分类号_____ 学号 M201873043

学校代码 10487 密级_____

华中科技大学

硕士学位论文

（学术型 ☑ 专业型 □）

一种基于非易失存储器的用户态文件系统的研究与实现

学位申请人：文 多

学 科 专 业：计算机系统结构

指 导 教 师：谭志虎 教授

答 辩 日 期：2021 年 05 月 23 日

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Master Degree in Engineering

Research and Implementation of a User Mode File System Based on NVM

Candidate : WEN Duo

Major : Computer Architecture

Supervisor : Prof. TAN Zhihu

Huazhong University of Science and Technology

Wuhan 430074, P. R. China

May, 2022

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于　保　密□，在_____年解密后适用本授权书。

　　不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

在现今大数据时代，文件系统作为重要的基础软件之一，是支撑数据中心和云基础设施的关键技术。基于硬盘的文件系统技术上已经相当成熟，但随着访问性能接近内存的新型非易失存储设备的推出和商用，逐渐有替代硬盘成为新的底层存储的趋势。而传统的文件系统技术不能有效发挥出非易失存储器的性能和特性，如何解决两者结合存在的性能低下的复杂软件栈、文件系统原子语义和用户态文件系统元数据同步开销等问题成为新的研究热点，给学术界和产业界提出了新的挑战。

为充分发挥非易失存储器的性能，消除用户态文件系统元数据同步开销，并提供可靠的文件系统原子语义，提出了一个用户态访问的新型文件系统 UnvmFS。UnvmFS 设计了一个非易失存储分配器管理元数据和数据，用户态各进程独立更新元数据，消除中心元数据模块与用户进程的通信开销。持久化的元数据设计，能节省掉电后系统重构恢复或文件系统正常重启的时间，重启即可用。针对非易失存储器字节寻址无法保证文件系统原子语义的问题，UnvmFS 设计了日志结构数据更新机制对文件进行原子更新；设计了轻量级写前日志机制保证目录操作原子性，维护文件系统数据和元数据一致性。

基于真实的非易失存储器（Intel Optane DC Persistent Memory），实现了 UnvmFS 系统原型，并使用权威文件系统测试工具 filebench 进行了系统测试。基本读写的 Microbenchmark 负载测试，UnvmFS 比 2020 年新发表的 libnvmmio 写带宽提高约 15%至 35%，读写带宽比 nova 和 ext4-dax 提高约 27%至 182%。模拟真实访问的 Macrobenchmark 负载测试，UnvmFS 比 libnvmmio 性能提高约 16%至 134%，比 nova 和 ext4-dax 提高约 18%至 214%。实验结果表明，UnvmFS 的设计改进可行且有效。

关键词：文件系统；非易失存储器；内存映射；用户态；原子更新

Abstract

As one of the important basic software, file system is the key technology supporting data center and cloud infrastructure in the era of big data. The file system based on the hard disk is quite mature in technology, but with the introduction and commercial use of new non-volatile storage devices with access performance close to the memory, there is a tendency to replace hard disks as the new underlying storage. However, traditional file system technology cannot effectively exert the performance and characteristics of non-volatile memory. How to solve the problems of low-performance complex software stacks, file system atomic semantics, and user-mode file system metadata synchronization costs that exist in the combination of the two has become a new research hotspot, and has become a new challenge for academic and industrial circles.

In order to give full play to the performance of non-volatile memory, eliminate user-mode file system metadata synchronization overhead, and provide reliable file system atomic semantics, a new file system UnvmFS with full user-mode access is proposed. UnvmFS designs a non-volatile storage allocator to manage metadata and data, and each process in the user mode independently updates the metadata, eliminating the communication overhead between the central metadata module and the user process. The persistent metadata design can also save the time of system reconstruction and recovery after power failure or normal restart of the file system, which can be used after restart. In view of the problem that non-volatile memory byte addressing cannot guarantee the atomic semantics of the file system, UnvmFS designs a log structure data update mechanism to perform atomic updates on files, and a lightweight pre-write log mechanism to ensure the atomicity of directory operations and maintain file system data and Metadata consistency.

Based on the real non-volatile memory (Intel Optane DC Persistent Memory), a prototype of the UnvmFS system was implemented, and the system was tested using filebench, an authoritative file system testing tool. In the basic read and write Microbenchmark load test, UnvmFS has a 15% to 35% increase in write bandwidth compared to the newly released libnvmmio in 2020, and a 27% to 182% increase in read and write bandwidth compared to nova and ext4-dax. The Macrobenchmark load test that simulates real access shows that UnvmFS is 16% to 134% higher than libnvmmio, and 18%

to 214% higher than nova and ext4-dax. Experimental results show that the design improvement of UnvmFS is feasible and effective.

Key words: File System, Non-volatile Memory, Memory Map, User Mode, Atomic Update

目 录

1 绪论

1.1 研究背景与意义

（1）大数据时代海量存储需求与应用

在当前大数据时代背景下，以“数据为中心”的模式逐渐替代了“计算为中心”的模式。数据中心[1][2]对当前数据应用不仅仅追求存储容量的增长，更是要挖掘出海量数据的巨大价值，利用云基础设施[3][4]给用户提供最快的响应和最高的带宽。2020年初，面对突如其来的新冠疫情，大数据在疫情防控和保障人民正常学习生活中发挥了至关重要的作用。国家基层管理和公共卫生管理存在基层干部人员不足、数据识别管理和预判能力不够等短板，但政府以疫情数据为基础，构建疫情分布图、疫情信息反馈、风险预警和健康码等机制，建立起标准化及智能化的疫情管理机制，补齐了这些短板，也使得中国从疫情当中快速复苏。不仅如此，中国工业界和企业界在疫情下仍不断前进，大力发展在线医疗、在线教育、在线会议和在线办公等信息化新兴产业，使得大数据信息得以应用，给传统企业的数字化转型带来新的机遇，塑造自身新的核心竞争力。

（2）存储与计算瓶颈问题

在近几十年来，计算机硬件技术高速发展和进步，处理器从以前靠集成电路工艺提升增加晶体管数量的单 CPU 核处理器模式，到现在集成电路成熟靠 CPU 多核技术的发展，其计算速度提高一直遵循每十八个月性能翻倍的“摩尔定律”[5]。而一般存储器件如机械硬盘的机械部件旋转速度已达极限，虽然固态硬盘使用电子闪存芯片性能高了一个量级，但是存储器件的性能提升仍然远远落后于处理器性能。计算与存储相辅相成，计算与存储的性能差距产生的巨大瓶颈，在计算机系统应用中愈发明显，限制了云计算与大数据的发展。外存的存储速度相比于内存始终隔着无法逾越的差距，又迫于当前应用对存储系统性能的要求越来越高，在如今内存容量不断增大而价格不断走低的情形下，研究者们和开发者们开始把大量的数据直接放在易失的内存，衍生出新的概念“内存计算”[6]；这种方式虽然可以降低内存与外存频繁的拷贝

替换，但也产生了数据不一致和异常、掉电导致大量数据丢失的问题，使得系统崩溃无法使用。

（3）新存储介质非易失内存（Non-Volatile Memory，NVM）发展

在实际应用中，系统可靠性比系统性能更重要，使用易失的内存存储数据始终存在安全性问题。幸好近些年，非易失存储技术[7][8][9]得到了很大的发展，新型非易失存储器结合了内存和硬盘双方的特点。NVM 拥有与内存接近的访问开销、与内存一样能直接通过内存总线访问，从而具有字节访问的特性，但其价格却远低于内存，容量远高于内存且具有数据掉电不丢失的优势。与外存设备如机械硬盘和固态硬盘相比，NVM 能被 CPU 直接访问，访问延迟很低，能耗少，还有比固态硬盘长的寿命。在未推出商用产品时，非易失存储设备便已成为学术界争相研究的对象。并随着近几年英特尔等公司已经推出商用产品，企业界也开始应用其提高存储性能。随着非易失存储器的工艺不断改进有取代硬盘和内存的趋势。

（4）NVM 为文件系统性能提升带来新机会

文件系统作为最常使用的基础软件之一，也是核心的存储系统软件之一，其 I/O 速度能直接反映上层应用的性能好坏，非易失存储技术的发展给了文件系统性能提升提供了的新台阶。每次新存储产品的技术变革都给计算机系统软硬件架构带来巨大挑战，文件系统和非易失存储器件的结合，势必会带来数据结构、硬件架构、软件堆栈、事务技术和编程工具等各方面的改变与优化。以前文件系统的性能优化集中在对文件系统组织架构的改进和针对机械硬盘特性将随机转顺序访问模式的优化；但在现有文件系统直接使用 NVM 不能充分发挥其性能，还会引起数据一致性的问题。而且基于内核的传统虚拟文件系统（Virtual File System，VFS）软件层在 NVM 的访问开销是不透明的，也不适应于 NVM 的硬件特点。重新从 NVM 特点的角度，开发一套能完全挖掘其访问性能，且维护文件系统原有操作语义的新型文件系统是现今的研究热点。

（5）文件系统与 NVM 结合存在的问题亟待解决

目前迫切需要定制化的基于非易失存储器特性的文件系统，来最大化文件系统和非易失存储器各自的优势，从而达到“1+1>2”的效果。目前学术界已经推出了一些

基于非易失存储器的文件系统，有的在内核态实现对 NVM 直接访问以此简化软件栈，有的将 NVM 的访问放在用户态减少进入内核次数，有的则致力于解决数据访问的安全性问题，实现了很多有建设性的研究。但是目前实现的基于 NVM 的内核态文件系统仍依赖于统一的 VFS 层兼容原有应用，复杂的软件栈开销使得非易失存储器性能无法充分发挥；而实现的基于 NVM 的用户态文件系统依赖传统的内核态文件系统对元数据进行管理和访问，两套文件系统的组合拳方式并不等同于一个完整的新文件系统，有科研价值但实用价值却微乎其微；并且已实现的用户态文件系统受限于用户空间进程间地址独立，需要额外的模块处理并发访问下元数据同步，存在单点问题。NVM 字节寻址的特性，为适配现有文件系统语义，使得文件系统要开发功能维护自身数据和元数据访问的原子性。NVM 容量较硬盘还是比较有限的，现有的一些预分配空间的文件系统以空间换时间的方式，有待商榷。以上等等都是目前未研究到或还存在优化空间的问题，本文基于此背景决定设计并实现一种基于非易失存储器的高性能全用户态访问的文件系统，充分发挥非易失存储器性能，维护文件系统原子语义，并消除用户态下元数据同步开销。

1.2 国内外研究现状

本节将首先介绍非易失存储器件和文件系统软件栈的相关知识，然后分别对基于非易失性存储的内核态和用户态文件系统相关研究进行介绍和分析。

1.2.1 非易失存储器件

近些年固态硬盘引领了存储技术的变革，而非易失存储器件将在后闪存时代发挥至关重要的作用。本文的非易失存储器件特指字节寻址的非易失内存存储设备，非易失存储器件并非指特定的某种存储介质而是对一系列非易失存储介质的统称。非易失存储器件既有易失内存的性能优势还兼顾了外存的特点，具有非易失、存取速度快、存储容量大、静态功耗低和每比特价格较低等优点，已经成为大数据时代炙手可热的存储产品，能解决学业界和工业界在存储器体系上追求的目标，以相对较低的价格获得较高的性能。常见的非易失存储器件介绍如下：

相变存储器[10][11]（Phase Change Memory，PCM）由硫族化合物合金材料构成，

利用材料在非晶体和晶体状态的可逆转的相位变化存储信息。PCM 利用有序晶态的低电阻和无序非晶态的高电阻，两者间相差大约 4 到 5 个数量级的阻抗，实现存储逻辑。PCM 通过注入一个窄而强的电流到存储单元，使熔点达到熔态并快速冷却，转为非晶态进行写“0”；注入一个宽而弱的电流到存储单元，达到晶化温度但熔点以下，转为晶态进行写“1”。读过程则通过对存储单元电极上加电压，若测量到的反馈电流很微弱则电阻很大，数据为非晶态的“0”，反之电流很大则电阻较小，数据为晶态的“1”。PCM 写操作无需先擦除，基于电阻存数据比 FLASH 更可靠。

