然而, 处理器缓存是硬件控制的, 它会打乱数据写回到非易失主存的顺序, 破坏了数据的可恢复性. 因此, 非易失主存的编程模型需要设计简单通用的编程接口, 提供应用程序自管理的持久化功能, 保证持久性数据的一致性.

图 7 展示了几种可能的持久性内存设备访问模式. 更具体的描述可参考文献 [30]. Linux 4.3 及更高的版本已经默认包含了 NVDIMM 驱动, 持久性内存将以一种设备的形式出现在/dev/pmem*. 应用程序可以直接打开该设备, 然后进行裸设备读写形式访问持久性内存空间; 当然, 程序员亦可在该设备上部署传统的文件系统, 然后通过文件形式访问持久性内存空间. 值得注意的是, 裸设备访问没有任何一致性保障, 其存储的数据有丢失、不一致的风险, 而通过传统文件系统访问则要忍受额外的软件开销, 性能损耗比较严重. 目前, 更为流行的方法是通过专用的持久性内存文件系统管理持久性内存空间, 或通过新型的编程模型库对其进行管理.

研究者们近年来设计了新的编程模型, 例如 PMDK [12], NVM DIRECT [26], Mnemosyne [11] 和 NV-Heaps [20] 等. PMDK 是目前较为流行的非易失主存编程模型, 它在用户态提供了较为全面的功能支持, 为上层程序员和开发者提供了工业界认可的编程接口. 直接访问持久性内存将引入新的编程挑战, PMDK 正是以此为出发点, 为开发人员提供了libpmem, libpmemobj, libpmemblk, libpmemlog, libvmmalloc, libpmempool, librmem 等功能库, 以解决一些实际的编程问题. PMDK 支持现有的 CPU 硬件指令和现有的 C/C++ 语言, 不需要额外的硬件指令和编程语言支持.

Algorithm 1 PMDK's transactional interfaces

```
1: TX_BEGIN(pop) {
2:     entry ⇐ TX_NEW(struct hash_entry);
3:     D_RW(entry) → key ⇐ key;
4:     D_RW(entry) → value ⇐ value;
5: } TX_END.
```

如算法 1 所示, PMDK 还提供了基于 undo 日志的事务接口, 上层应用可以高效稳健地完成非易失主存上的更新操作. 程序员通过 TX_NEW 申请一块主存区域, 并通过 D_RW 接口找到这个指针对应的非易失主存地址. 通过基于 undo 日志的事务机制, PMDK 能保证 TX_BEGIN 和 TX_END 之间

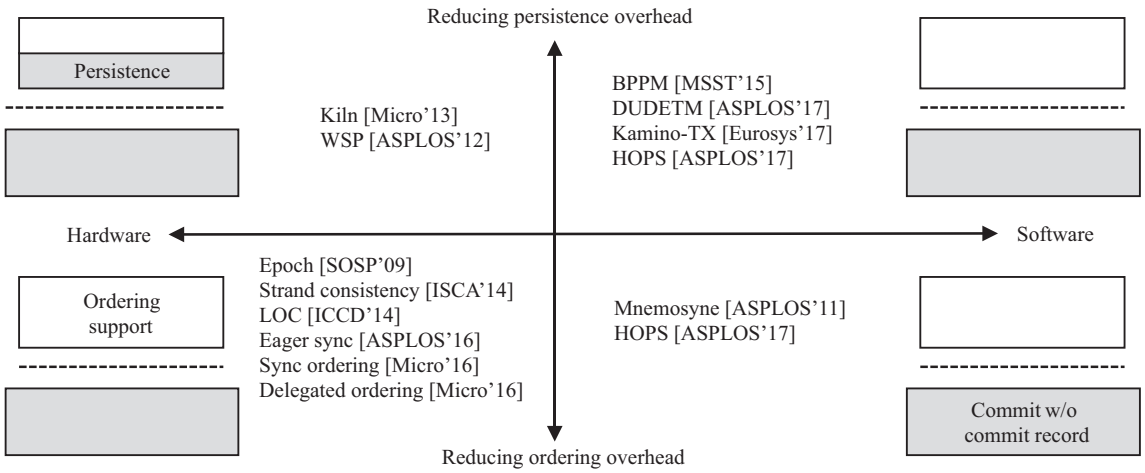


图 8　一致性优化机制分类与对比

Figure 8　The comparison of different consistency mechanism

的代码的原子性和一致性.

Intel 已经在 2019 年 4 月正式发布其基于 3D-XPoint 技术的持久性内存设备 Optane DC Persistent Memory. 目前, 其单条容量分别达到 128, 256 和 512 GB. Optane 持久性内存可以工作在内存模式 (memory mode) 和应用直访模式 (APP-direct mode) 两种模式下. 其中, 当配置为内存模式时, 应用程序和操作系统将其构建为易失性内存池, 这与普通的 DRAM 使用模式的情况完全相同. 在此模式下, DRAM 充当热点访问数据的缓存, 而 Optane DC 持久性内存则提供大容量空间, 应用程序中不需要特定的编程手段, 但是在系统断电时不会持久性地存储数据. 该模式适用于对内存容量需求很大, 但无需将数据立即持久化的一些应用程序, 例如内存计算框架 Spark 等. 在应用直访模式下, 应用程序和操作系统明确知道平台中有两种类型的内存, 并且可以指示对哪种类型的内存进行数据读取. 内存数据库等需要及时持久化数据的应用适用于使用该模式. 为更好地管理持久性内存设备, Intel 还提供了 ipmctl 和 ndctl 两个系统工具, 用于灵活地切换持久性内存的工作模式以及管理名字空间.

4.2　编程模型的一致性优化机制

编程模型需要支持系统在异常掉电或失效等故障出现后将保存在 NVM 的持久性数据恢复到一致的状态. 为了达到这个目的, 现有的系统往往采用 redo 或者 undo 日志的策略, 即在修改某个数据之前先完成这个数据备份. 然而, 日志机制会引入额外的持久化开销, 此外, 为了避免缓存的乱序执行导致的不一致问题, 编程模型往往需要调用 clflush 等处理器指令进行强制缓存行刷新, 但它们会带来昂贵的开销. 因此, 编程模型需要设计低开销的一致性优化机制. 在一致性的优化机制上, 目前分别有从软件、硬件的角度出发, 分别考虑顺序性和持久性的优化方法, 总结如图 8 所示.

(1) 顺序性方面的优化机制. 顺序性指的是处理器数据需要根据数据依赖关系有序地持久化到非易失主存. 现有的编程模型依赖 clflush 操作保证持久化操作的顺序性. 然而, 多个 clflush 操作之间存在强依赖关系, 后一个 clflush 操作必须等待前一个 clflush 操作执行完成后才能接着执行, 这就产生了昂贵的堵塞开销.

(2) 持久性方面的优化机制. 持久性指的是处理器数据从多级易失性处理器缓存替换到非易失主存, 其中可能存在冗余的持久化开销. 例如, 由于非易失主存只支持 8 字节原子写操作 [11], 任何大于

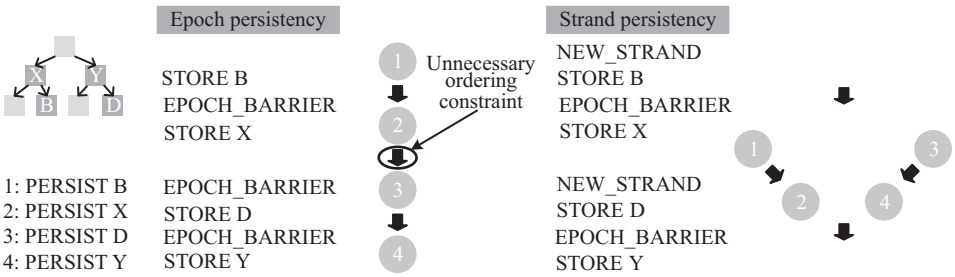


图 9 Epoch 和 Strand 的对比

Figure 9 The comparison of Epoch and Strand

8 字节的写操作都可能由于系统崩溃或者断电处于不一致的中间状态. 所以现有系统往往使用日志机制保证持久化操作的原子性, 引入了额外的日志持久化开销.

4.2.1 顺序性方面的优化机制

为了降低顺序性开销对性能的影响, 大量研究工作通过在处理器缓存中以硬件的方式提供顺序性的支持, 从而大幅降低软件显式顺序性的开销. Microsoft 研究院提出增加新的硬件指令: epoch 指令 [31]. 该指令要求前一个 epoch 的数据必须在后一个 epoch 到达之前持久化完成. 程序员通过 epoch 指令将程序划分成多个执行单元, 这种机制利用硬件保证不同的执行单元之间遵循持久化顺序约束, 而每个执行单元内可以通过对写操作重排序来提高性能, 从而通过硬件的顺序化命令降低软件顺序化的开销. 与此类似, CLC [32] 同样在处理器缓存硬件中提供顺序性, 同时提供给程序状态查询的接口.

虽然 epoch 指令支持每个执行单元中持久化操作的异步提交, 但每个执行单元之间依然具有很强的顺序依赖关系, 因此性能的提升效果较为有限. 清华大学提出了放松一致性机制 LOC [33]. LOC 将预测执行技术引入到处理器缓存的数据持久化管理中. LOC 允许持久化的数据以重排序的方式刷回非易失主存, 以降低顺序化引发的带宽浪费. LOC 通过对数据日志的组织以及计数式提交协议等方式提供预测失败后的事务回退. 通过上述预测持久化技术, LOC 显著降低了顺序化的开销. PTM [34] 也采用了类似方式扩展 CPU 缓存, 以提供一致性保证.

然而, 持久化指令依然使用了一种严格的持久性模型, 它将持久化操作间的依赖关系强制等同于写操作间的依赖关系. 这导致原本没有依赖关系的持久化操作 (例如不同线程针对不同地址的持久化操作所产生的 epoch 指令) 依然需要串行执行, 带来过高的顺序性开销.