磁变随机存取存储器[12]（Magnetic Random Access Memory，MRAM）利用存储介质磁化特性存储数据，外磁场不变磁化特性不变，在掉电时依然永久存储数据。MRAM 的工作原理是，给金属三明治结构的存储单元在字线和位线施加一定电流，通过产生的磁场改变上层磁层的 N-S 走向，根据上层与下层平行或反平行两种状态来表示“0”和“1”。MRAM 可以达到 CPU 高速存储的速度水平，寿命极长，但其存储密度低，目前更适合作为 CPU 的缓存存储器。

自旋转移力矩存储器[13][14]（Shared Transistor Technology Random Access Memory，STT-RAM）是第二代的 MRAM 技术，其不是通过电流改变磁场方向来写入数据，而是让磁位元被一种极化旋转电流翻转，此种方式可以降低功耗并增大容量。美国硅谷新创公司 Grandis 称其结合了 SRAM 性能优势、DRAM 成本优势和 FLASH 非易失性的 STT-RAM，有庞大潜力逐步取代前三者。

铁电存储器[15]（Ferroelectric Random Access Memory，FeRAM）的工作原理是在施加一个电场给铁电晶体时，中心原子在晶体里会顺着电场的方向进行移动，产生的两个传统的的移动位置作为保存数据的“0”和“1”。施加电场击穿电荷，原子移动不同位置，设置存储单元；撤离电场，中心原子保持原状态不动，保存当前存储状态。

阻变随机存取存储器[16]（Resistive Random Access Memory，RRAM）作为一种阻性存储器，通过施加高低电压在金属氧化物上，使其呈现高阻态和低阻态，对应开启或关闭的电流流动状态，表示“0”和“1”状态。三星电子成功研发的 RRAM 可显著提高数据传输速度和耐久性，还可以大幅度降低使用的电流量，容量也有进一步扩展的可能性，在工业界产生了极大的反响。

聚合物随机存取存储器（Polymer Ferroelectric Random Access Memory，PFRAM）是一种基于聚合物塑料的非易失存储器，凭借着材料科学的三位堆叠技术具有很高的密度，从而达到大容量，且成本比 NOR 型 FLASH 降低约 90%。该技术成熟后，生产 PFRAM 可能会像印刷照片一样简单、快速且便宜。但是其缺点也十分明显，读写次数非常有限，且读取是破坏性的。

1.2.2 文件系统软件栈

文件系统依据不同的用户需求，可以选择直接访问（Direct Access，Dax）、O_DIRECT、O_SYNC、fsync 和 mmap 等不同的标志或操作，走不同的软件路径，一般无特殊设置的路径会经历最深的软件栈。Dax 挂载模式和 O_DIRECT 参数设置时会跳过页面缓存直接到达块设备层，但不会等待数据落盘后才返回；同步写 O_SYNC 参数会在数据持久化到存储设备再返回；fsync 操作主动将缓存在页缓存中的数据全部下刷并持久化到盘；mmap 通过将内核态的文件映射到用户态地址空间的方式，实现在用户态直接对文件数据的访问。

如图 1-1 所示，目前主流的文件系统都是在内核态中实现，当应用程序发起文件系统操作请求时，会触发对应系统响应，经由系统响应进入内核；初进入内核时，由统一的 VFS 层接口处理，再下发到具体的文件系统，由具体文件系统执行文件或目录操作。由于 CPU 无法直接访问传统机械硬盘和固态硬盘，需要先将文件数据拷贝到内存中，利用这部分内存作为缓存可以提升读写性能。不过有时需要考虑数据一致性和可靠性，会使用 Dax 等模式跳过页缓存直接持久化到盘，防止掉电数据丢失。从文件系统写入到盘的过程中，还需要经过通用块设备、I/O 调度等相关软件层，最终才到达盘。

这些复杂的软件栈时延对于慢速的硬盘设备是可忽略不计的，而对于高速的非易失存储器却占据了近半访问时延。为了充分利用非易失存储器的性能和特点，业界开发了一些新的访问模式以简化软件栈。第一种依然需要从用户态进入内核态 VFS 层，借助于非易失存储器可直接挂载在内存总线上的特点，以内存映射方式直接访问非易失存储器，跳过复杂的块设备相关层和页面缓存，直接写入非易失存储器。第二种方式的颠覆性更大，软件栈更加简洁，直接在用户态拦截文件操作，跳过内核态的

VFS 层，直接在用户态对非易失存储器进行访问，尽量简化访问路径深度，降低软件处理时间。但用户态文件系统仍然存在容易被内核程序乱写等安全性问题以及用户空间地址独立元数据管理和同步麻烦等问题未得到解决，仍处于研究阶段，是目前最新的研究热点。

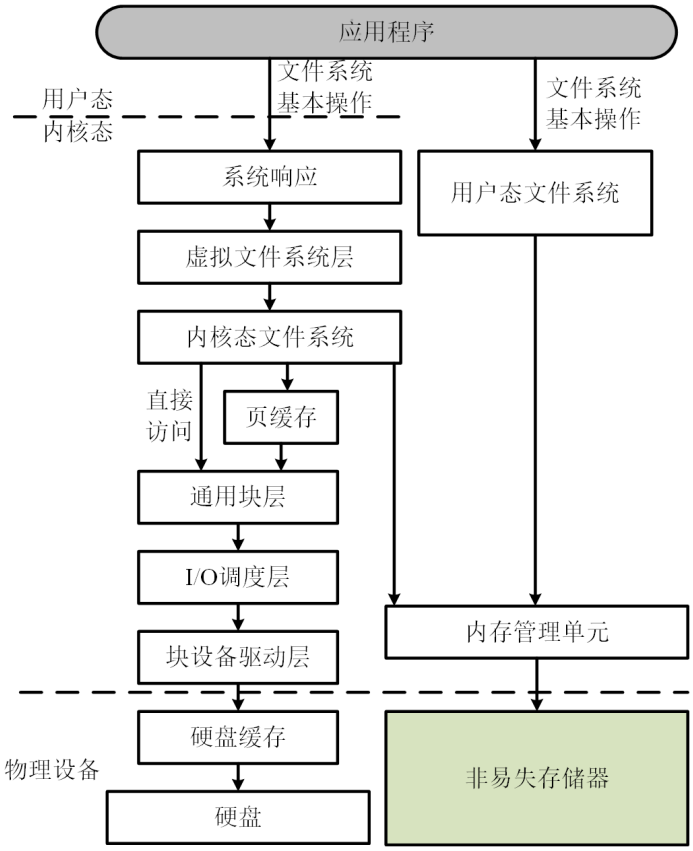

图 1-1　文件系统操作软件栈结构图

1.2.3　基于非易失存储的内核态文件系统

内核态文件系统借助内核提供的文件系统实现框架，能够较容易搭建起新的文件系统，由于不对原有的文件系统接口进行变动，已有的应用能直接适配使用，并且数据在内核态进行访问处理，对于数据的安全性也更有保障。现有研究如表 1-1 所示。

BPFS[17]和 PMFS[18]①是较早基于 NVM 的内核态文件系统的研究工作。BPFS 直

① Persistent Memory File System. https://github.com/linux-pmfs/pmfs

接挂载在内存总线上，使 CPU 访问文件系统跳过多余的块设备层，并提出了一个 epoch 的软件写屏障，保证 CPU 缓存保留脏数据的同时能按 epoch 顺序写回非易失存储器。BPFS 在硬件中提供一个简单的原子写原语实现原子更新，实现一种新的短路影子分页技术，结合非易失存储器就地修改的特性，只需拷贝部分数据就可完成更新操作，能加速更新，但移植性较差；此外，BPFS 借助 FUSE 框架辅助完成操作，一次操作带来两次内核态和用户态切换，反而增加了开销。

表 1-1　基于 NVM 的内核态文件系统研究

文件系统	主要思想	局限性
BPFS	epoch 机制保证写顺序性 修改硬件实现原子更新 短路影子分页技术减少写放大	修改硬件可移植性差 基于 FUSE 框架额外开销高
PMFS	提供硬件原语保证 NVM 写顺序性 元数据日志更新，支持元数据一致性 支持大的数据页，加快查找	未提供数据一致性保护 线性表目录组织元造成数据访问瓶颈
Ext4-Dax	维持原有 Ext4 特性 跳过页缓存层，保护一致性	软件栈过于复杂 基本没有利用 NVM 特性
SCMFS	修改并利用 MMU 管理 NVM 空间 文件预分配减少 MMU 函数调用开销 给文件分配连续地址空间简化访问	不支持文件系统原子更新 TLB 命中率低
NOVA	日志更新方式维护一致性 内存映射方式访问 NVM inode 为单位划分日志，提高并发 页面划分 NVM，不要求连续空间	索引结构在内存，文件系统重构速度慢 没跳过 VFS 软件栈和内核切换

英特尔研究的 PMFS 借助 NVM 挂载在内存总线的特性， CPU 通过 Load/Store 指令直接对 NVM 执行轻量级访问。PMFS 直接对 NVM 进行管理，跳过块设备层和页面缓存层来提供对非易失存储器的同步读写。PMFS 为了加快虚拟地址的查找速度，支持大的数据页结构，借此最小化旁路转换缓冲[19]（Translation Lookaside Buffer，TLB）的访问。PMFS 通过写元数据的回滚日志保护其一致性，但是未提供数据的原子性保护机制。PMFS 使用线性表的文件系统组织方式，导致了较差扩展性以及较慢的元数据访问速度。

Ext4-Dax[20]是对 Ext4 文件系统直接访问的扩展，文件系统组织方式和基本操作方式与 Ext4 相同，不过采用 PMFS 相同的 Dax 模式跳过了页面缓存，维护数据的一

致性。Ext4-Dax 使用元数据的写前日志机制保证了其原子更新，但是没有提供数据的原子更新保护。

SCMFS[21]复用了内存管理单元[22]（Memory Management Unit，MMU）以简化对 NVM 的空间管理，充分利用 MMU 内部的 TLB 和缓存硬件的支持进行加速。SCMFS 简化了管理和跟踪分配给文件空间的复杂数据结构，特别是大文件被切分成很多段进行管理的方式。为解决此问题，SCMFS 将整个文件系统放在虚拟地址空间，将文件整个映射到连续的地址空间实现简单和轻量级的访问，并对文件空间进行预分配，避免对内存管理函数的频繁调用。但 SCMFS 不支持文件系统的原子语义，且文件数量增多会导致 TLB 命中率降低。

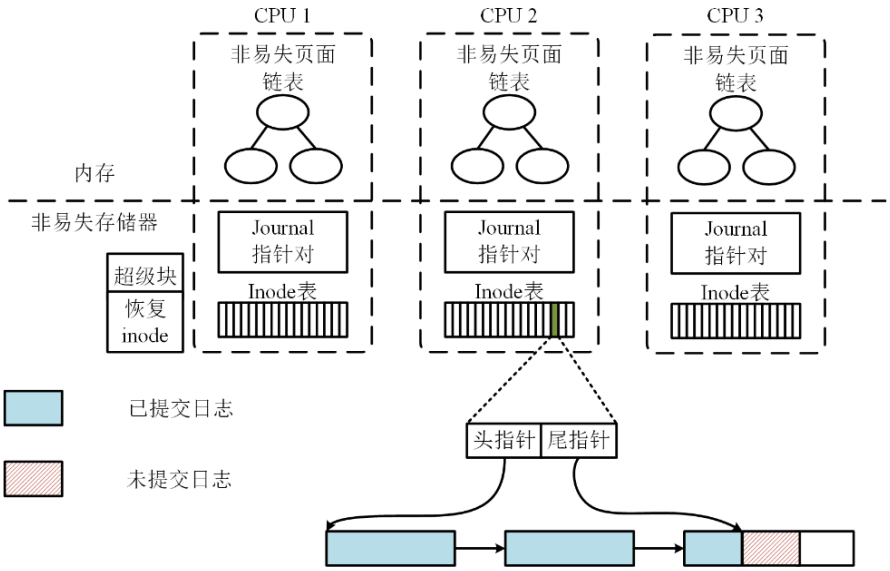

图 1-2　NOVA 数据结构图

NOVA[23][24]是一个加速版的日志型非易失存储文件系统，使用传统的日志型文件系统来充分挖掘非易失存储器的随机访问性能。如图 1-2 所示为 NOVA 的基本数据结构布局，NOVA 为每个 inode 维护一个日志 log 来提高并发性，日志页间以链表形式组织，无需为日志准备连续的存储空间。NOVA 不在日志中存储数据，日志 entry 指针指向实际的数据页面以减少日志大小，加快崩溃恢复速度。NOVA 在每个 CPU 核维护自己的页面链表、inode 表和多节点操作的 journal 指针对，减少全局锁的使用以缓解可扩展瓶颈。NOVA 实现的日志机制和轻量级 journal 机制，以恰当的顺序规则更新数据和元数据，使得可以很容易提供文件系统基本操作的原子性，同时使用一

些新的英特尔 clflushopt、clwb 等指令保证写顺序一致性。

1.2.4 基于非易失存储的用户态文件系统

早期基于内核态的文件系统的研究比较多，现在学者们的研究都转向跳出原有文件系统框架，把尽量多的文件系统操作都搬到用户态来，减少用户态和内核态切换的开销，进一步降低软件栈复杂性。现有研究如表 1-2 所示。

表 1-2 基于 NVM 的用户态文件系统研究

文件系统	主要思想	局限性
Aerie	实现用户态访问接口，简化软件栈 中心进程同步元数据	数据保护，权限控制困难 存在单点问题 元数据更新开销高
ZoFS	同权限文件集中管理 基于 MPK 技术权限管理 批量文件管理减少内核态交互	支持的权限组合受限于 MPK 权限更改会带来很高的开销
SplitFS	提供三种一致性模式选择 以 2 MB 粒度内存映射访问 提出 relink 技术实现追加写零拷贝	主要对 append 操作优化
Libnvmmio	实现用户态库日志保证一致性 支持多粒度的数据块更新 版本控制机制降低原子更新开销	依赖底层文件系统管理元数据 预分配文件空间与性能难平衡

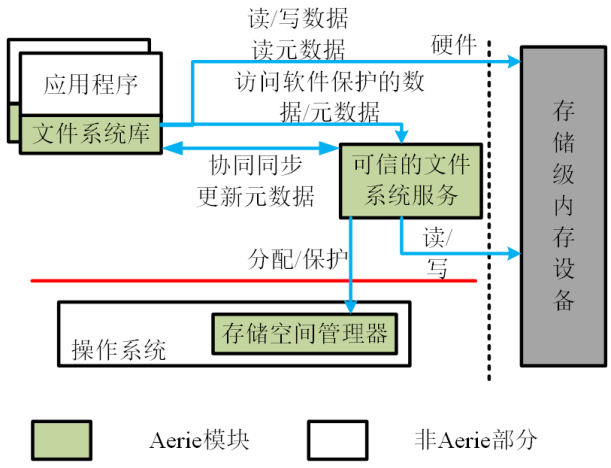

图 1-3 Aerie 系统结构图

如图 1-3 所示，Aerie[25]包括用户态库 LibFS 和用户态进程可信文件系统服务（Trusted File-System Service，TFS）两部分。LibFS 提供文件系统接口完成从存储级内存（Storage Class Memory，SCM）中对数据进行直接访问和对元数据的读取。TFS

则协调多个 LibFS 间元数据更新，负责从内核 SCM 管理模块获取分配的空间，并将其映射到 LibFS 地址空间。SCM 空间被暴露在用户态空间，Aerie 还提供了一个 extent 机制来对数据的访问控制进行保护。但是 TFS 与多个 LibFS 之间存在大量的进程间通信以维护元数据信息的一致性，对多并发数据访问不友好，还可能引起单点故障造成文件系统崩溃。

ZoFS[26]致力于在用户态访问 NVM 以提升文件系统性能，并提供对数据充分的隔离与保护。为此，ZoFS 提出了 Coffer 的新概念，以 Coffer 为单位管理具有相同权限的 NVM 数据页面。如图 1-4 所示，基于 Coffer，ZoFS 提出了新的 NVM 文件系统架构 Treasury，将 NVM 的空间管理和保护分隔开，用户空间 Lib 库完全控制管理该进程内的 Coffer 页面，一个 Coffer 一次只能被分配给一个用户进程访问控制。而内核模块负责 Coffer 的分配以及对用户进程的 Coffer 分配映射，并使用英特尔的内存保护键（Memory Protection Keys，MPK）技术实现对 Coffer 的权限进行管理。ZoFS 以 ioctl 的方式实现了一系列 Coffer 的分配、删除、扩大和缩小等操作，用户态 Lib 库只需少次与内核态的交互即可完成文件系统基本操作。ZoFS 缺点是不能同时控制数据和元数据，对元数据的间接更新会造成系统性能下降，而且受限于 MPK 技术支持的权限数量有限。

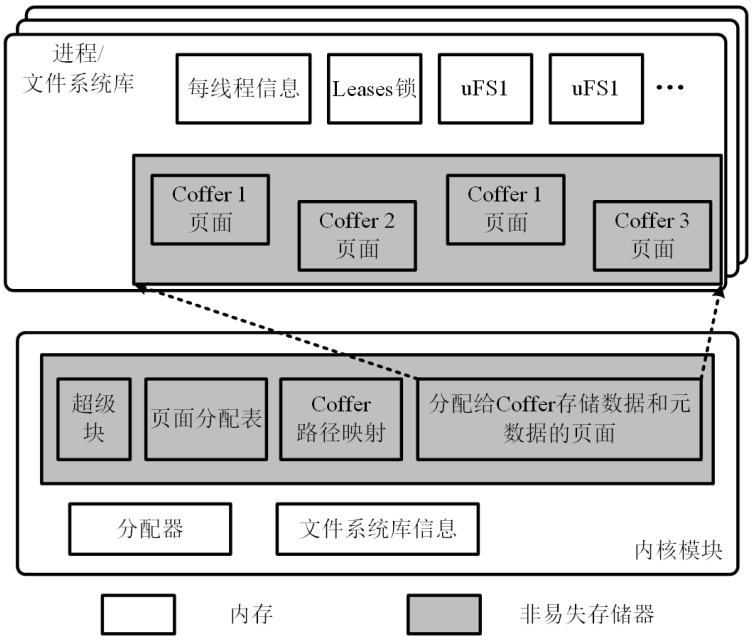

图 1-4　ZoFS 系统结构图

SplitFS 结构[27]如图 1-5 所示，分为用户态模块 U-Split 和内核态模块 K-Split 两部分。U-Split 处理 posix 文件系统所有的数据访问基本操作，将访问文件的 2 MB 空间通过内存映射到用户地址空间进行访问操作的处理。U-Split 将元数据操作重定向到 K-Split 进行处理，元数据的操作依赖于原有内核态 Ext4 文件系统。

SplitFS 支持 posix、sync 和 strict 三种一致性模式，posix 模式依赖 Ext4-Dax 实现的元数据一致性；sync 模式在 posix 模式基础上保证所有操作是同步的，但不能保证操作的原子性；strict 模式在 sync 模式基础上保证单个文件操作的原子性。SplitFS 优化追加写的 append 操作，提出 relink 的新概念，在执行 append 操作将多个物理块整合时，不同于传统方式需要将这些物理块拷贝到一个新物理块，SplitFS 会统一用一个逻辑文件将这些物理块组合在一起，只需要对元数据进行修改无需数据拷贝，在后续访问时经过适当的路由即可定位。SplitFS 局限于主要优化 append 操作，文件系统的其它操作性能相较于其它用户态文件系统没有优势。

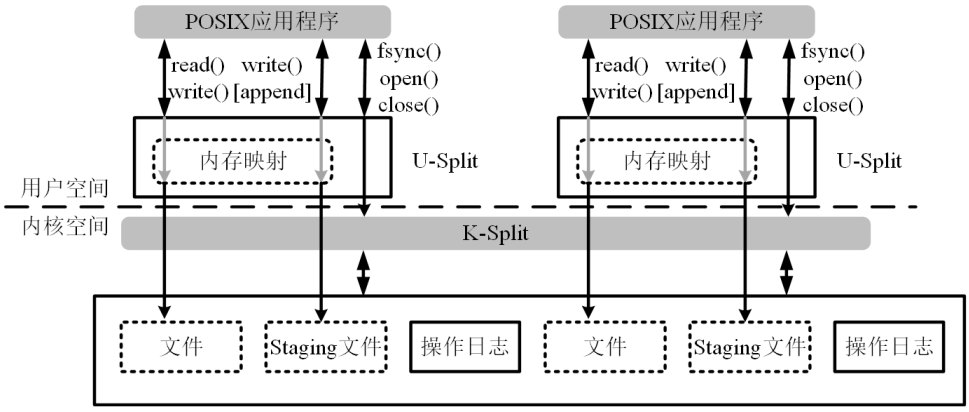

图 1-5　SplitFS 系统结构图

Libnvmmio[28]是 2020 年最新发表的相关的论文，旨在消除多余软件开销保证低时延、高并发的同时提供有效的数据原子更新。如图 1-6 所示，Libnvmmio 只实现了一个用户态库，内核态文件系统负责处理相关元数据操作；实现可伸缩的日志，将文件划分成可变大小的数据块，并为每个数据块创建一个日志 entry，支持数据块为中心的高并发日志。数据读写以文件内的日志 entry 为单位对映射到用户态的 NVM 日志数据块直接进行读写。日志模式 undo 和 redo 可选，会根据读写比例自动动态调整，同时只能存在一种模式。

Libnvmmio 提出版本控制的后台检查点机制，定时唤醒检查点线程来拷贝和清理日志 entry，将日志数据同步到底层文件。在应用调用 sync 操作时，要将日志转为提交状态，基于单调递增的版本号维护全局版本，此时不用同步将日志数据下刷到原文件，更新版本号即可返回，对用户响应影响很小。Libnvmmio 严格执行数据更新后再元数据更新的顺序，在崩溃恢复阶段，检查日志 entry 的版本号，小于全局版本号的数据块为已提交的数据操作，等于全局版本号则为未提交状态，并根据日志类型进行重做或撤销，同时使用多线程加速恢复过程。Libnvmmio 元数据由底层内核态文件系统处理，且对文件进行预分配和提前内存映射，在文件大小变动时可能会带来很多的内核态操作，反而增加了很多额外开销。

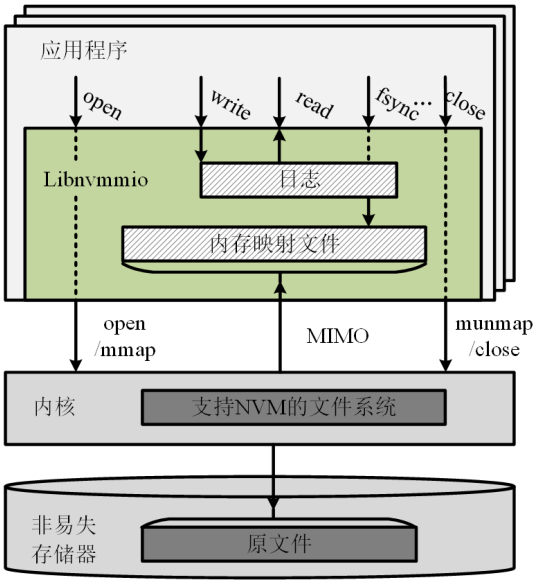

图 1-6　Libnvmmio 系统结构图

1.3　相关技术分析

1.3.1　英特尔傲腾非易失内存

在 2015 年，英特尔和美光联合发布了非易失的 3D XPoint[29][30]存储产品，其耐用性和访问速度超过当前 FLASH 闪存约 1000 倍，使数据存储迎来了重大性能飞跃。Optane（傲腾）NVMe[31]固态硬盘 Clodstream 性能和 Optane 内存（Apache Pass，AEP）作为首次发布的基于 3D XPoint 的产品，其价格和性能在 DRAM 和 NAND 闪存之间，如图 1-7 展示了其在存储设备的位置。

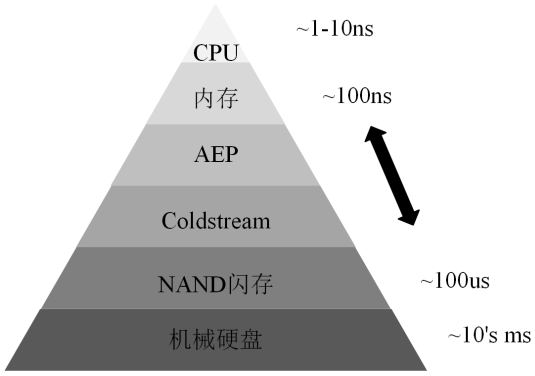

图 1-7　存储设备时延图

Clodstream 在数据传输效率上实现了跨越式提升的 NVMe 协议，是世界上访问速度最快、可用性和可服务性最好的固态硬盘。AEP 是基于内存访问原语 Load/Store 指令的高性能和高灵活性而设计的革命性非易失存储内存产品。AEP 是本文使用的文件系统底层存储介质，其主要特点如下：