为了解决这个问题, University of Michigan 提出了 Strand persistency 机制 [35], 设计了放松的持久性模型. 如图 9 所示, 该模型基于程序语义将 I/O 依赖关系分成多个并行线程, 而并行线程之间无 I/O 依赖关系可并行执行. Strand persistency 通过并行方式降低依赖顺序性的开销. 此外, Intel 公司于 2014 年设计了新的扩展指令 [15], 通过 clwb 指令既避免持久化指令之间的依赖关系, 又避免写回的缓存行数据失效, 从而能够供后续访问继续使用, 减少缓存缺失操作带来的性能影响. 当上层应用需要保证持久化操作的顺序时, 它们可以通过内存屏障指令 (例如 mfence) 控制持久化操作的顺序. 此外, Intel 公司还提出废弃 PCOMMIT 指令, 将其使命交由支持异步 DRAM 刷新 (asynchronous DRAM refresh, ADR) 的平台完成, 进一步降低了编程模型中所需要的持久化开销.

University of Michigan 基于 epoch, Strand 和 clwb 这 3 种指令构建了延迟提交事务机制 DCT [36]. 传统事务机制需要在事务提交后才能释放更新数据的锁, 而 DCT 尝试在修改数据后就释放锁, 从而降低对其他冲突事务在准备 undo 日志项时的阻塞延迟. 前面的工作主要是在处理器高速缓存层次控

制顺序约束关系. University of Michigan 随后提出了委托顺序化机制 (delegated ordering), 显示地将偏序顺序约束暴露给持久性主存控制器 (PM controller) [37]. 持久性主存控制器管理和控制持久性主存读写操作的时序, 对访存物理地址分配到具体哪一个存储体 (bank) 具有充分的认识, 从而可以高效发挥内存调度过程的灵活性, 有效提高非易失主存的并发访问能力.

除了硬件方面的优化策略, University of Wisconsin-Madison 还通过软件途径为非易失主存设计了持久性主存系统 Mnemosyne [21]. 它提出了两种降低顺序性开销的软件方法: Torn-bit 方法和异步检查点方法. Torn-bit 在每个 64 比特数据块中利用一个比特位标识数据是否写入. 具体做法在每次写入之前将所分配空间的 Torn-bit 的比特位清除. 这样在恢复时通过该比特位即可判断数据是否已完成持久化. 异步检查点方法通过后台的方式异步地将检查点数据持久化至数据原有位置. 在事务空闲时间较多时, 异步检查点方法可以充分利用后台时间, 而不影响程序的正常执行. 通过 Torn-bit 方法和异步检查点方法, 程序的顺序性开销得以降低.

4.2.2 持久性方面的优化机制

研究人员同样通过硬件和软件两个角度降低持久性方面的开销. 从硬件角度, 数据的持久化需要从处理器缓存的不同层级 (如 L1, L2 等) 刷回到持久性主存中. 因而, 在其中的部分或所有层次采用非易失性存储器可以缩短持久化路径, 降低持久化开销. Microsoft 研究院提出了全系统持久化技术 (whole system persistence, WSP) [38], 它将所有处理器缓存均采用了非易失存储器, 并采用后备电源方式保证了在系统掉电后总线上的数据传输. University of Pennsylvania 提出了 Kiln [39], 它仅在处理器末级缓存上使用了非易失性存储器, 通过在末级缓存上提供数据新版本的持久化, 降低事务中持久化的开销. 这些做法需要在处理器的缓存层次中采用非易失性存储器, 需要对处理器硬件进行改造.

由于部分非易失性内存 (例如 PCM) 写入操作速度与数据保持力 (retention) 和可靠性呈反比关系. 因此, 在数据保持力和可靠性要求可以放松的场景下, 可以采用快速的写入策略. 利用该特性, 清华大学设计了 DP2 [40], 它针对日志和数据分别采用了不同的写入策略. 由于日志部分的保持力要求较短, 因而可以在牺牲部分数据保持力的前提下采用快速写入策略, 提高日志写操作的性能. 这种做法通过写入速度调节可降低持久化开销, 但仅适用于特定的非易失性主存器件.

在软件方面, PMDK [12] 使用了基于 undo 日志的事务机制: 任何一个更新操作都需要先将旧数据持久化到日志中, 然后才能修改旧数据. 然而, undo 日志会导致每一次更新操作都引发昂贵的持久化指令和额外的写操作, 带来了较高的持久性开销. Mnemosyne [11] 提供了一个基于 redo 日志的轻量级持久性事务接口, 它只需要在事务提交阶段将新修改的数据持久化到日志中, 减少了持久化操作的开销. 然而, 基于 redo/undo 日志的事务机制仍然需要先将数据持久化到日志中, 然后再将数据持久化到原数据区, 这导致后者的持久性开销出现在程序执行的关键路径上.

针对这个问题, 清华大学设计了 BPPM [10] (如图 10 所示). 因为日志已经保证了事务提交数据的持久性, 所以将数据从日志区写回到原数据区的过程中, 数据不需要立即持久化. 只有当日志空间不足时, 它才将缓存中的数据持久化到非易失主存中, 从而减少了持久化操作带来的延迟. 此外, 它重新组织了日志结构, 可以发现日志中尚未提交的数据, 从而允许事务在执行过程中将尚未提交的数据直接持久化到日志中, 避免了在 CPU 缓存中维护额外数据版本所带来的开销.

之前的事务机制为了保证一致性, 总是需要先将数据额外拷贝一份放在日志中, 避免持久性数据在更新过程中因为系统崩溃等原因而不可恢复. 然而, 数据拷贝过程往往发生事务的关键路径上, 产生了较高的拷贝延迟. University of California, San Diego 提出了 Kamino-TX [41], 它维护了数据的额外副本, 新数据可以直接更新在原数据区, 避免关键路径上的日志拷贝延迟. 此外, 它通过对象粒度的

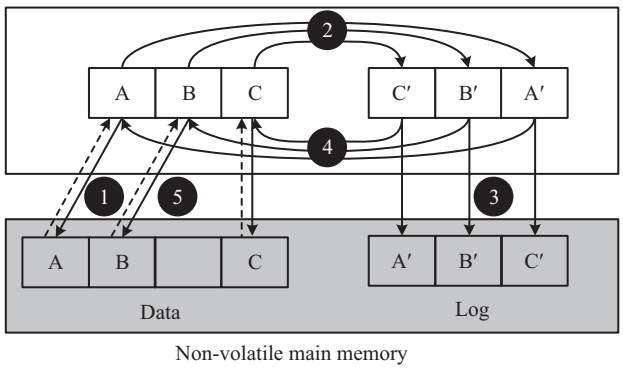


图 10 BPPM 架构图 [10]

Figure 10 The architecture of BPPM [10]

锁机制实现了异步的数据拷贝, 只有将已更新对象同步到数据副本后才释放该对象的锁, 从而避免了后续事务对同一对象的修改操作而导致的系统不一致性问题. 最后, 它通过只维护频繁访问的对象的数据副本, 有效地降低了数据副本的存储开销.

然而, Kamino-TX 依然无法避免写放大问题带来的能耗和耐久性问题. 清华大学提出了基于日志结构的持久性事务主存系统 LSNVMM [13], 它将整个非易失主存组织成一个日志结构. 对于所有分配操作, 它将新数据直接添加到日志末尾, 然后修改位于易失性主存中的地址映射表, 无需再将新数据写回到原数据区, 从而降低了事务的持久性开销.

4.3 编程安全性保障机制

NV-heaps [20] 是 University of California, San Diego 开发的轻量级持久性对象系统. 它设计了一套灵活健壮的编程模型, 以消除非易失主存上编程产生的错误. 首先, 它定义了指针错误, 并通过指针类型检查避免危险的指针问题. 其次, 它利用基于引用计数的引用完整性检测, 实现了自动垃圾回收机制, 避免内存泄漏等错误. NV-heaps 的设计思想影响了后续众多编程模型的设计, 例如 PMDK [12], NVM DIRECT [26] 等. 虽然 NV-heaps 的设计思想有效地避免了非易失主存编程过程中易于产生的编程错误, 但是较为复杂的编程接口带来了糟糕的编程体验, 增加了程序员的编程负担.

为了解决这个问题, 橡树岭国家实验室设计了基于 LLVM 编译器的静态分析技术 NVL-C [42]. 它支持与 C 语言相似的编程接口, 并且能自动分析持久性指针引入的程序错误. 它还利用引用计数自动避免了内存泄露问题. 此外, 它在编译过程中自动识别无需事务机制保护的变量, 从而减小了事务开销.

4.4 小结

目前主流的非易失主存编程模型提供了简单通用的事务接口, 以保证应用在更新持久性数据时数据的一致性和持久性. 其次, 研究人员通过软硬件协同的方式, 有效地降低了一致性机制在持久性和顺序性方面的开销. 最后, 研究人员通过合理的编程模型和编译器技术, 提高了在非易失主存上编程的安全性.

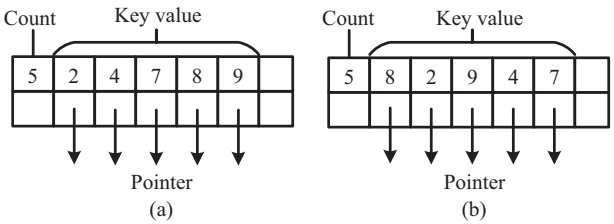


图 11　无序的树节点设计 [44]

Figure 11　The unsorted tree node design [44]. (a) Sorted; (b) unsorted.

5　非易失主存的数据结构

单个键值的查询/插入/更新/删除操作和多个键值的范围查找操作都是存储系统中频繁使用的操作. B+ 树 [43] 同时支持单值和范围查找操作, 是基于外存的存储系统中广泛使用的有序索引结构, 因此, 近年来很多研究工作针对 B+ 树的特性设计了大量优化机制. 然而, B+ 树的排序和平衡操作可能在非易失主存上触发昂贵的持久化开销. 因此, 研究人员在部分场景下也针对非易失主存的特性, 对其他索引结构 (如 Hash 表等) 进行优化. 其中, Hash 表执行范围操作时需要扫描整个索引结构, 这将导致十分糟糕的性能, 但是它在单键操作场景下取得了良好的性能. 下文将分别介绍 B+ 树和其他索引结构的优化机制. 非易失主存上的数据结构设计需要考虑以下几个问题:

(1) 读写不对称优化机制. 非易失主存的写延迟远高于读延迟, 并且存在耐久性方面的问题. 传统 DRAM 主存不存在读写不对称的问题, 因此数据结构存在大量的写操作, 例如标准 B+ 树中存在频繁的排序写操作. 这些频繁的写操作在 NVM 上带来严重的性能问题和磨损问题. 所以如何设计优化非易失主存上数据结构的更新操作, 降低写开销是一个重要问题.

(2) 一致性优化机制. 传统内存 DRAM 不存在持久性的问题, 但是非易失主存的所有更新都会被持久化下来, 如果突然断电或者系统崩溃, 就可能将不一致的状态保留在非易失主存中, 所以非易失主存上的索引结构需要提供一致性的保证. 虽然上述研究减少了 B+ 树的写操作, 但是它们无法保证 B+ 树在发生系统错误时数据的一致性. 虽然持久性事务主存系统为应用程序提供了一个通用性的一致性编程接口, 但是基于日志的事务机制会引发额外的持久化开销. 如何设计优化非易失主存上数据结构的一致性保证机制, 成为一个十分重要的研究问题.

5.1　读写不对称优化机制

Intel 和 Microsoft 公司提出了一种 PCM 友好的 B+ 树 [44]. 他们发现因为 PCM 读写不对称性的特点, B+ 树的插入/删除操作引发的排序操作会带来昂贵的写开销, 导致性能远低于查询操作. 因此, 如图 11 所示, 他们使用无序的树节点结构, 避免了插入和删除操作中排序操作引入的持久化开销, 但是也增加了查询操作的延迟.

University of Pennsylvania 发现无序的树节点在分裂时会引发昂贵的排序操作 [45]. 为了解决这个问题, 他们提出了部分平衡的无序节点模式 (sub-balanced unsorted node scheme): 分裂操作以 $O(n)$ 的时间复杂度找到中间那个键值对, 然后分别将小于它的键值对和大于它的键值对移动到不同的分裂节点, 从而避免了分裂操作带来的排序开销. 另外, 他们推迟了分裂操作对父节点的更新, 将多个父节点的更新操作进行聚集后统一处理.

Path Hash [46] 是华中科技大学针对非易失主存设计的新型持久性 Hash 表. 他们发现现有的 Hash 冲突的处理制在非易失主存中会产生大量写操作. 因此, 它设计了路径 Hash, 通过位置共享技术 (po-

sition sharing) 避免了大量的 Hash 冲突, 并且不会产生额外的写操作. 此外, 它通过双路径 Hash 和路径放缩技术优化了 Hash 表的空间利用率和操作延迟.

5.2 数据结构的一致性优化机制

针对 B+ 树的特点, 研究人员通过系统提供的持久化原语, 设计了更加精细的一致性更新策略. CDDS-Tree [47] 是惠普实验室针对非易失主存设计的一致性 B+ 树. 它为每个数据项分配了一个版本号区间. 更新操作为每个更新的数据项生成一个新的版本, 并将其插入到树节点的合适位置. 针对删除操作, CDDS-Tree 通过版本号的设置便可轻松完成, 且整个过程不影响旧数据项. 在适当的时机, CDDS-Tree 才会回收这些旧数据项, 从而保证它在发生系统错误时能找到正确的数据版本. 然而, 基于版本号的一致性更新机制会引发严重的写放大问题. 因此, 后续很多工作尝试从不同方面减少 B+ 树的一致性开销.

中国科学院设计了 wB+Tree [48]. 它同样采用无序的树节点, 避免了排序操作带来的一致性开销. 对于普通的插入/更新/删除操作, 它利用 bitmap 来保证它们的一致性. 对于复杂的平衡操作, 它利用日志机制来保证索引结构在更新过程中一直有一个正确的数据版本. 另外, wB+Tree 还设计了间接排序的 slot 数组, 支持在无序的树节点上执行二分查找操作. 但是, 维护日志和 bitmap/slot 数组会引入额外的持久化开销.

NV-Tree [49] 是 Nanyang Technological University 设计的保证局部一致性的 B+ 树. NV-Tree 只保证叶节点的一致性, 这是因为内部节点只是用于加速叶节点的查找性能, 它在系统崩溃时完全可以基于叶节点进行重构. 所以, 该方法降低了整个索引结构的一致性开销. 它同样采用了无序的叶节点, 但是叶节点的查询操作需要线性扫描整个节点的所有键值对. 最后, 它将所有内部节点维护在一个连续的主存区域, 提高了 NV-Tree 的空间局部性. 然而, 当内部节点数量溢出时, 这会引入额外的重建开销.

FPTree [29] 是 Technische Universität Dresden 针对混合主存设计的 B+ 树. 它将叶子结点存放在 NVM 上, 内部节点存放在 DRAM 上, 避免了内部节点的持久化开销. 此外, 它为叶节点上每个无序的键值对配置了一个单字节的 Hash 项. 在查找过程中, 只有目标值的 Hash 值与键值对的 Hash 值相同时, 它才会访问对应的键值对, 从而降低了叶子结点的顺序查找开销.

FAST+FAIR [50] 是蔚山国立科技大学设计的可容忍临时不一致性的 B+ 树. 它通过控制更新操作的持久化顺序, 实现了即使在系统崩溃后也能确保树节点也处于一种可识别的不一致性状态, 并通过这一技术避免了排序和平衡操作的日志开销. 此外, 因为写操作导致的树节点不一致状态是可以识别的, 所以它还设计了 lock-free 的查找操作, 避免了查找操作的锁开销.

WORT [51] 是蔚山国立科技大学设计的持久性基数树. 它是一种确定性的树结构, 不会引发排序和平衡操作的持久性开销. 它的所有操作都是原子性的, 不会因为日志机制而产生过高的持久性开销. 此外, 它通过保证一致性的路径压缩技术, 提升了基数树的空间利用率.

5.3 其他优化机制

Hash 表动态扩容. 将 Hash 表应用于非易失性内存面临的另一个挑战是动态调整大小时开销很大. 随着负载因子的增大, 为了保证访问性能和减少冲突, Hash 表需要调整大小. 传统的方法一般需要分配一个大小为旧表两倍的空间的新 Hash 表, 然后把所有的键值对都重新 Hash 到新表中. 这会造成大量的写入操作, 影响性能并增加非易失性内存的磨损. Level-Hashing [52] 是华中科技大学针对非易失主存设计的可动态扩容的持久性 Hash 表. 该系统巧妙地提出了一种分层的 Hash 表方案, 从而

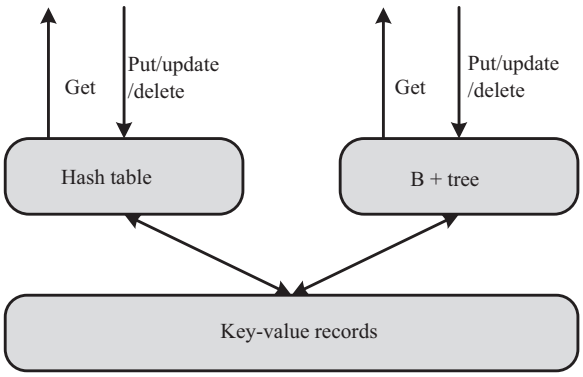


图 12 HiKV 结构 [53]

Figure 12 The architecture of HiKV [53]

实现了原地扩容, 大大降低了额外的写入操作.

索引效率. HiKV [53] 是中国科学院设计的一种混合索引结构. 如图 12 所示, 它将 B+ 树存储在 DRAM 上, Hash 表存储在 NVM 上. 它利用持久性 Hash 表支持索引结构的单键操作, 从而显著降低了持久性 B+ 树的写开销. 因为 Hash 表是无序的, 所以 HiKV 利用易失性 B+ 树支持范围操作. 然而在执行范围操作前, 它需要堵塞后续的更新操作直到 B+ 树与 Hash 表具有相同的状态. 因此, 在那些具有较多更新操作和范围操作的工作负载中, HiKV 的性能可能会有所下降.

5.4 小结

因为 B+ 树有效地支持单值和范围操作, 所以它被广泛使用在传统存储系统. 研究人员针对非易失主存的特性, 通过避免排序操作, 有效地减少了传统 B+ 树的持久化开销. 此外, 它们通过选择一致性或者控制持久化的顺序等方式, 降低了一致性开销. 因为 B+ 树的排序和平衡操作会引入较高的持久化开销, 研究人员在部分场景下也针对其他索引结构做了大量优化.

6 非易失主存的文件系统

文件系统是操作系统中最基础的模块, 它将设备存储空间以文件的形式组织为可索引的文件目录树, 从而方便用户存取数据. 为兼容现有的应用程序, 将非易失内存组织成文件系统是十分便捷的途径.

一种简单的方法是直接使用现有的外存文件系统管理非易失内存空间. 例如, 通过 RamDisk 将持久性内存模拟成块设备, 从而兼容现有的外存文件系统 (如 Ext4, XFS, BtrFS 等). 通过这种方法, 传统文件系统无需作出任何修改, 可直接构建在非易失内存模拟的 RamDisk 块设备之上. RamDisk 形式的传统文件系统构建方案能够快速收益于内存级的数据持久化, 相比于外存, 性能有数量级的提升. 但是, 其缺陷是软件层次开销大, 无法充分利用非易失内存的优势. 具体的原因包括以下几个方面.