（1）以字节方式进行寻址，具有非易失性，单条容量可达 512 GB，无需先擦除再写入。

（2）在 CPU 内部的内存控制器支持下，使用 DDR4 接口，支持 CPU 缓存行的访问，单 CPU 支持最大容量 3TB 的 AEP。

（3）可以使用 Dax[32][33]的方式进行访问，允许应用程序直接使用 Load/Store 指令读写，即文件系统通过内存映射直接对 AEP 进行访问。

（4）英特尔专门为其开发了 PMDK[34][35]①项目，提供了很多类型的库供开发者使用高级语言对 AEP 进行访问。

随着 AEP 技术提升导致容量不断增大，现今研究方向已将 AEP 从配合 DRAM 和 SSD 作为元数据或数据缓存的多级存储方式，转向作为对数据可靠性和读写性能要求很高的应用系统的持久化存储。

1.3.2　原子操作更新

非易失存储器以字节方式进行寻址，没有实现对上层应用的多字节原子访问。而现代处理器也只提供 64 位对齐的缓存行大小的原子操作，对于现在一般文件系统的

① Persistent Memory Development Kit. https://github.com/pmem/pmdk/

复杂操作和更大的数据原子更新是远远不够的。比如现在执行一个 4 KB 或更大的文件写操作，如果中途发生系统崩溃或掉电，导致只完成了部分的数据更新，需要花费很大的力气和代价检测部分更新的数据块并加以纠正。为了在数据库、文件系统等重要应用中避免部分完成的非原子更新错误，研究者们提供了一些能保障原子更新的技术，这些技术以不同的方式保证更新的原子性，在性能方面也是各有千秋，具体特性如下：

写时复制（Copy on Write，COW）或**影子分页技术**（Shadow Paging）。写时复制机制基本流程是，用户更新使得要对一个数据块执行更新操作，第一步分配一个新的空闲数据块，第二步将原数据块的数据拷贝到新数据块，最后再把要写的新数据计算偏移，写到新的数据块；写时复制进行异地更新而非对原数据块执行就地更新。写时复制技术不仅能保证数据更新原子性，也可以对索引执行异地更新操作。写时复制在文件系统基于树结构进行索引组织时，会产生漫游树问题[36]（Wandering Tree Problem），目录或文件的创建、移动和删除等会更新索引结构，原本改变的子节点更新后，其父节点异地更新关于子节点的索引，父节点的父节点再迭代传播直到根节点更新完成才结束。写时复制的使用给文件系统带来了很多额外的软件开销和数据拷贝。第一，它急剧地增加了写放大，在只有数字节大小的更新操作下也需要以页粒度大小执行。第二，写时复制技术会使 TLB 缓存失效，每次更新都会指向新的数据页面，使得本地 TLB 数据每次都要下刷，并且还要进行处理器间通信通知失效。

写前日志[37][38]（Write Ahead LOG，WAL）。又称 Journaling 机制，被广泛使用在数据库和文件系统场景来保证数据和元数据的原子性和一致性。它在更新原文件之前先维护一份新数据或旧数据的拷贝，如果文件系统在更新中失败，则可以用新数据或旧数据的日志进行重写或撤销操作，恢复一致性。两种日志策略可供选择，重做日志和回滚日志。重做日志机制先将新数据写到重做日志持久化，后期再找时机覆盖更新原文件的数据。如果覆盖原文件的写操作失败，可以用重做日志中的数据重新执行提交。对于读操作，则要先去重做日志查找最新的数据。回滚日志机制则是先将旧数据拷贝到回滚日志持久化保存为副本，再对原文件执行覆盖写操作就地更新。如果覆盖原文件的写操作失败，则可以拷贝旧数据回去，回滚此次更新操作。回滚日志下读

操作，可以直接在原文件访问最新的数据。两种日志机制都要执行两次写操作，一次日志，另一次原文件，同样会增加数据拷贝开销。重做日志可以在后台非文件操作的关键路径，对原文件更新，在一定程度上可以减轻对性能的影响。

日志结构文件系统[39][40]（Log-structured File System，LFS）。日志结构技术最开始以牺牲部分读性能，来换取随机写转换为硬盘高性能顺序写的代价为设计目标，后面被借鉴于实现原子更新。日志结构以追加写日志的方式，顺序地在整个的数据块后面进行元数据或数据更新，再借以跳表、前缀树、B+树等查找树结构，提高读取速度，实现读写访问。每次数据更新操作时，将新数据信息写到日志尾，然后更新日志尾部指针作为更新完成标志，再将旧数据置为失效待后台回收，即完成此次更新操作。日志结构与写前日志相比，只需要一次正常的数据拷贝没有额外的同步开销，但是需要借助辅助的数据结构来加速读取写过的数据。日志需要进行旧数据的定期清理和日志的合并，这会增加额外的软件开销和拷贝，会在一定程度上，降低系统性能。当前研究了一些方法减轻日志垃圾回收的影响，比如使用多日志的方法增加访问和回收的并行性、冷热文件识别方法优先回收很少被访问的文件，减少应用和后台垃圾回收对文件的访问冲突等。

1.3.3 直接访问与内存映射

传统的文件系统访问和直接访问、内存映射访问的本质区别在于 I/O 路径不同，如图 1-8 所示。一般的文件系统访问模式，会先通过系统调用进入内核态，在 VFS[41] 层处理下，到达具体文件系统，将用户态数据写入页缓存，后再将数据从页缓存写入 NVM，或从 NVM 读取数据到页缓存，再从页缓存拷贝数据到用户态内存。传统访问方式要从用户态经过内核态，需要将数据从用户态内存拷贝到内核态内存，才能写入 NVM，带来了多一次的数据拷贝。而直接访问模式，可以直接跳过页缓存，将数据直接写入 NVM 或者从 NVM 直接读取数据到用户态内存。一般的内存映射 mmap 模式，先对文件进行映射系统调用，将文件的全部或部分内容映射到用户态一段连续的地址空间，将文件数据与内存地址联系起来，之后对该文件的操作都直接以对这段地址的内存访问方式进行操作，不走原来 VFS 层的结构；不过此时的内存映射依然会先写入页面缓存，会带来性能和一致性影响。而现在的技术将直接访问和内存映射

进行结合，在用户态以内存方式直接访问非易失存储设备，不经过页缓存，也不经过内核态，直接一次拷贝非易失数据到用户内存，实现更精简的软件栈。

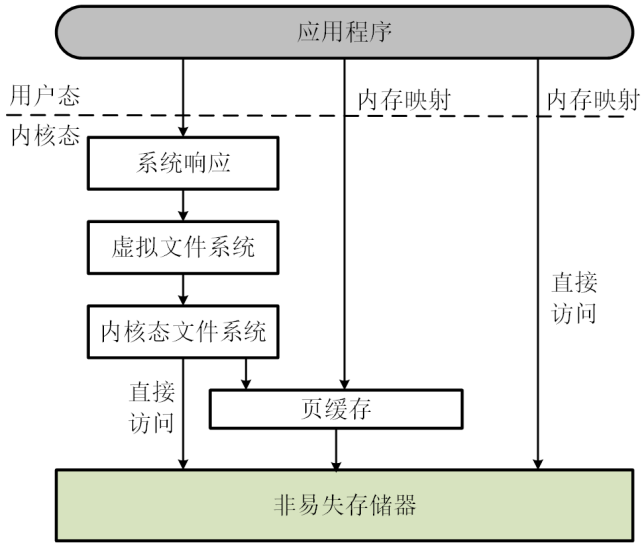

图 1-8　直接访问与内存映射软件栈图

1.4　研究内容与论文结构组织

在当前大数据背景下，数据存储的需求不仅仅是大容量的问题，更要追求极短的用户响应时间、极大的带宽和极高的数据可靠性。现今遍布数据密集型应用，文件系统是数据存储至关重要的一环，非易失存储设备的出现，给当前性能和安全性已经很成熟的文件系统带来了新的革命性优化思路和挑战。

本文致力于构建一个基于 NVM 的用户态文件系统 UnvmFS，借助 NVM 通过内存总线访问的特性，在用户态空间以内存映射方式直接访问 NVM，跳过复杂的 VFS 软件栈和页面缓存，精简访问路径以提高性能。UnvmFS 以日志结构的形式访问进行记录，并借助原子更新等技术，解决字节访问的 NVM 在文件系统原子语义下的局限性，实现 POSIX 访问标准的兼容和数据一致性。UnvmFS 还专门设计了非易失分配器来管理元数据和数据，消除元数据同步问题，并对元数据持久化管理。

本文的组织结构分为以下五个章节：

第一章，介绍了本文在当前时代的研究背景与研究意义，接着讲述了现在国内外学业界和产业界对于 NVM 和基于 NVM 的各种文件系统的研究现状；然后分析了

AEP、原子更新、内存映射等关键技术。

第二章，是本文的系统设计部分，对文件系统与 NVM 结合存在的潜在问题进行了分析，指明了本文的设计目标；先给出了系统整体结构的设计，再分别介绍了本文文件系统的组织结构、空间布局、操作流程以及页面分配和垃圾回收的具体设计。

第三章，对本文文件系统具体实现细节进行阐述，描述系统用户库和内核模块的主要结构和功能，接着对各个功能模块的功能和实现细节进行了详细说明。

第四章，本章列出了当前系统的测试环境，并用文件系统测试工具 Filebench 对多种负载进行了测试，根据对比测试结果，给出了具体分析和总结。

第五章，对全文进行了总结，并对当前的系统存在的不足提出了改进想法，并为之后的研究工作做出了展望。

2 基于 NVM 的用户态文件系统 UnvmFS 设计

在本章中，先抛出当前文件系统亟待解决的软件时延、数据一致性、原子性以及用户态文件系统的中心化同步等问题并进行相应分析，接着介绍本文用户态文件系统 UnvmFS（Non-volatile Memory File System in User Mode）的整体架构及详细设计。

2.1 现有文件系统在 NVM 上的问题分析

（1）复杂软件栈高时延开销问题

现今的应用系统都使用传统的文件系统操作接口，从用户态系统响应接口进入内核，经过复杂的 VFS 层，再进入具体的文件系统进行读写；VFS 为上层应用和下层文件系统提供了统一的接口，起到适配的作用。然而不同于在慢速的硬盘下的读写，冗长的软件栈以及中断、用户态内核态线程切换等一系列开销在高速的非易失存储器下都是不可忽略的，继续走现在的软件栈无法充分发挥出非易失存储器的性能。内存映射的出现，通过将 NVM 空间映射到内存地址空间的形式，直接访问 NVM，可以跳过繁杂的软件栈处理，减少拷贝次数，极大地降低访问时延。

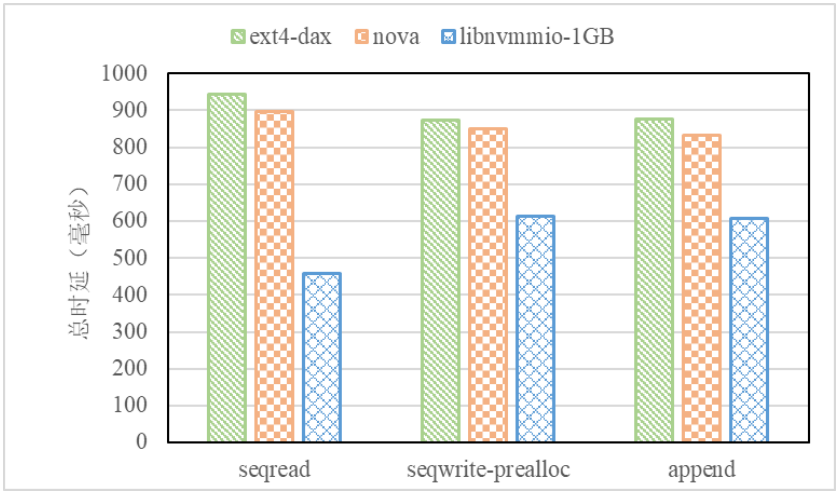

图 2-1 文件系统读写时延对比测试

在传统文件系统 ext4-dax、内核态 NVM 文件系统 nova 和用户态 NVM 文件系统 libnvmmio 三种场景下，对单个文件以单线程形式进行 1 GB 数据的顺序读写和追加写，间接对比去除冗余软件栈和内核线程切换开销对性能提升的效果。如图 2-1 所

示，libnvmmio 顺序读时延比 ext4-dax 降低约 52%，顺序写时延降低 30%。libnvmmio 文件系统的顺序读时延比 nova 降低 49%，顺序写时延降低 28%。libnvmmio 在默认给文件分配预分配 1 GB 空间时，文件从头至尾追加写和顺序写时延基本一致，在预分配空间充足无额外处理操作时两者性能相当。观察结果可知，nova 相对 ext4-dax 采用内存映射的方式访问非易失存储器，能够跳过块设备层的软件栈，并减少一次用户态缓存到内核态缓存和一次由盘到内核态缓存的拷贝，从而取得更高的性能。libnvmmio 相对 nova，直接在用户态对非易失存储器读写数据相比传统经过内核线程切换和 VFS 的路径再到底层存储设备的方式，性能获得了进一步提升。

（2）额外内核态操作增加时延问题

已经验证 libnvmmio 每次文件创建直接预分配 1 GB 的存储空间，再将 1 GB 文件空间内存映射到用户地址空间，能得到较好的访问性能；但文件初始创建时就要预分配大额度的空间存在一定的存储空间浪费，而且减小预分配会因频繁进入内核的元数据操作增加很多额外时延，预分配空间的度难以把握。预分配空间指数式减小预示着附加的元数据处理操作次数呈指数式的增长，额外时延也会随之急剧增长；增加的额外操作时延甚至会超过原有内核态文件系统的软件栈所带来的时延，不容小觑。除此之外，libnvmmio 的数据空间使用量在无预分配时也是一般文件系统两倍，因为其要保存一份日志数据和一份原文件数据，且日志数据不能定期清理，否则要通过原文件处访问数据。

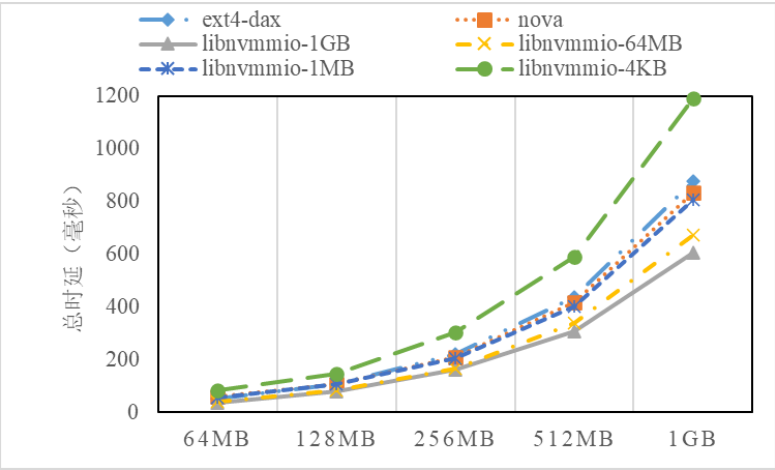

图 2-2　libnvmmio 多粒度预分配追加写时延对比

在 libnvmmio 预分配不同粒度存储空间时，对单个文件单线程追加写时延进行测试，预设以 4 KB 粒度追加写至文件达到 64 MB 至 1 GB 大小，结果如图 2-2 所示。从图中可以明显感知到随着预分配空间缩小，追加写时延逐步增加；在 4 KB 预分配空间时，基本追加写时延都要高于 ext4-dax 和 nova；而在 1 MB 预分配空间时，追加写时延已经与 nova 和 ext4-dax 接近。

究其原因，在于 libnvmmio 借助于底层内核态文件系统创建文件，在用户态以日志形式记录更新，对原文件预分配空间后内存映射到用户地址空间，后期以内存拷贝的方式在后台下刷数据到原文件。预分配空间对追加写埋下问题，libnvmmio 在对文件进行追加写，文件大小增加到预分配空间不足时，要先释放原文件的内存映射再重新分配更大空间重新映射以及对相关元数据的重新分配与释放，一次追加写操作可能涉及到多次进入内核的操作，元数据处理时延反而超过数据拷贝。以 4 KB 预分配空间为例，以 4 KB 的粒度追加写 1 GB 的数据，从第二次追加写开始，每次都要重复原文件的额外内核态处理操作，总共为 220-1 次。libnvmmio 总要平衡性能与存储空间使用的问题，预分配空间的大小直接影响到内核元数据处理次数进而成倍增加访问时延。且现在的 NVM 容量相较硬盘还是有一定差距，要尽量减少空间浪费。

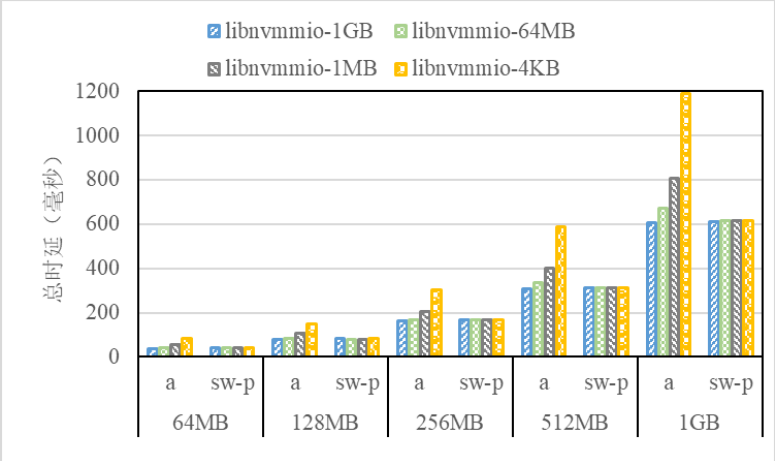

图 2-3　libnvmmio 多粒度预分配追加写与顺序写时延对比

继续从图 2-3 中观察，libnvmmio 在各预分配空间下，文件从头到尾的顺序写和追加写之间性能差异。此处测试的顺序写是已经生成对应文件大小的文件顺序写，libnvmmio 不会有额外预分配空间不足的操作。随着预分配粒度的减小，顺序写和追

加写间性能差异也逐渐拉大，到 1 GB 文件时 4 KB 预分配空间时追加写时延已经接近顺序写一倍。随着对文件追加写总量增加，不同预分配空间时延差距也在拉大，说明 libnvmmio 预分配空间方式对于大文件追加写场景极不友好。单独观察各文件大小下，顺序写在不同的预分配粒度性能表现基本一致，从图 2-2 已知 libnvmmio 顺序写表现优于 ext4-dax 和 nova。原因是 libnvmmio 顺序写未改变文件大小，就地更新在日志上，满足在用户态直接数据更新，无其它复杂软件操作的条件，这是用户态文件系统应有的性能追求，但 libnvmmio 在多数场景下并不能达到此条件。

（3）用户态中心化元数据同步问题

libnvmmio-1GB 与 nova 和 ext4-dax 的性能对比可以看出，文件系统直接在用户态对 NVM 中的数据读写能够有效降低访问时延，提高系统的性能。出于对性能的考虑，基于 NVM 的文件系统访问迁移到用户态已经是必然。但是现在已有的一些用户态文件系统的研究工作，总是被用户态进程地址空间相互独立的特性所局限。比如 libnvmmio 需要借助于内核态文件系统管理其元数据来实现文件系统元数据同步，意味着 libnvmmio 的元数据操作每次都要经过内核态进行处理，在数据和元数据操作的混合访问负载下，libnvmmio 的性能会大打折扣，这是 libnvmmio 的另一局限。

还有一些其他用户态文件系统在被多进程访问时，常采取的措施，或者是统一的内核态管理模块，或者是用户态进程管理服务，总是需要一个单独的中心化内核模块或用户态进程进行元数据的同步操作，来维护文件系统状态的一致性。不管是元数据处理在内核态模块还是用户态进程的同步通信，都带来了一定的时间开销，降低了系统并发性，而且增加了系统设计的复杂性。如何解决用户态地址空间独立，消除同步开销，实现文件系统去中心化的问题也是本文重点之一。

（4）文件系统一致性维护问题

一个优秀实用的文件系统绝不仅仅是依靠其优异的读写性能和高并发；如何能在高并发或是一些崩溃、掉电等异常情况下能保证数据的不丢失，提高文件系统可靠性也尤为重要。一般的硬盘文件系统还会带有页缓存功能，旨在提高访问速度，减少对块设备和块设备层的访问，但是也因此带来数据不一致的安全性问题和之后的一系列同步操作开销，这也是设计基于 NVM 的文件系统时要考虑的问题。内存性能这

么高，内存计算的概念被追捧，但其易失的缺点使得其即使在以后的技术革新下容量和价格接近硬盘，也始终都不能取代硬盘成为新的外存，只能一直作为缓存存在。在使用与易失内存性能差异很小的非易失存储器时，页缓存只会带来冗余的拷贝和数据不一致性，带来性能的下降。作为缓存的内存使用实无必要，于是，跳过传统的页缓存机制，直接对设备进行访问，来保证文件系统可靠性，是可行的方式，这是本文的优化之一。

当然，NVM 能解决页缓存的一致性问题，自身按字节存取的特性，也带来了新的数据非一致性问题。现行的 POSIX 标准文件系统接口要求保证操作的原子性，现今已有的应用也依靠文件系统原子性来保证程序运行的正确性。在使用 NVM 时，只能保证单字节的原子性，即使借助 CPU 缓存特性也只能保证缓存行大小的原子性，如何对动辄千字节、兆字节的文件系统操作保持原子性，兼容应用的原子语义需求也成了必须解决的问题之一，原子更新可采用的技术可以参考 1.3.2 节。

为了解决以上问题，本课题设计提出了一种去中心化的基于非易失存储器的用户态文件系统，具体设计见设计部分。

2.2 用户态文件系统 UnvmFS 整体设计

2.2.1 设计目标

本文目标是设计一个跳过一般文件系统复杂软件栈，无元数据同步，去中心化，全用户态访问的非易失存储器的文件系统。遵循以下设计目标和实现策略：

（1）低时延。文件系统性能放在第一点。UnvmFS 经内核模块注册及初始化，后将整个非易失存储器空间映射到用户态进程内存地址空间，文件系统操作通过用户态库直接访问 NVM，元数据和数据访问都在用户态，精简软件栈，达到低时延目标。UnvmFS 的内核模块，只负责处理基本初始化及挂载、内存映射和垃圾回收等后台功能，之后的文件系统操作全在用户态完成。

（2）原子性。POSIX 标准和可靠性要求文件系统操作语义的原子性。UnvmFS 是以日志结构的方式进行访问记录更新的，采用记日志的方式实现文件操作的原子性。当更新日志尾指针后，操作才算完成提交，返回操作成功给用户态；否则该操作

回滚，返回操作失败给用户态。对于目录操作，会使用轻量级的写前日志机制，拷贝极少量元数据，实现操作失败后的正确回滚，维护原子性。

（3）高并发性。提供细粒度日志，为每个 inode 节点创建一个单独的日志。日志由非连续的数据块以链表链接，不要求日志存储空间的连续性。各用户侧线程处理不同文件时可以并发执行，互不干扰，对其它线程透明。使用读写锁，同一文件读操作可并发，实现文件的并发读。设置每个 CPU 的本地资源分配结构，提高对资源获取的并发性。

（4）去中心化和消除元数据同步开销。UnvmFS 将 NVM 空间内存映射到用户态地址空间实现直接访问；设计非易失分配器，将本要中心模块负责同步的元数据通过分配非易存储失空间直接持久化，之后各进程直接通过 NVM 的映射基址和地址偏移即可访问到元数据，从而去除中心模块并消除同步开销。元数据持久化后还能减少文件系统掉电后恢复和正常系统关闭重启后的元数据重构的时间。