(1) 操作系统层次软件开销. 操作系统对不同文件系统进行统一抽象, 以屏蔽它们在实现上的差异性, 从而形成了具有统一接口的虚拟文件系统. 由于传统外存的延迟极高, 带宽有限, VFS 管理了一部分内存空间, 形成位于文件系统之上的缓存系统, 其将频繁访问的文件数据块、元数据块等直接放入内存, 以提升性能. 然而, 非易失内存的访问性能与 DRAM 非常接近, 因此, 在基于非易失内存的文件系统中, DRAM 缓存将不再高效. 同时, VFS 本身软件逻辑复杂, 软件开销高. 另外, 用户态应用程

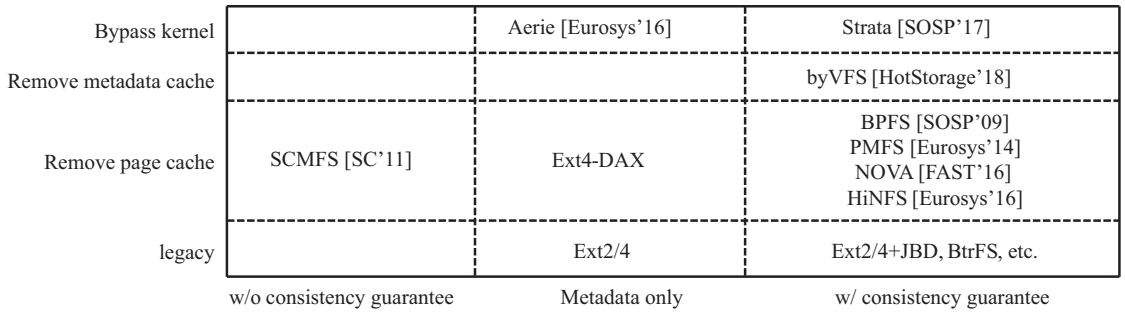


图 13 非易失内存文件系统分类与对比

Figure 13 The comparison of different persistent memory file systems

序依靠系统调用访问内核文件系统, 此过程将引起频繁的现场切换和缓存替换, 同样也会影响性能.

(2) 非易失内存字节可寻址特性无法充分发挥. 外存存储设备 (如磁盘, 固态硬盘等) 均为块设备, "块" 为最小访问粒度, 因此, 传统文件系统均按照块粒度进行数据摆放. 如果在非易失内存上继续使用类似的数据摆放策略, 将引入两个方面的挑战: (a) 数据写放大, 例如, 文件 inode 通常为几百个字节, 且大多数元数据修改只修改其中的几个字节, 块粒度的修改将引入 KB 级别的写放大效果. 文件数据的更新同样面临类似的问题. (b) 一致性管理更加复杂, 块设备能够保障 "块" 粒度的原子性更新, 而持久性内存只能保证 8 字节粒度的原子性更新. 因此, 如何充分利用非易失内存的字节可寻址特性设计新的数据布局方式和一致性管理策略, 是值得深入探讨的问题.

目前已有大量工作提出了不同的文件系统架构和优化机制, 以应对上述挑战. 图 13 从两个维度分析和对比了相关工作, 下文将逐类介绍.

6.1 内核文件系统

目前, 非易失内存文件系统的相关研究大多在内核态实现. 将文件系统实现在内核态, 可以继续保留传统系统调用的访问模式, 以兼容现有的应用程序. 这些文件系统主要集中在设计新的一致性保障机制、消除内核缓存、提升多核扩展性等方面.

6.1.1 新型一致性保障机制

在持久性内存中, NVM 通过内存总线接入处理器, 存储访问延迟极低. 与外存块粒度访问不同, 持久性内存可由处理器以字节粒度访问. 这些差异促使了设计新型一致性管理方案.

Microsoft 研究院于 2009 年研究提出字节寻址的持久性内存文件系统 BPFS [54], 旨在利用 NVM 的字节寻址特性实现文件数据的原子性更新. BPFS 引入树状结构在持久性内存上构建文件系统数据结构, 并通过写时复制机制 (copy-on-write) 实现数据的原子性更新; 不同于传统的写时复制方法, BPFS 充分利用了 NVM 字节可寻址的特性, 以短路影子页 (short-circuit shadow paging) 方式提供数据的原子更新 (如图 14 所示), 从而减少了传统技术带来的级联更新整棵文件树的多余开销; 同时, BPFS 还提出一种 Epoch 提交策略将顺序性与持久性解耦合, 以缓解刷新缓存的开销.

Intel 公司于 2014 年研究提出了 NVM 直写的 PMFS 文件系统 [55]. PMFS 是面向持久性内存提供 POSIX 接口的文件系统, 因而兼容传统应用程序. PMFS 通过不同的方法分别维护数据和元数据的崩溃一致性: NVM 能够保证 8 字节数据的原子性更新, 因此, PMFS 采用原子的 in-place 更新和细粒度日志机制保证元数据更新的原子性. 文件数据的更新粒度更大, PMFS 则采用 undo 日志和写入时复制 (copy-on-write) 混合方式保证数据的一致性.

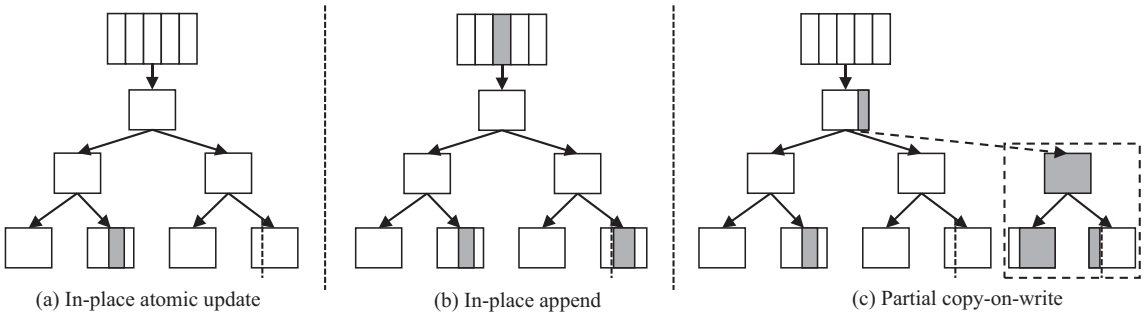


图 14　BPFS 的短回路影子页技术

Figure 14　The short-circuit shadow paging mechanism in BPFS

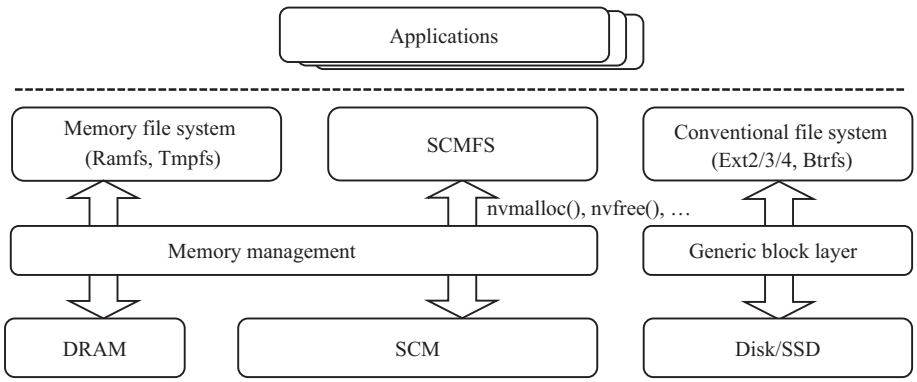


图 15　SCMFS 系统架构

Figure 15　The architecture of SCMFS

University of California, San Diego 将传统日志结构文件系统 (LFS) 进行重新设计和扩展, 应用在持久性内存之上研制出日志结构持久性文件系统 NOVA [56]. NOVA 同时提供了元数据、数据、内存映射 (mmap) 操作的原子性: NOVA 将每个文件 inode 组织为一个日志, 因此, 其对单个 inode 的修改可以直接通过在日志尾部追加更新数据即可, 原子性可以很轻易地被保证; 重命名等操作通常涉及对多个日志记录的更新, 因此, NOVA 还采用了轻量级的日志技术 (journaling) 以保证这些操作的原子性; 与 PMFS 相同, NOVA 对文件数据使用写时复制技术.

6.1.2　移除页缓存

操作系统管理的页缓存在非易失内存中将造成冗余的数据拷贝, 严重影响性能. 针对这个问题, Ext4, BtrFS 等传统文件系统均兼容了直接访问模式 (DAX). 通过这种方法, 应用程序可以直接访问非易失内存中存储的文件数据, 而不需要将数据额外拷贝到页缓存中. 针对持久性内存进行重新设计的 PMFS, NOVA, BPFS 等文件系统则通过内存映射的方式绕开了文件系统页缓存, 从而避免在持久性内存中的数据拷贝.

此外, Texas A&M University 于 2011 年研究提出内外存融合管理的 SCMFS 文件系统 [57], 旨在利用操作系统中现有的内存管理模块提供持久性内存的数据分配与管理. SCMFS 文件系统通过页表映射方式, 使得文件系统中的文件具有连续的地址空间 (如图 15 所示). 通过这种数据组织的优化, 应用程序可以进行连续的数据块访问, 提供了优越的性能.

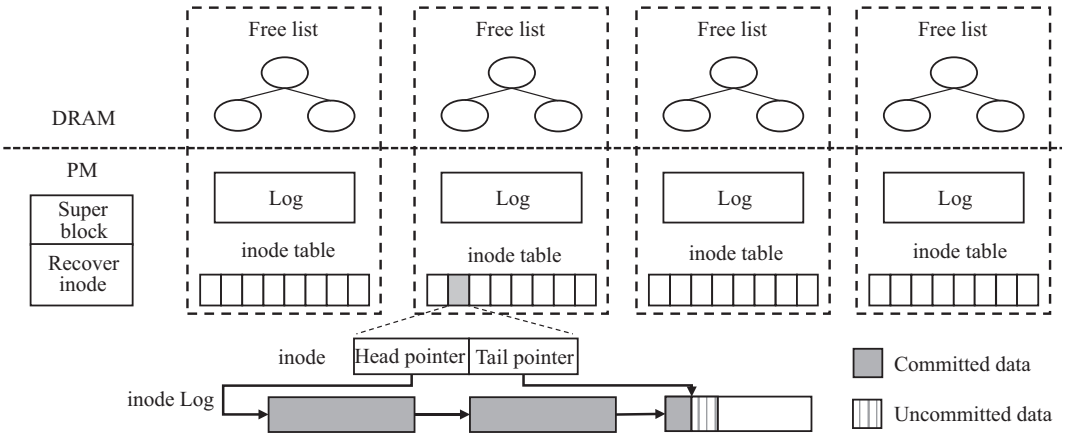


图 16 NOVA 系统架构

Figure 16 The architecture of NOVA

6.1.3 移除元数据缓存

VFS 除了管理数据页缓存, 还会将频繁访问的元数据放在 DRAM, 以提升在传统外存设备下的访问性能. 然而, 中科院计算所的研究人员发现, 基于持久性内存的文件系统在 VFS 层次的执行时间占比超过 50%, 而这些时间主要花费在元数据的路径查询上. 基于上述发现, 他们设计了 byVFS 文件系统 [58], 并提出元数据直接访问模式: VFS 不再缓存文件系统元数据, 而是直接在物理的文件系统镜像上进行元数据索引与更新. 通过这种方法, byVFS 能够将路径查询性能提升 7.1% 到 47.9%, 应用程序执行时间缩短 26.9%.

6.1.4 多核扩展性

在后摩尔时代, 多核架构是处理器未来发展的方向. 在多核场景下, 如何设计可扩展的文件系统以充分利用持久性内存的高吞吐特性变得至关重要. 目前, 多数持久性内存文件系统均未考虑文件系统的扩展性需求, 并使用了大量的全局数据结构, 严重影响了系统扩展性.

NOVA [56] 在改造传统日志式文件系统时, 重点考虑了其多核扩展性. 如图 16 所示, 不同于传统日志式文件系统的单日志结构, NOVA 具有以下特点: (1) 将每个 inode 组织为一个日志, 从而避免了并发向单个日志追加记录时难扩展的问题; (2) 日志中仅记录元数据, 而将文件数据单独通过写时复制机制管理, 日志变得非常轻量, 有效降低了垃圾回收的额外开销. 数据页以 4 KB 粒度进行管理, 废旧的数据页在释放时均能够及时被回收; (3) 将空闲空间切分给不同核心, 线程在申请空闲数据页时仅向本地空闲空间申请, 因此核心之间不会发生竞争. 通过上述设计, NOVA 具有极强的扩展性, 相比于 PMFS 等文件系统, NOVA 能够提供更加优越的访问性能.

6.2 用户态文件系统

如前文所述, VFS 所管理的页缓存、元数据缓存限制了非易失内存的性能. 基于非易失内存专门设计的文件系统均不同程度解决了这些问题. 然而, VFS 还存在复杂的软件执行逻辑, 粗粒度的锁管理等, 应用程序执行系统调用陷入内核时还需要额外的现场切换、缓存替换等操作, 这些缺陷均是内核文件系统无法避免的. 因此, 一种可行的方案是直接将文件系统部署在用户态.

Aerie [59] 是 University of Wisconsin-Madison 于 2014 年首次提出的用户态持久性内存文件系统,

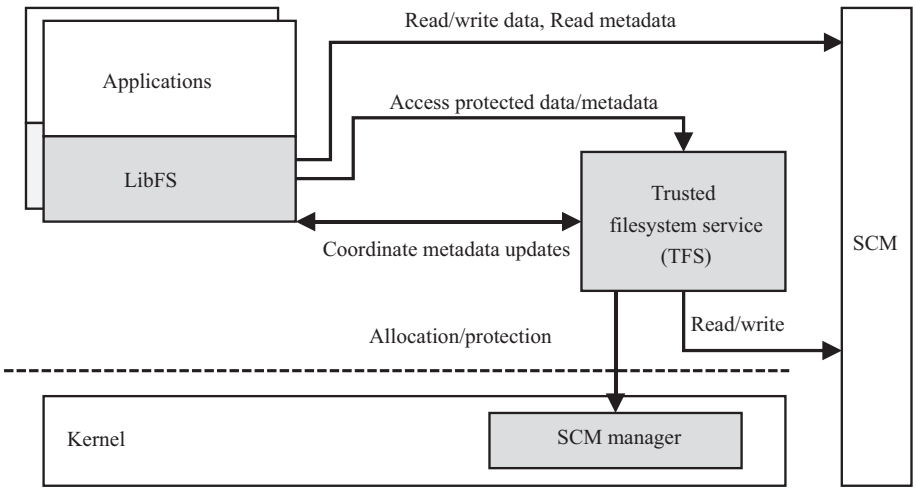


图 17 Aerie 系统架构

Figure 17 The architecture of Aerie

用于解决操作系统进行文件抽象时的低效性问题. 如图 17 所示, Aerie 由 libFS, TFS 和 SCM Manager 3 个模块构成. libFS 是一个用户态的客户端链接库, 它能够直接在用户态进行元数据索引和数据读写, 因此, 大多数文件系统操作均可在用户态实现. libFS 封装了一系列文件系统操作, 应用程序只需链接到该链接库, 便可实现用户态的文件直接访问. 元数据是文件系统的核心数据, 因此其完整性保护更为重要, 基于此考虑, Aerie 将元数据的修改操作交由第三方的可信服务 (trusted file system service, TFS) 完成. 当应用程序需要修改元数据 (如 creat, unlink 等操作), 则通过 libFS 向 TFS 发送修改请求, 然后由 TFS 完成元数据修改. 除此之外, TFS 还通过一个分布式的锁服务器进行不同应用程序之间的并发协调. 为了防止恶意程序随意篡改文件系统镜像, Aerie 还引入了一个 SCM Manager, 进行粗粒度的空间分配及权限管理. 只有可信的应用程序才能拥有文件系统镜像的访问权限.

University of Texas at Austin 于 2017 年提出的 Strata [60] 同样实现在用户态. 不同的是, Strata 同时管理多种存储设备 (NVM, SSD, HDD), 通过合理的数据调度, 使得 Strata 在性能、容量等方面均表现优秀. 同样, Strata 也将文件系统的角色进行拆分, 由 libFS 和 KernelFS 两部分构成. Strata 为每个进程构建了一个日志空间, 应用程序写数据时, 只需通过 libFS 将更新内容追加到当前进程的日志末尾即可. KernelFS 则在后台异步地将各进程日志中的数据进行消化, 整理成结构化的文件数据. 同时, KernelFS 还会根据不同数据块的冷热状态, 裁定数据块的具体摆放位置. University of California, San Diego 于 2019 年进一步提出 Ziggurat [61], 其针对多存储介质下的数据摆放进行了进一步的优化.

6.3 小结

文件系统是数据中心系统软件的核心组件, 如何利用持久性内存设计文件系统是研究人员的热点关注问题. 目前, VFS 的页缓存、元数据缓存等造成的冗余拷贝问题、NVM 中数据崩溃一致性问题等已经得到了较好的解决. 为彻底规避内核抽象带来的软件开销, 一部分工作选择将文件系统直接部署在用户态, 然而, 这也一定程度上造成了新的安全性问题. 同时, libFS 和 TFS 之间仍基于套接字进行通信, 这还是会重新引入内核开销. 因此, 如何充分结合用户态和内核态各自的特点, 设计既安全、又高效的文件系统是未来值得探索的方向.

7 非易失主存的分布式系统

近年来, 全球数据总量正以指数级的速度迅猛增长, 大数据应用在存储容量、存取速度等方面均对数据中心存储系统提出了新的需求. 非易失内存作为新型存储介质, 为数据快速存储提供了新的机遇. 另一方面, 远程直接内存访问 (remote direct memory access, RDMA) 能够在远端 CPU 不参与的情况下直接读写远端内存, 提供了高吞吐 (100 Gbps)、低延迟 (1 μs) 的零拷贝网络传输. 因此, 结合 RDMA 和非易失主存构建高性能的分布式存储系统正被广泛研究.

然而, 研究者发现, 简单地将 NVM 和 RDMA 这些新的器件应用到现有的系统软件中并没达到理想的性能. 这主要由以下几方面的因素造成.

(1) 软件逻辑冗余, 传统软件大多采用模块化的设计, 软件层次分工明确, 这种层次化的设计为软件的版本迭代和团队协同提供了便利. 但是, 直接将这种模块化的思想应用到新的硬件时, 将会导致软件逻辑冗余, 效率低下. 例如, 传统的分布式文件系统需要部署在本地文件系统之上, 并通过本地文件系统管理数据, 再通过分布式软件层次建立跨节点的统一视图. 在这种软件架构模式下, 客户端通过 RDMA 读取文件数据时, 将造成多次数据的冗余拷贝. 值得注意的是, 这些冗余拷贝开销在传统硬件场景下并不明显, 但是, 在 NVM 和 RDMA 等高性能硬件场景下, 多次拷贝将占用大量的执行时间, 严重影响性能.

(2) 分布式协议低效. 传统的分布式协议 (例如两阶段提交协议、两阶段锁协议) 都是基于套接字进行数据传输, 这种收发模型已经存在了数十年. 但是, RDMA 却提供了完全不同的数据传输模式, 它可以在远端 CPU 不参与的情况下直接读写远端内存, 因此, 这为分布式协议的设计也带来了很大的冲击, 迫切需要设计新的分布式协议以及数据传输模型来提升分布式系统性能.

为更好地融合 NVM 和 RDMA 两种技术, 研究者主要从分布式文件系统的软件栈精简、分布式事务系统的协议革新、分布式内存的数据传输模型等方面展开了研究.

7.1 分布式持久性内存文件系统

非易失性内存的访问延迟在百纳秒量级, 远低于磁盘的毫秒量级. RDMA 的远程数据访问延迟在微秒量级, 优于传统以太网的数十微秒量级. 然而, 现有的分布式软件系统设计复杂、层次众多, 引入了较高的软件开销. 尽管现有软件系统的延迟相比于磁盘的高延迟并不明显, 但是在高速的 NVM 与 RDMA 硬件之上软件延迟占比很高.

为有效利用非易失主存和 RDMA 网络的硬件特性, 发挥高速硬件的性能优势, 清华大学于 2017 年提出的分布式持久性内存文件系统 Octopus [62], 通过紧密结合 RDMA 特性, 重新设计了文件系统软件逻辑. 具体地, 各个节点将数据存储区注册到网卡, 并共享到集群使之可被远程直接访问, 进而构建持久性共享内存, 而元数据区域则由服务节点进行本地管理 (如图 18). Octopus 通过引入持久性共享内存以降低数据冗余拷贝, 进而提供接近硬件的读写带宽; 引入客户端主动式数据传输来重新均摊客户端和服务端之间的网络负载; 引入自识别远程过程调用协议以提供低延迟元数据访问性能.