（5）完整性。UnvmFS 自设计非易失分配器持久化元数据，在用户态实现元数据和数据的管理与访问，不用借助已有的内核文件系统进行元数据管理和同步。

2.2.2 整体结构

为了充分利用 NVM 性能，并适配原有 POSIX 标准接口，本文设计了一种全用户态访问的基于非易失存储器的日志结构文件系统 UnvmFS。设计总体结构如图 2-4 所示，UnvmFS 分为内核态模块 UnvmFS-Kernel 和用户态库 UnvmFS-Lib 两部分。

（1）内核态模块 UnvmFS-Kernel。鉴于之前的文件系统将元数据操作都放在内核态带来同步的开销和性能扩展的局限性，本文内核态模块仅承担最基本的初始化和后台工作。内核模块对文件系统 UnvmFS 进行注册，响应用户挂载命令，完成对 NVM 的初始化工作，初始化超级块及相关元数据资源。用户态库初始化时，向内核态模块发送内存映射请求，以内核态模块修改后的内存映射的方式将整个 NVM 空间映射到该进程地址空间，保存映射基址。除此之外，再无内核模块和用户态进程间的交互，内核模块在后台维护整个文件系统的垃圾回收工作，为页面池提供充足的非易失页面。

（2）用户态模块 UnvmFS-Lib。用户态库提供 POSIX 标准文件系统访问接口，

拦截来自用户的文件系统操作，将文件系统操作从原有进入内核处理的路径转向UnvmFS 用户态库执行；用户态库直接处理包括目录、权限、读写等全部文件系统操作，所有操作更新都在用户态直接访问 NVM 来完成。在用户态库第一次响应用户文件系统操作时，需要先执行简单的初始化工作，首先将整个 NVM 空间映射到用户态进程地址空间，再对前缀树、页面池、链表等基本资源分配结构全局访问指针初始化为 NVM 空间在该进程对应的映射地址位置，方便后续操作直接访问本地资源。

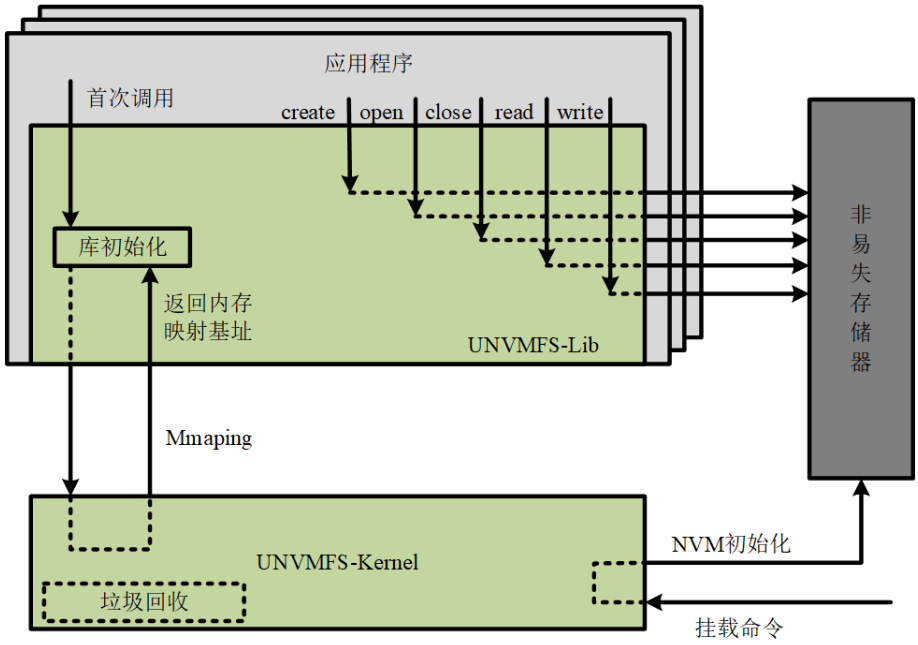

图 2-4　UnvmFS 文件系统整体结构图

2.3　用户态文件系统组织结构与布局设计

本节将以文件系统空间布局开始描述，用户态多进程以偏移地址方式，在不同的地址空间共享整个非易失存储空间；接着介绍文件系统元数据和数据的组织结构，提供高效简洁的目录组织和文件定位功能；最后描述索引用的前缀树的设计结构及其叶子节点存放的内容。

2.3.1　文件系统空间布局

整个 NVM 的空间划分如图 2-5 所示，切分粒度以页面为单位，数据页面以 Linux 系统常用的 4 KB 为标准。

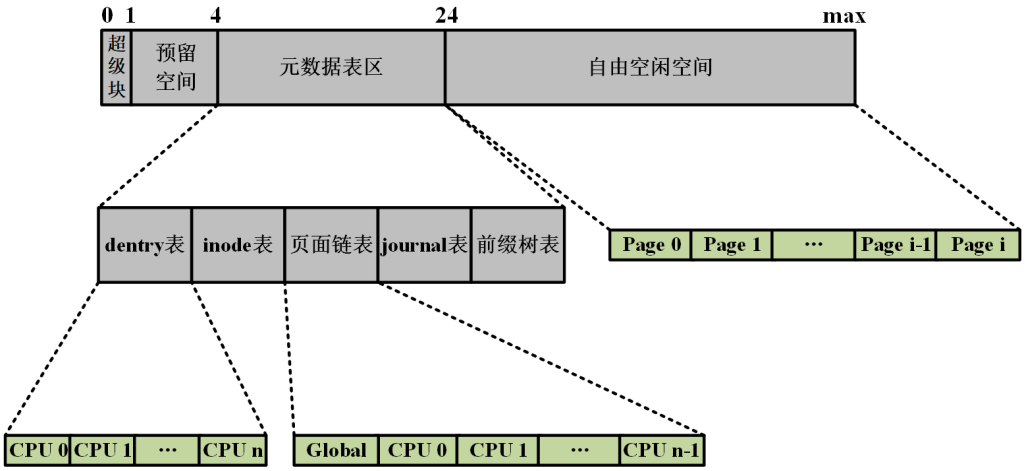

图 2-5　文件系统空间布局图

超级块分配偏移为 0 的一个页面，超级块实际大小以其数据结构占用空间为准，不要求填满整个页面。预留三个页面，维持 4 个页面对齐的格式。自由空闲空间切分成页面粒度的块，挂载在全局池和局部链表，设计相应的页面分配算法供给整个文件系统。元数据表区的 dentry 表、inode 表、页面链表、journal 日志表和前缀树表的头部结构各占据 4 个页面空间，能支持较大量的服务器 CPU 核数。其中 dentry 表、inode 表、journal 表和前缀树表，按照系统初始化获取的当前 CPU 核数划分各表的本地链表头部结构所在偏移位置，之后线程需要获取这些资源时在其所运行的 CPU 本地链表获取即可，可以最大实现资源获取并发，减少加锁带来的冲突。inode 链表在最开始还分配了一个全局链表池，自由空闲空间的页面初始全部挂载在该链表，之后本地页面链表空间不足时从此处批量获取页面，获取数量由页面分配机制自动动态控制。

2.3.2　文件系统索引组织

本文文件系统的索引组织，以哈希和前缀树结合的形式，对目录、文件或数据页面进行索引查找。每个目录和 inode 的路径被哈希为一个 32 位的有符号整型文件描述符（File Description，FD）；超级块、目录 entry 和 inode 各自维护自己的前缀树，以 FD 为前缀树的键值组织前缀树，查找目录或 inode 时提供 FD，查找文件内数据页面时提供页面在文件的偏移即可快速查找定位。

（1）元数据组织结构

元数据组织结构如图 2-6 所示，目录或文件创建时，路径哈希成前缀树键值，dentry 或 inode 挂载在叶子结点，之后采用本文设计的前缀树的方式进行索引，效率不低于哈希方式的索引。各级目录在创建时，都有挂载在超级块所在的前缀树的叶子节点上，只需查找一次前缀树即可找到所有目录的 dentry 结构，再通过对目录上前缀树的一次查找即可定位到所要查找的文件 inode 结构。不只是文件 inode 结构，当前目录的下一级目录 dentry 结构也会挂载在当前目录的前缀树的叶子节点，方便之后的目录操作的开展。

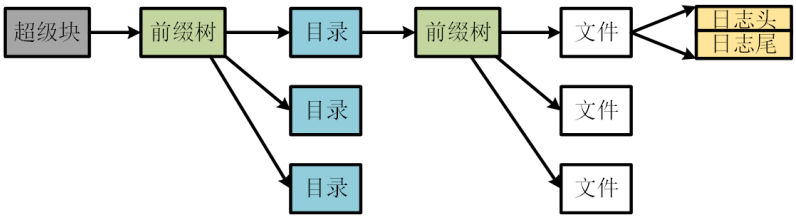

图 2-6　元数据组织图

（2）数据组织结构

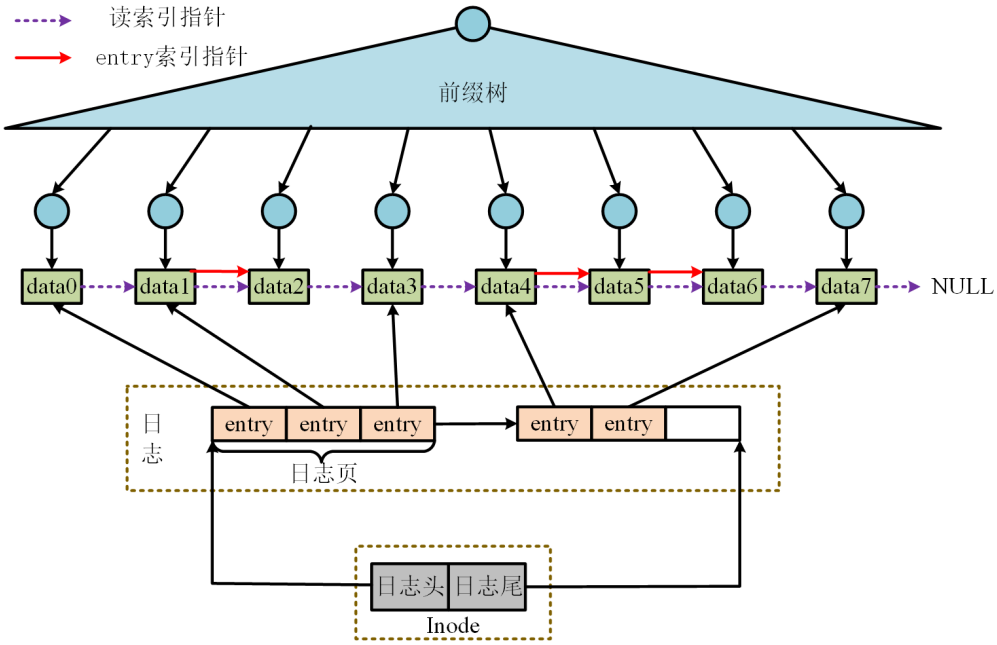

图 2-7　数据组织图

每个文件 inode 节点都有自己单独的日志，提供文件间高并发的访问模式，而无需额外的同步操作开销。如图 2-7 所示为单个文件 inode 的组织形式，inode 维护一组日志头指针和尾指针，供更新操作从日志尾进行数据更新，供垃圾回收操作从日志

头开始回收。由于本文把 NVM 空间按页面进行划分与分配使用，在日志页使用完时，每次为文件日志分配固定页数的页面，日志页面间以简单的链表形式进行链接；该方式使日志空间分配更加简单，不需要分配大块连续的存储空间，大大降低非易失存储器的内存管理分配的难度。

inode 日志以日志 entry 的形式保存数据更新的元数据信息，元数据包括以指针的形式描述的实际数据页面的存储位置。数据更新时从本地页面链表分配新的数据页面异地更新，在日志尾写入新的日志 entry，最后更新日志尾指针完成一次更新，更加简单高效，没有更新的数据量变化带来的重复的拷贝工作。此外，在垃圾回收时，回收日志页面只需回收旧的 entry 信息，拷贝工作更少，以页面的粒度进行回收，可以不回收整个日志；对于无效的数据页面，只需一些指针赋值操作即可回收进本地页面链表或全局页面池，步骤简单快捷。