Octopus 构建于分布式持久内存池之上, 即通过 RDMA 网络将不同服务器上的持久性内存互联构建成持久性共享内存池, 并在该共享内存池上构建文件系统. 与数据部分不同, Octopus 只允许元数据被每个服务器本地修改, 从而提供细粒度的高效元数据管理, 降低了复杂性. 客户端通过全路径 Hash 方式定位文件所在服务器, 然后发送文件操作请求至该服务器进行文件读写. Octopus 文件系统对元数据和数据采用了不同的方式进行访问. 对于数据的访问, Octopus 采用了分布式持久共享内存的直接远端访问方式, 并重组了 I/O 流程以均衡服务端处理能力和网络通讯能力. 通过分布式共享内

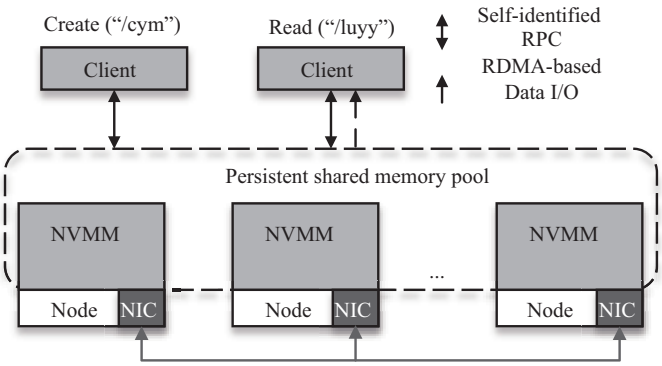


图 18 Octopus 架构 [62]

Figure 18 The architecture of Octopus [62]

存池减少了冗余数据拷贝, 并利用 RDMA 重构了数据 I/O 流以均衡服务器与网络负载, 提高了数据 I/O 效率. 对于元数据访问, Octopus 基于 RDMA_write_with_imm 原语提出了自识别的元数据 RPC, 并利用 RDMA 的单向访问原语设计了 "收集 – 分发式" 分布式事务, 降低了元数据延迟, 提高了吞吐.

University of California, San Diego 于 2019 年提出 Orion 分布式持久性内存文件系统, 其本地存储仍基于 NOVA [56], 通过 RDMA 网络将其扩展到分布式环境. Orion 具有以下特点: (1) 不同于传统的分布式系统软件, Orion 在内核态实现, 因此, 该系统完全兼容现有的 POSIX 语义. 同时, Orion 还维护了多个副本的元数据和数据, 因而具备很强的容灾能力. (2) Orion 充分结合了 NVM 和 RDMA 的特性, 通过 RDMA 的单向原语直接读取远端日志, 极大降低了单个元数据服务器的处理压力. (3) Orion 在文件系统中引入了乐观并发控制, 以协调并发的应用程序. 同时, Orion 也精细地挑选了不同的 RDMA 原语, 用于不同场景下的跨节点通信, 以充分发挥 RDMA 的硬件优势.

Ohio State University 于 2016 年提出的 NVFS [63], 将非易失内存和 RDMA 结合起来, 用于加速 HDFS. 但由于 HDFS 本身软件设计厚重, NVFS 并未充分发挥 NVM 和 RDMA 的硬件特性. IBM 在 2016 年提出的 Crail 文件系统 [64], 其元数据管理基于其之前开发的 DaRPC [65], 而数据传输依赖于 RDMA 单向原语. DaRPC 是一个基于 RDMA 的 RPC 系统, 它将消息处理和网络传输紧密结合, 提供高吞吐、低延迟的跨网传输性能. Crail 相比于 Octopus 多一次冗余拷贝, 因此其带宽不如 Octopus; 最近, Crail 开源项目已经融合了多种存储介质, 以提供大容量、高吞吐的数据存储性能.

7.2 非易失主存的远程复制

为保证 NVM 中数据的一致性存储, CPU 需执行刷写指令强制将缓存中的数据持久化到 NVM 中. 然而, 刷写带来的极高开销却制约了系统的整体性能. Mojim [66] 发现, 基于 RDMA 的异地备份方案性能强于基于本地缓存刷写的持久化方案. 这主要是因为 CPU 刷写缓存数据以 cache line 为粒度, 且并发的数据持久写被 CPU 强制顺序性执行, 并行度严重受到限制. 因此, RDMA 网络为分布式持久性内存系统中的数据持久化和远程备份提供了新的机遇.

Mojim 是一个构建在 NVM 上的内核态系统, 图 19 描述了 Mojim 的系统架构. Mojim 采用了基于主层和辅层的双层架构, 其主层包括 1 个主节点和 1 个镜像节点, 辅层包含一个或多个备份节点. 它可以向上层提供基础的数据读写接口和不同级别的持久化接口, 具体包括 M-sync, M-async 和 M-syncsec. 根据不同的持久化级别, 写入 Mojim 的数据同步或异步地流入到主节点、镜像节点和备份节点. 其中, 以 M-sync 模式写入时, Mojim 将数据首先写入到本地 (不执行 CPU 缓存数据刷写), 然

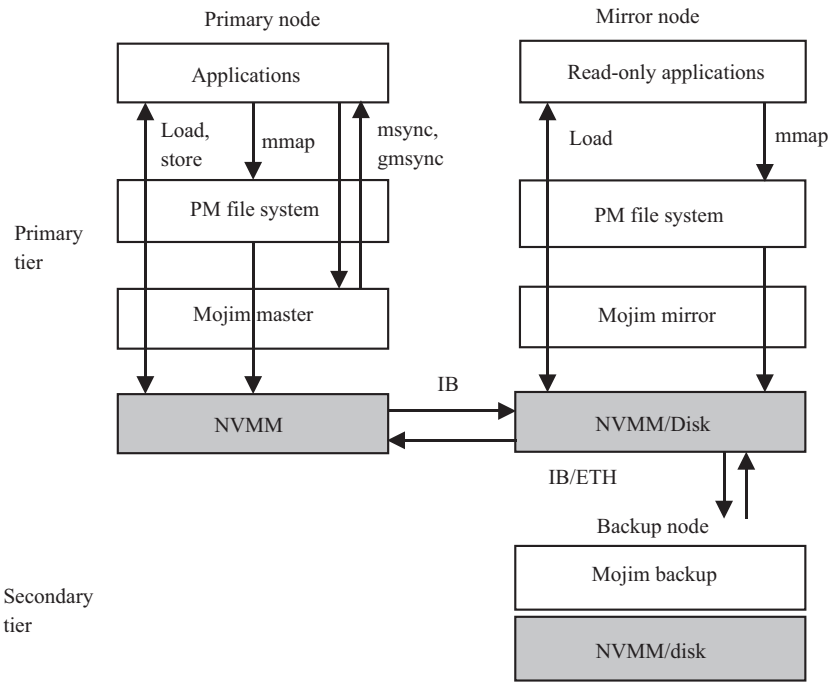


图 19 Mojim 架构 [66]

Figure 19 The architecture of Mojim [66]

后将数据通过 RDMA 网络传输到镜像节点, 等待对方网卡反馈确认信息后成功返回. 而镜像节点异步将数据传递到备份节点. 以 M-async 模式写入时, 主节点首先刷写缓存数据, 然后将数据传输到镜像节点, 在不等待确认信息时直接成功返回. M-syncsec 则同步等待数据写入到备份节点之后才成功返回. 以上 3 类持久写方式具有不同级别的可靠性、可用性和一致性.

7.3 分布式持久性共享内存

分布式共享内存的概念早在 20 世纪 80 年代就已经被提出 [67~70], 其核心思想是将多个服务器的内存通过网络进行统一管理, 向应用程序抽象出统一的全局地址空间. 然而, 传统以太网由于带宽低、延迟高, 严重制约了分布式共享内存的性能, 从而也限制了其进一步发展. 近年来, 随着快速网络技术的发展, RDMA 的硬件性能已经越来越靠近内存, 这为分布式共享内存的进一步发展注入了新的活力.

Hotpot [71] 是 Purdue University 于 2017 年提出的分布式持久性共享内存框架, 它向应用程序提供了全局共享的持久性内存空间, 应用程序可以像访问本地内存一样通过 load/store 指令访问远端持久性内存 (如图 20 所示). Hotpot 可以使单节点应用程序拥有跨节点海量内存空间, 同时也能帮助共享内存应用程序提升持久性内存存储性能. 为方便用户管理数据, Hotpot 提供了一套持久性名字空间, 应用程序可以通过 open/close 等接口访问命名后的持久性数据. 为提升数据存储可靠性, Hotpot 还增加了复制机制. 与传统方式不同, Hotpot 巧妙地结合了本地缓存和数据复制的特性, 设计出了单层系统架构: 即本地缓存的同时, 缓存的数据也被管理为一份持久性复制数据. 为保证多份数据的一致性, Hotpot 还引入了一致性提交协议, 用于支持 “多读多写” 和 “多读单写” 等访问模式.