日志以顺序模式进行更新，每次从日志尾开始更新，因此链表也不会影响更新效率。然而遍历链表的方式，对文件进行读取的方式浪费大量时间与 CPU 资源，因此采用两种措施优化读操作。一是，增加前缀树索引，将每个数据页面挂在前缀树叶子节点，加快对读取操作首页面的查找速度；二是，如图中紫色虚线箭头所示，增加文件内数据页面链表，本文称之为快读链表，索引到首页面后，直接向后遍历链表读取后面的数据，而不用每个页面重复进行前缀树索引；针对大块的文件读操作，按照每个页面读取都要对前缀树重复查找的方式性能不友好；本文快读链表对每个数据页面设立一个 next 指针指向其下一个页面，每次读取数据，查找一次前缀树，即可通过 next 指针顺序查找到其它页面。设计上不将前缀树指向日志 entry 而直接指向数据页面可以简化查找层次和流程，增加的快读链表也解决了指向数据页面的前缀树单个页面查找的问题。

2.3.3 前缀树结构

前缀树作为一种多叉树结构，常用于字符串的快速检索，相较于 B+树、红黑树等其它的一些自平衡二叉树或多叉树结构，前缀树基于序列前缀相同的特点，规则比较固定，没有一些额外增加加锁时间，影响并发的旋转等自平衡开销。在文件系统组织结构，以目录名、文件名作为前缀树节点键值，以目录或文件路径为全局键值进行

查找似乎是很好的选择。但是，采用此种方式前缀树可能会随着目录深度加深，而使得树的层数增加，影响到查询效率。前缀树宽度不影响查询效率，于是本文决定增加前缀树宽度，牺牲部分空间，换取查询的速度。

如图 2-8 是本文的前缀树索引结构，采取固定 4 层的前缀树，提高查询速度且简化实现。前缀树的键值总长 32 位整型值，每层取 8 位，每个节点拥有一个指向 256 个节点的桶；目录及文件路径字符串通过 CRC32 哈希算法转化成唯一的 32 位整型值，记为 FD，作为前缀树键值；至于文件内部数据页面索引，则将文件字节偏移左移 12 位，取文件偏移的 12-43 位也即页面偏移，作为前缀树键值。前缀树的中间节点不存储数据，保存指向下一节点的偏移地址。叶子节点则根据前缀树类型保存 dentry 结构或 inode 结构或数据页面的地址偏移值。

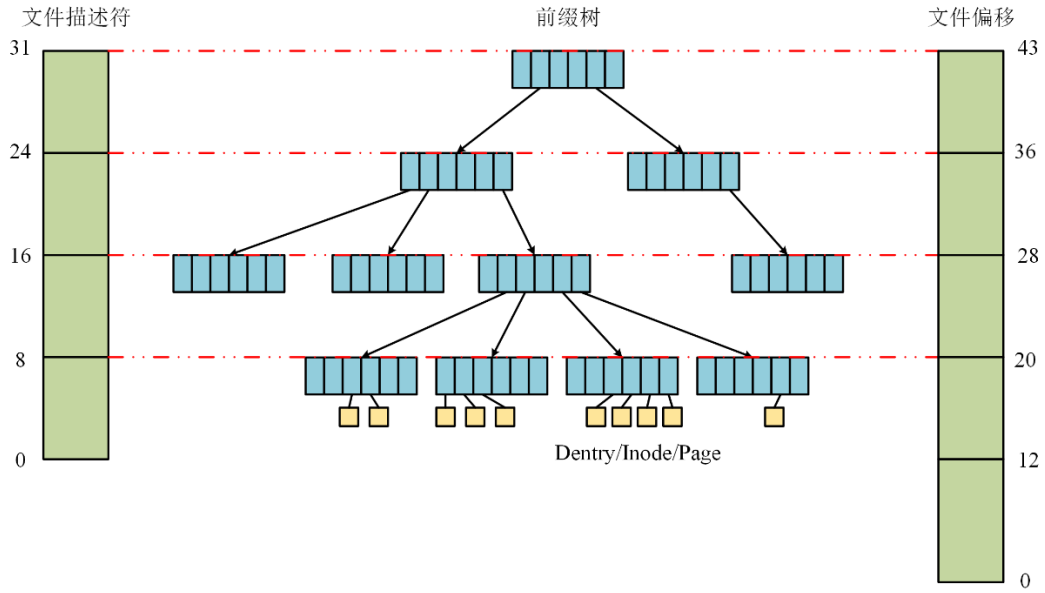

图 2-8　前缀树结构图

2.4　原子更新操作流程设计

POSIX 标准文件系统语义要求保证文件系统操作是原子的，操作状态只有成功或不变，很多应用借助文件系统操作原子性来保证它们程序执行的正确性，原子语义是文件系统的必要属性。UnvmFS 以日志结构的方式实现，每个文件拥有自己独立的日志，更新日志尾指针才算更新操作完成，以日志尾更新的原子性保证操作的原子

性。目录操作涉及多个目录或文件，对于一些权限更改、重命名等涉及多个目录或文件的操作，采用轻量级 journal 机制，增加极少量的元数据拷贝开销，实现操作原子性。

2.4.1 文件操作

文件操作此处指的是单个文件的读写操作，文件数据以日志的方式进行记录，以前缀树的方式进行查找。数据更新的步骤顺序很重要，涉及到原子更新操作后的正确回滚。而读取操作的流程，不涉及数据的改动，步骤简单很多。

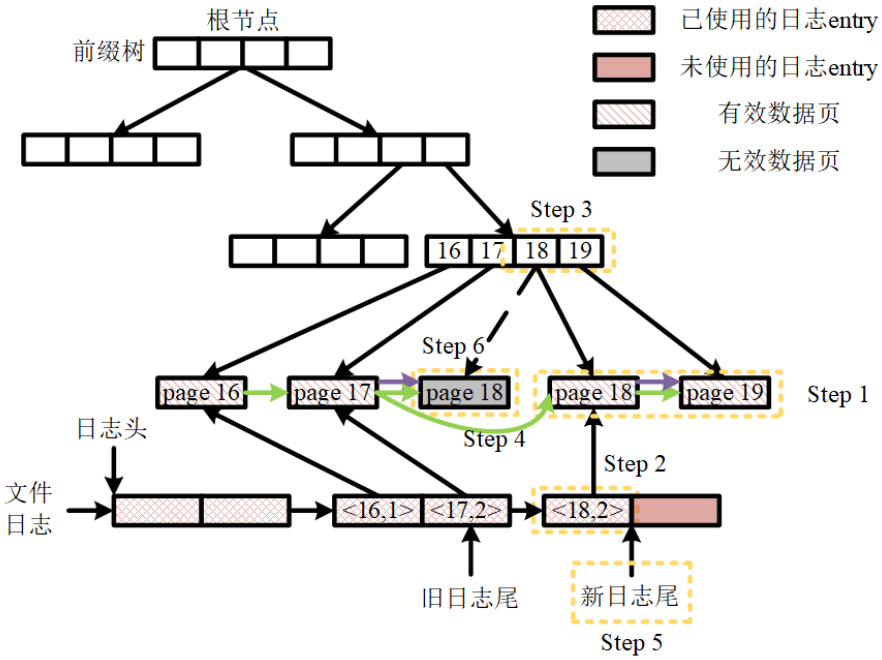

图 2-9 文件读写操作图

文件更新。如图 2-9 所示，描述了更新操作的基本步骤，日志<页偏移, 页数量>表示当前更新操作所在的文件页偏移和更新的页面数量；如最新的一次写操作<18, 2>表示文件页偏移 18，写操作大小为 2 个页面，其具体步骤可切分如下六步：

（1）根据写操作大小一次性分配所需的数据页面，将用户缓冲区数据拷贝到分配的非易失页面。

（2）在文件日志尾追加日志 entry，entry 记录了页面偏移、新数据页指针、写页面数量、更新时间、旧数据页面位置等信息。

（3）更新文件内部的前缀树，叶子节点指向新更新的数据页面；前缀树结构保

存在非易失存储空间，持久化掉电不丢失，无需重启后重构进行恢复。

（4）更新快读链表，指向新数据页面；数据页面中绿色链表为加速读操作的链表，即在此操作需要更新的链表。紫色链表指向当前 entry 的写操作拥有的数据页面。

（5）更新日志尾指针，指针大小在 64 位以内，用原子操作直接更新，完整此次更新操作。

（6）旧 entry 置位失效页面，后续后台线程自动回收失效页面。

若是大写，则会拆分成多个日志 entry，追加到日志尾，最后一次更新日志尾指针，完成操作。

对于读操作，流程简单很多，前缀树与链表结合的方式，也极大提高了查找定位的速度，流程分三步如下：

（1）计算读操作首页面的偏移，利用前缀树定位需要读取的数据所在的首页面在 NVM 的位置。

（2）以遍历快读链表的形式读取所需的页面，拷贝数据到用户缓冲。

（3）最后原子更新 inode 节点的访问时间，完成读取的原子性。

2.4.2 目录操作

本文的日志结构文件系统只能利用其天然的原子语义保证单个文件数据更新的原子性。如果对元数据也使用日志结构的更新模式，对之后元数据的读取会增加很大的复杂度，造成对一个元数据结构的一次访问都要进行多次反复查找与读取日志。对于涉及多个 inode 或 dentry 的元数据操作并不采用日志结构模式，而是引进一个轻量级的 journal 机制。journal 机制拷贝参与操作的 inode 或 dentry 元数据结构副本到 NVM 空间，形成链表挂载在 journal 表。系统掉电或崩溃后，可根据旧的元数据副本结构，进行操作的回滚，进而实现原子语义。

如图 2-10 所示，在 NVM 空间的 journal 段给每个 CPU 核预留了充足的 journal 指针对空间，最大可支持 CPU 核数量的并发操作。<链表头，链表尾>对初始为空值。开始操作时 start 指针指向参与操作的 inode 或 dentry 第一个结构副本，end 指针指向参与操作的最后一个结构，用于操作失败的回滚。操作完成或回滚完成后再次置为空值。每一个 CPU 核每次只能开启一个 journal 事务，不同 CPU 核间的 journal 事务互

不干扰，若涉及到同一个 inode 或 dentry 会预先加有读写锁进行同步。

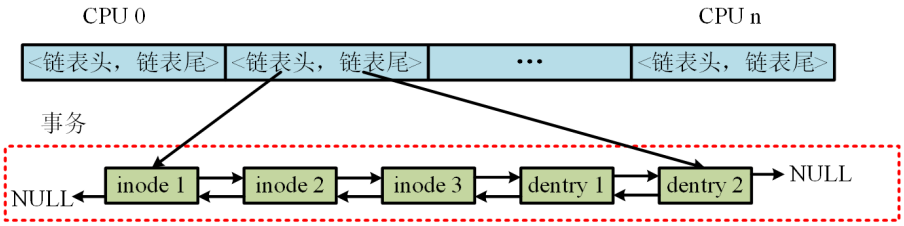

图 2-10　journal 操作记录图

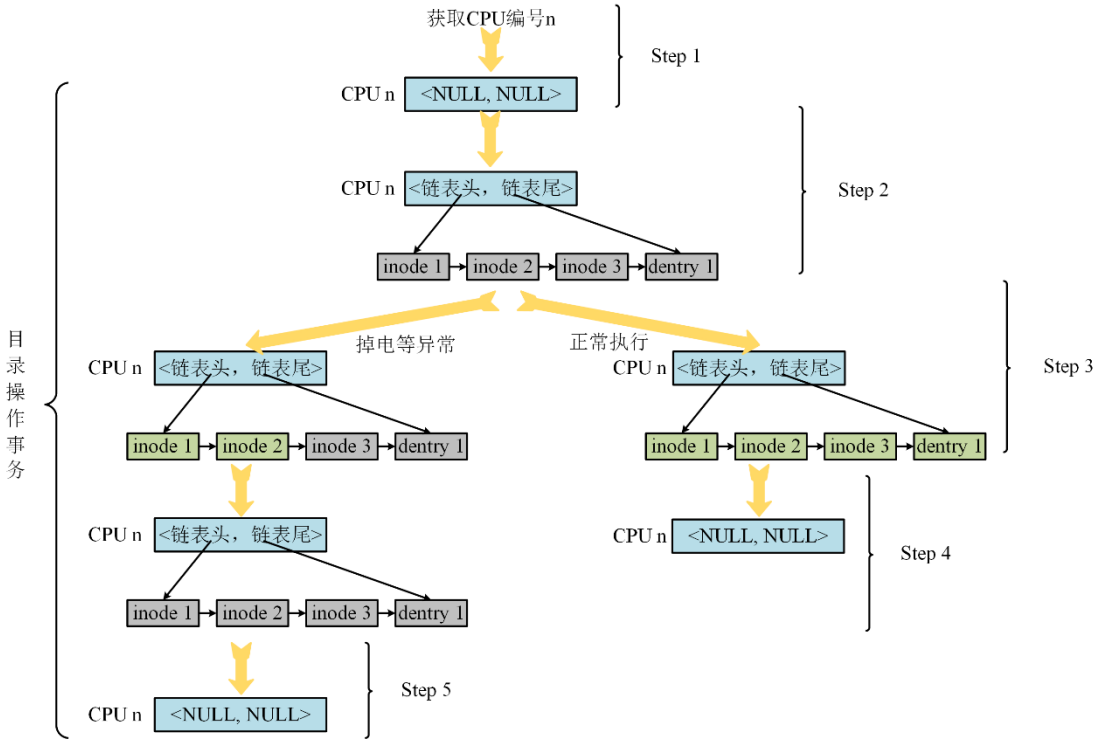

图 2-11　journal 操作流程示意图

如图 2-11 记录了 journal 操作基本步骤，包括正常处理和掉电恢复回滚处理：

（1）获取当前 CPU 核 ID 号，找到 journal 段对应 CPU 核的 journal 指针对空间，初始 journal 指针对为空值。

（2）然后以双向链表的形式，逐个记录参与操作的 inode 或 dentry 副本，并获取对应 inode 或 dentry 的读写锁，start 指针指向链表头，end 指针指向链表尾。