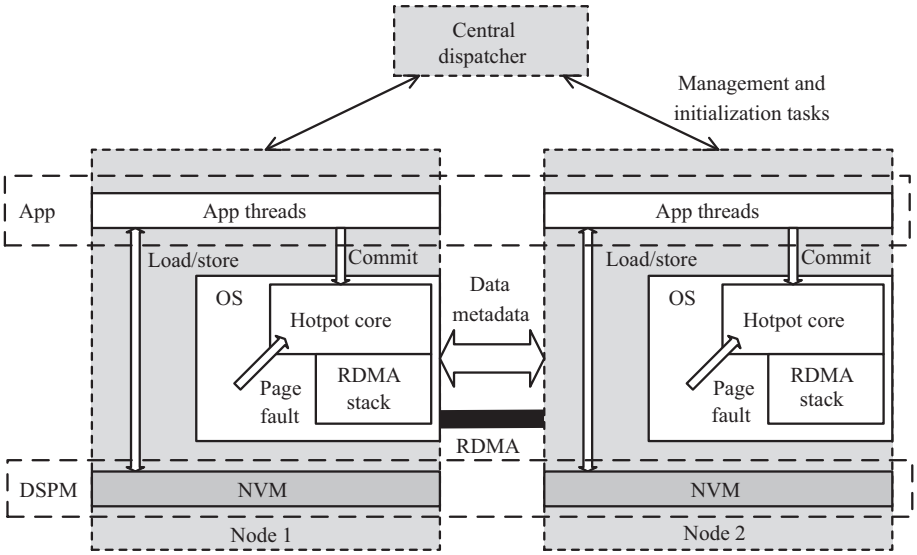


图 20 Hotpot 架构 [71]

Figure 20 The architecture of Hotpot [71]

7.4 分布式事务系统

FaRM [72, 73] 是 Microsoft 研究院在 2014 年提出的基于 RDMA 的分布式内存计算平台, 旨在提供高一致性、高可用性和高性能的分布式事务处理能力. FaRM 将各节点 DRAM 通过 RDMA 互连成共享内存, 向应用程序提供内存级数据存储. UPS 系统保证了 DRAM 中数据掉电不丢失, 一旦发生断电事故, 数据将被及时从 DRAM 写回到 SSD. FaRM 提出了 3 种优化技术, 用于避免在配备高速硬件后 CPU 成为新的瓶颈: (1) 减少消息发送次数; (2) 尽量使用单向 RDMA 原语 (RDMA read/write), 而避免使用消息通讯; (3) 尽量让事务处理可并行化.

FaRM 首先将各节点内存划分为 2 GB 的内存区, 然后通过一致性 Hash 算法决定内存分配的策略. 为保证内存的一致性更新, FaRM 还提供了基于分布式事务的共享内存读写接口, 通过乐观锁和两阶段提交协议来保证事务执行的原子性和可序列化. 图 21 具体展示了 FaRM 的事务执行逻辑, 其中包含 1 个协调者 (C)、3 个参与者, 以及他们的备份节点. 此外, FaRM 通过增加 cache line 粒度的版本控制实现无锁读, 在不加锁的情况下执行事务读取, 并感知事务的一致性, 判断是否发生脏读. 同时, FaRM 在事务执行过程中还实时进行数据备份, 从而提高了数据可用性.

7.5 小结

RDMA 网络和持久性内存等新型硬件的出现为现有的分布式系统软件带来了新的机遇与挑战. 区别于传统网络和外存存储, 全新的数据访问接口和访问延迟将彻底改变现有的软件设计方式, 这为数据布局与索引、网络 I/O 机制、分布式协议构建等方面提供了新的设计需求. 合理的软件重构更能充分发挥新型硬件的优势, 提升系统的整体性能.

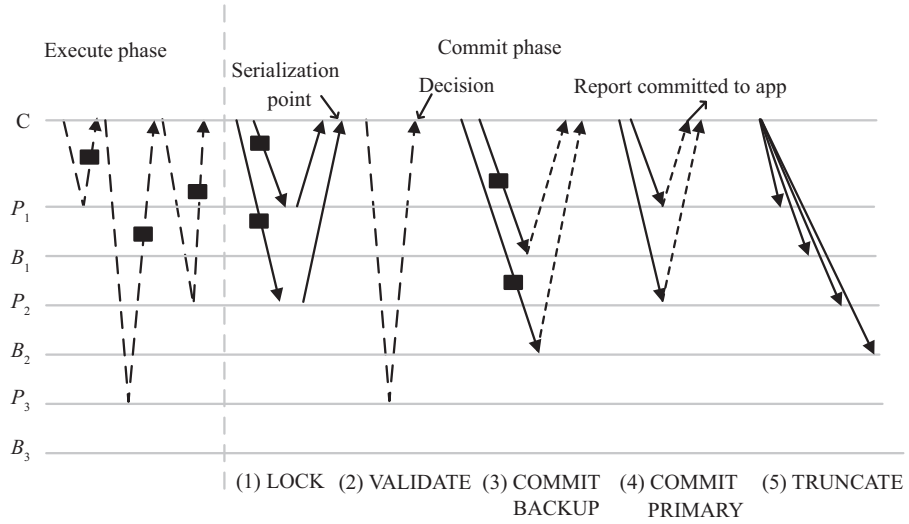


图 21 FaRM 架构 [71]

Figure 21 The architecture of FaRM [71]

8 总结与展望

现有工作从空间管理机制、编程模型设计、索引结构设计、文件系统、分布式存储系统等方面入手, 有效地解决了非易失主存中存在的若干问题. 下面, 本文结合新型网络通讯协议 RDMA 和新型计算机架构方式 NUMA 等带来的变化, 展望后续的研究工作:

(1) 基于 RDMA 和 NVM 的分布式持久性事务主存系统设计. 现有工作主要针对的是单机场景下非易失主存上的事务问题. 然而, 单机系统面临着扩展性的问题, 具备良好扩展性的分布式事务系统可以满足日益增长的应用需求. NVM 技术支持低延迟的持久性数据访问, 而 RDMA 技术提供高效的远程主存数据传输, 这改变了传统分布式存储系统中数据持久化和通信操作延迟过高的假设, 为分布式系统的设计提供了新的机遇. 如何针对 NVM 和 RDMA 良好的性能, 减少传统系统软件栈的开销, 提供保证一致性的分布式事务接口, 这将是基于 RDMA 和 NVM 的分布式系统设计中重要的挑战.

(2) 基于 NUMA 和 NVM 的索引结构设计. 现有工作消除了传统持久性 B+ 树中的平衡/排序开销. 然而, 它们并不是 NUMA 架构感知的, 没有针对 NUMA 架构中远近端主存访问性能不同的特性设计相应的优化技术. 随着多处理器技术的发展, 基于 NUMA 架构的服务器被广泛使用在数据中心中, NUMA 架构感知的索引结构变得越来越重要. 如何针对 NUMA 和 NVM 的特性, 设计 NUMA 架构感知的数据分布机制和并发控制机制, 这将是未来索引结构设计中急需解决的一个问题.

参考文献

1 Turner V, Gantz J F, Reinsel D, et al. The digital universe of opportunities: rich data and the increasing value of the internet of things. IDC Anal Future, 2014, 2: 3

2 Gerber D, Hellmann S, Bühmann L, et al. Real-time RDF extraction from unstructured data streams. In: Proceedings of the 12th International Semantic Web Conference, 2013. 135–150

3 Bryant R E, O'Hallaron D R. Computer Systems: A Programmer's Perspective. Gewerbestrasse: Pearson Schweiz AG, 2015

4 Bedeschi F, Resta C, Khouri O, et al. An 8 Mb demonstrator for high-density 1.8 V phase-change memories. In: Proceedings of Symposium on Vlsi Circuits, 2004. 442–445
5 Chen E, Apalkov D, Diao Z, et al. Advances and future prospects of spin-transfer torque random access memory. IEEE Trans Magn, 2010, 46: 1873–1878
6 Lee H Y, Chen Y S, Chen P S, et al. Evidence and solution of over-RESET problem for HfOX based resistive memory with sub-ns switching speed and high endurance. In: Proceedings of International Electron Devices Meeting, 2011
7 Parkin S S P, Hayashi M, Thomas L. Magnetic domain-wall racetrack memory. Science, 2008, 320: 190–194
8 Newsroom I. Intel optane memory series 32 GB M.2 80MM. 2018. https://www.intel.com/content/www/us/en/products/memory-storage/optane-memory/optane-32gb-m-2-80mm.html
9 Swanson S, Caulfield A M. Refactor, reduce, recycle: restructuring the I/O stack for the future of storage. Computer, 2013, 46: 52–59
10 Lu Y Y, Shu J W, Sun L. Blurred persistence in transactional persistent memory. In: Proceedings of the 31st International Conference on Massive Storage Systems and Technology, 2015
11 Volos H, Tack A J, Swift M M. Mnemosyne: lightweight persistent memory. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011. 91–104
12 Bianca G, Pawel P. PMDK: persistent memory development kit. 2003. https://github.com/pmem/pmdk
13 Hu Q D, Ren J L, Badam A, et al. Log-structured non-volatile main memory. In: Proceedings of USENIX Annual Technical Conference, 2017. 2–44
14 Grossman G, Star Z. Intel-64 and IA-32 architectures software developers manuals combined. 2019. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
15 Janek M, Pmem B. Intel architecture instruction set extensions programming reference. 2016. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf
16 Zhou Y Q, Alagappan R, Memaripour A, et al. HNVM: Hybrid NVM Enabled Datacenter Design and Optimization. Microsoft Research Technical Report. 2017
17 Nishtala R, Fugal H, Grimm S, et al. Scaling memcache at Facebook. In: Proceedings of USENIX Conference on Networked Systems Design and Implementation, 2013. 385–398
18 Rumble S M, Kejriwal A, Ousterhout J. Log-structured memory for DRAM-based storage. In: Proceedings of USENIX Conference on File and Storage Technologies, 2014
19 Hertz M, Berger E D. Quantifying the performance of garbage collection vs. explicit memory management. In: Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages and Applications, 2005. 313–326
20 Coburn J, Caulfield A M, Akel A, et al. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, 2011. 105–118
21 Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory. In: Proceedings of the 9th ACM SIGOPS/EuroSys European Conference on Computer Systems, 2014
22 Bhandari K, Chakrabarti D R, Boehm H J. Makalu: fast recoverable allocation of non-volatile memory. In: Proceedings of the ACM Sigplan International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2016. 677–694
23 Oukid I, Booss D, Lespinasse A, et al. Memory management techniques for large-scale persistent-main-memory systems. In: Proceedings of the 43rd International Conference on Very Large Data Bases, 2017. 1166–1177
24 Schwalb D, Berning T, Faust M, et al. nvm_malloc: memory allocation for NVRAM. In: Proceedings of the 6th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, 2015. 285–287
25 Hwang T, Jung J, Won Y. HEAPO: heap-based persistent object store. Trans Storage, 2015, 11: 1–21
26 Bridge B. NVM direct. 2015. https://github.com/oracle/nvm-direct
27 Mathur A, Cao M M, Bhattacharya S, et al. The new ext4 filesystem: current status and future plans. In: Proceedings of Linux Symposium, 2007
28 Kannan S, Gavrilovska A, Schwan K. pVM: persistent virtual memory for efficient capacity scaling and object storage. In: Proceedings of the 11th ACM SIGOPS/EuroSys European Conference on Computer Systems, 2016

29 Oukid I, Lasperas J, Nica A, et al. FPTree: a hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: Proceedings of ACM International Conference on Management of Data, 2016

30 Bob B, Hans B. NVM programming model. 2017. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf

31 Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory. In: Proceedings of ACM Symposium on Operating Systems Principles, 2009. 133–146

32 Moraru I, Andersen D G, Kaminsky M, et al. Persistent, Protected and Cached: Building Blocks for Main Memory Data Stores. CMU Parallel Data Lab Trechnical Report, 2012

33 Lu Y Y, Shu J W, Sun L, et al. Loose-ordering consistency for persistent memory. In: Proceedings of IEEE International Conference on Computer Design, 2014. 216–223

34 Wang Z G, Yi H, Liu R, et al. Persistent transactional memory. IEEE Comput Arch Lett, 2015, 14: 58–61

35 Pelley S, Chen P M, Wenisch T F. Memory persistency. In: Proceeding of International Symposium on Computer Architecuture, 2014. 265–276

36 Kolli A, Pelley S, Saidi A, et al. High-performance transactions for persistent memories. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 2016. 399–411

37 Kolli A, Rosen J, Diestelhorst S, et al. Delegated persist ordering. In: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2016

38 Narayanan D, Hodson O. Whole-system persistence. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, 2012. 401–410

39 Zhao J S, Li S, Yoon D H, et al. Kiln: closing the performance gap between systems with and without persistence support. In: Proceedings of IEEE/ACM International Symposium on Microarchitecture, 2017. 421–432

40 Sun L, Lu Y Y, Shu J W. DP2: reducing transaction overhead with differential and dual persistency in persistent memory. In: Proceedings of the 12th ACM International Conference on Computing Frontiers, 2015

41 Memaripour A, Badam A, Phanishayee A, et al. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In: Proceedings of the 12th ACM SIGOPS/EuroSys European Conference on Computer Systems, 2017. 499–512

42 Denny J E, Lee S, Vetter J S. NVL-C: static analysis techniques for efficient, correct programming of non-volatile main memory systems. In: Proceedings of ACM International Symposium on High-Performance Parallel and Distributed Computing, 2016. 125–136

43 Bayer R, McCreight E M. Organization and maintenance of large ordered indexes. Acta Inform, 1972, 1: 173–189

44 Chen S M, Gibbons P B, Nath S. Rethinking database algorithms for phase change memory. In: Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, 2011. 21–31

45 Chi P, Lee W C, Xie Y. Making B^+-tree efficient in PCM-based main memory. In: Proceedings of International Symposium on Low Power Electronics and Design, 2014. 69–74

46 Zuo P F, Hua Y. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. IEEE Trans Parallel Distrib Syst, 2018, 29: 985–998

47 Tolia N, Tolia N, Ranganathan P, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In: Proceedings of USENIX Conference on File and Storage Technologies, 2010

48 Chen S M, Jin Q. Persistent B^+-trees in non-volatile main memory. In: Proceedings of the 41st International Conference on Very Large Data Bases, 2015

49 Yang J, Wei Q S, Chen C, et al. NV-tree: reducing consistency cost for NVM-based single level systems. In: Proceedings of USENIX Conference on File and Storage Technologies, 2015

50 Hwang D, Kim W H, Won Y, et al. Endurable transient inconsistency in byte-addressable persistent B^+-tree. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies, 2018

51 Lee S K, Lim K H, Song H, et al. WORT: write optimal radix tree for persistent memory storage systems. In: Proceedings of USENIX Research in Linux File and Storage Technologies, 2017. 257–270

52 Zuo P F, Hua Y, Wu J. Write-optimized and high-performance hashing index scheme for persistent memory. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2018. 461–476

53 Xia F, Jiang D J, Xiong J, et al. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In: Proceedings of USENIX Annual Technical Conference, 2017. 349–362

54 Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable persistent memory. In: Proceedings of the 22nd Symposium on Operating Systems Principles, 2009. 133–146
55 Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory. In: Proceedings of the 9th European Conference on Computer Systems, 2014
56 Xu J, Swanson S. A log-structured file system for hybrid volatile/non-volatile main memories. In: Proceedings of Conference on File and Storage Technologies, 2016. 323–338
57 Wu X J, Reddy A L. SCMFS: a file system for storage class memory. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, 2011
58 Wang Y, Jiang D Y, Xiong J. Caching or not: rethinking virtual file system for non-volatile main memory. In: Proceedings of Workshop on Hot Topics in Storage and File Systems, 2018
59 Volos H, Nalli S, Panneerselvam S, et al. Aerie: flexible file-system interfaces to storage-class memory. In: Proceedings of the 9th European Conference on Computer Systems, 2014. 4–14
60 Kwon Y, Fingler H, Hunt T, et al. Strata: a cross media file system. In: Proceedings of the 26th Symposium on Operating Systems Principles, 2017. 460–477
61 Zheng S G, Hoseinzadeh M, Swanson S. Ziggurat: a tiered file system for non-volatile main memories and disks. In: Proceedings of Conference on File and Storage Technologies, 2019. 207–219
62 Lu Y Y, Shu J W, Chen Y M, et al. Octopus: an RDMA-enabled distributed persistent memory file system. In: Proceedings of USENIX Annual Technical Conference, 2017. 773–785
63 Islam N S, Wasi-ur-Rahman M, Lu X Y, et al. High performance design for HDFS with byte-addressability of NVM and RDMA. In: Proceedings of International Conference on Supercomputing, 2016
64 Stuedi P, Trivedi A, Pfefferle J, et al. Crail: a high-performance I/O architecture for distributed data processing. IEEE Data Eng Bull, 2017, 40: 38–49
65 Stuedi P, Trivedi A, Metzler B, et al. DaRPC: data center RPC. In: Proceedings of ACM Symposium on Cloud Computing, 2014
66 Zhang Y Y, Yang J, Memaripour A, et al. Mojim: a reliable and highly-available non-volatile memory system. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, 2015. 3–18
67 Amza C, Cox A L, Dwarkadas S, et al. TreadMarks: shared memory computing on networks of workstations. Computer, 1996, 29: 18–28
68 Li K. IVY: a shared virtual memory system for parallel computing. In: Proceedings of International Conference on Parallel Processing, 1988
69 Bennett J K, Carter J B, Zwaenepoel W. Munin: distributed shared memory based on type-specific memory coherence. SIGPLAN Not, 1990, 25: 168–176
70 Zhou S, Stumm M, Li K, et al. Heterogeneous distributed shared memory. IEEE Trans Parallel Distrib Syst, 1992, 3: 540–554
71 Shan Y Z, Tsai S Y, Zhang Y Y. Distributed shared persistent memory. In: Proceedings of Symposium on Cloud Computing, 2017. 323–337
72 Dragojevic A, Narayanan D, Hodson O, et al. FaRM: fast remote memory. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, 2014. 401–414
73 Dragojevic A, Narayanan D, Nightingale E B, et al. No compromises: distributed transactions with consistency, availability, and performance. In: Proceedings of the 25th Symposium on Operating Systems Principles, 2015. 54–70

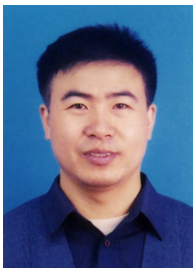
Development of system software on non-volatile main memory

Jiwu SHU*, Youmin CHEN, Qingda HU & Youyou LU

Department of Computer Science and Technology, Tsinghua University, Beijing 100084*, China*
* Corresponding author. E-mail: shujw@tsinghua.edu.cn

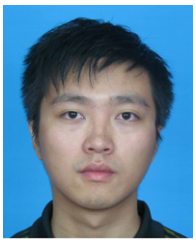
Abstract The rapid development of the Internet has led to explosive growth in the total volume of global data storage, leading to the development of data systems from computing intensive to data intensive. How to build a reliable and efficient data storage system has become an urgent problem to be solved in the era of big data. Compared with traditional disks, non-volatile main memory has the advantages of high performance and byte-addressable characteristics, and these unique advantages provide new opportunities for the construction of efficient storage systems. However, the construction of a traditional storage system is not suitable for non-volatile main memory. It cannot give play to the performance advantage of non-volatile main memory, and it is easy to result in high consistency overhead, low space utilization ratio, and low programming security. Therefore, this paper analyzes the challenges for the storage system based on non-volatile main memory, and summarizes the research development of the spatial management mechanism, new programming model, data structure, file systems, and distributed storage systems. Finally, this paper lists the potential research direction of the storage system based on the non-volatile main memory.

Keywords non-volatile main memory, system software, memory management, programming model, data structure, file system, distributed system



Jiwu SHU was born in 1968. He received his Ph.D. degree in computer science from Nanjing Univeristy, Nanjing, in 1998. Currently, he is a professor in Department of Computer Science and Technology at Tsinghua University. His research interests include storage security and reliability, non-volatile memory based storage systems, and parallel and distributed computing.

Youmin CHEN was born in 1993. He is currently a Ph.D. candidate in Department of Computer Science and Technology at Tsinghua Univeristy. His research interests include non-volatile memories, distributed systems and file systems.



Qingda HU was born in 1990. He received his Ph.D. degree in Department of Computer Science and Technology at Tsinghua in 2018. His research interests include non-volatile memories, distributed systems and file systems.

Youyou LU was born in 1987. He obtained his B.S. degree in computer science from Nanjing University in 2009, and his Ph.D. degree in computer science from Tsinghua University in 2015. Currently, he is an assistant professor in Department of Computer Science and Technology at Tsinghua. His research interests include non-volatile memories and file systems.