（3）接着开始从链表头，逐个执行单个文件或目录的更新操作，完成后转到 5）；若是出错或中途掉电，等新启动后跳转到 4）。

（4）从链表尾开始遍历，进行回滚，逐个对 inode 或 dentry 执行相应的回滚操

作即可恢复到原来的状态。

（5）最后，不管是经历了回滚，还是最终操作成功完成，重新将 journal 位置为空值，等待下一次操作执行。

2.5 非易失存储空间管理设计

2.5.1 非易失存储空间管理机制

非易失存储空间管理是 UnvmFS 实现元数据在用户态访问和更新的关键设计。

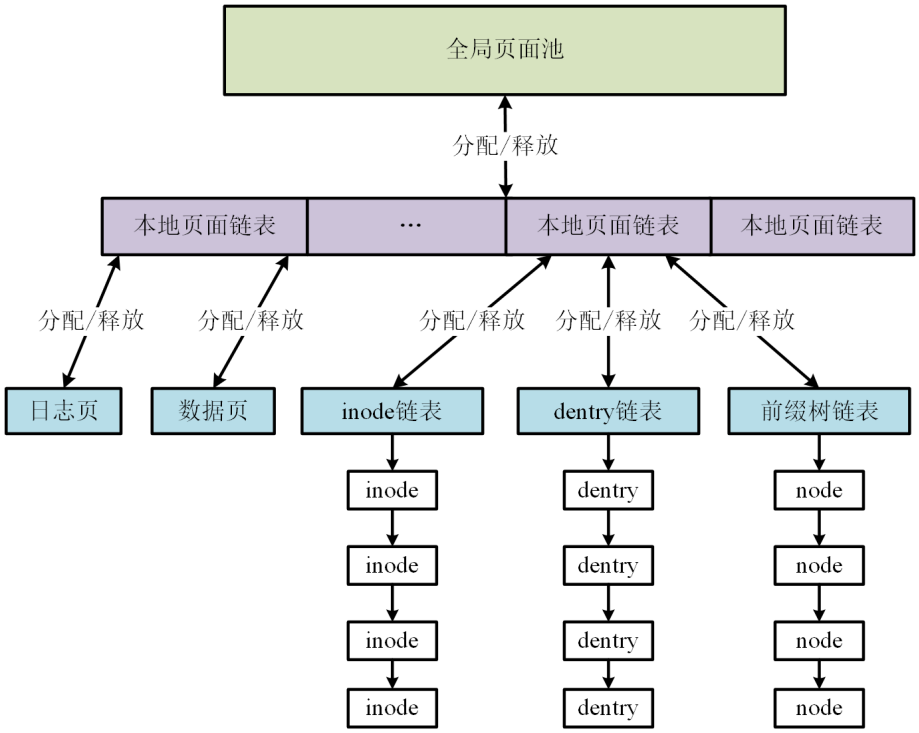

图 2-12 非易失存储空间管理分级图

相比其它用户态文件系统，UnvmFS 除了在用户态对数据进行拷贝，还在用户态通过相对寻址实现多用户进程间元数据共享，消除元数据更新所需要的同步开销。非易失存储空间管理分级如图 2-12 所示，NVM 的自由空闲空间被切分为整齐的页面放在全局页面池，每个 CPU 核的本地页面链表按需从全局池获取页面。本地页面链表管理对应 CPU 核的 NVM 资源，对文件系统数据和元数据的分配与回收。本地页面链表以页粒度提供文件的日志页和数据页。本地页面链表以页粒度提供非易失页面给 inode、dentry 和前缀树等结构链表，结构链表再对页面划分生成所需结构对象，

挂于结构链表，供需要取用。以上是非易存储失空间的三级管理结构的描述，结构有总有分，能支持较大的 CPU 核的并发资源申请。

2.5.2 页面分配状态

关于整个非易失存储器的自由存储空间划分，设计先有一个完整的全局非易失页面池，而每个 CPU 核有自己的本地非易失页面挂载链表。在本地链表页面用完或不足以分配给下一次操作时，主动向全局页面池批量申请新的页面。

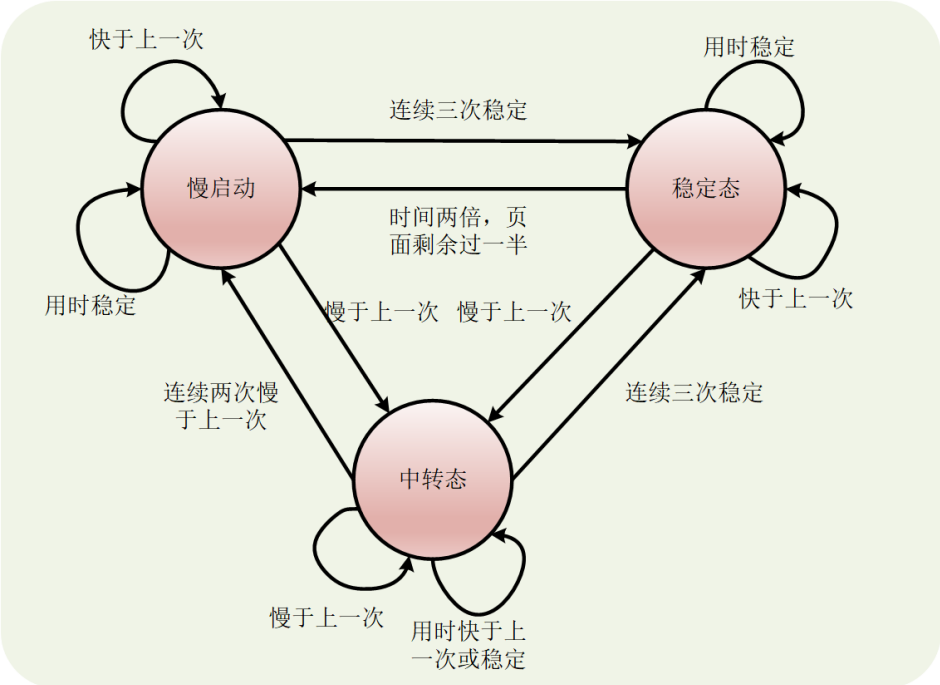

图 2-13 页面分配状态机示意图

CPU 核某个时段执行某些用户进程时，不同用户对文件的读写比例、读写大小、读写频率是不一致的。比如只读用户没有新分配数据页面需求，而只写需要快速分配大量新页面支撑写操作。因此 CPU 核本地链表根据所执行用户进程对页面使用量需求不同和页面使用速度不同，自动动态调整每次向全局池申请的页面数量。动态按需分配页面给本地链表能带来很多好处，一是避免固定化的分配，使得一些本地链表的数据页面迟迟得不到使用机会而造成空间浪费的情况发生；二是给一些大量使用页面的本地链表分配多的页面数量，减少本地链表与全局池的交互；因为本地链表访问全局池时对全局池加锁，多核间对全局池的交互次数增多，访问冲突变多会引起加锁

阻塞的时间增长。

为实现页面分配机制，借鉴网络拥塞算法，设计一个简单的状态机，如图 2-13 所示，初始时分配 2 个页面作为慢启动状态。之后若是页面用完速度快于上一次页面用完速度，则往上成倍增加分配页面数，如 4，8，16，32，64，128，256，512 个页面，以页面大小 4 KB 计算，最多一次批量分配 2 MB 大小的空间到本地链表；若是用完页面速度稳定，则获取页面数量保持不变，当连续三次从全局池获取页面时间间隔稳定，则进入稳定态；若是出现慢于上一次的情况，则进入中转态，获取页面数减半，中转态是本地链表页面获取数量的一个临时状态。

在稳定态，若是用时两倍于上一次且页面使用量不足 1/2，直接回慢启动态，且归还多余页面至全局页面池，重新开始计算本地链表获取最佳的页面数量的值。在稳定态，若是页面用完速度快于上一次则增加 4 个页面的分配量，稳定态每次页面分配数量固定；若是页面用完速度慢于上一次，则进入中转态，并将获取页面数减半处理。

若是在中转态页面用完速度继续减慢，则批量获取页面数继续减半；连续两次页面用完速度减慢，则回到慢启动状态，重新开始；若是连续三次用完速度稳定，则进入到稳定态；若是用完速度变快，则分配页面数翻倍。考虑到频繁的状态转变也会影响效率，采取比较宽松的误差判断，快与慢的定义，时间误差允许不超过 20%，20%以内默认为相等。

2.6 垃圾回收机制设计

日志结构的文件系统，在保证较高性能的前提下，还能维持文件操作原子性的语义，减轻应用层自己实现原子性的负担。但是，追加写的方式，旧的无效数据和日志 entry 的元数据需要定时进行清理，及时归还空间。页面回收不会有数据拷贝，但是也会附带一些额外的元数据日志 entry 分配和拷贝，占用一定 CPU 资源。垃圾回收应在后台及时回收失效数据，并尽量减少对前台操作的影响。因此本文在内核模块使用两个后台回收线程，一个定时清理线程，一个阈值清理线程，在适当时期及时清理无效页面，释放空间供给更多新来的写操作。至于为何在内核态实现回收线程，而非

在用户态 lib 库实现，是因为用户进程分配到的 CPU 时间片有限，lib 库的线程会在一定程度上占用用户进程的时间片，相当于文件系统额外占用了某些用户应有的资源，对某些用户进程的不公平。

那么在何时进行垃圾回收呢？阈值清理线程要对一段时间内的读写操作次数进行统计，确定垃圾回收的时机。考虑到写负载为主导的场景下，需要及时供给更多的空闲页面，因此需要在较低阈值时开始进行垃圾回收，约定默认阈值为全局页面池空间使用量 70%，停止回收阈值设置为 50%；而在读为主导的场景下，可以在达到默认阈值为 90%再开始垃圾回收，停止回收阈值设置为 70%；读写均衡场景下，开始回收阈值设定为 80%，停止回收阈值为 70%。再开启一个定时清理线程，增加定时回收的机制，开启时间范围内的定时垃圾回收，每次只回收一个 inode 节点的失效数据。两个线程不会同时开展垃圾回收，同一时间只允许一个线程在进行垃圾回收工作。

既然本文以 inode 作为写日志的基本单位，垃圾页面回收也依照 inode 为单位。如何选择合适的日志页面进行回收也成了一个新问题。为尽可能减少对用户进程文件操作的影响，需要选择最近都没被访问的最冷的文件进行垃圾回收。每次文件访问都会修改 inode 结构的 atime 参数记录最近访问时间，这是选择目标的一个重要参考因素。还需要考虑的是当前文件及文件内日志页面中无效页面分布与比例，如果当前文件主要是读操作和追加写操作，很少的修改操作，则有效页面占据大部分空间，可跳过该文件，选取下一个文件进行回收，这是需要考虑的第二要素。

于是垃圾回收分两步：

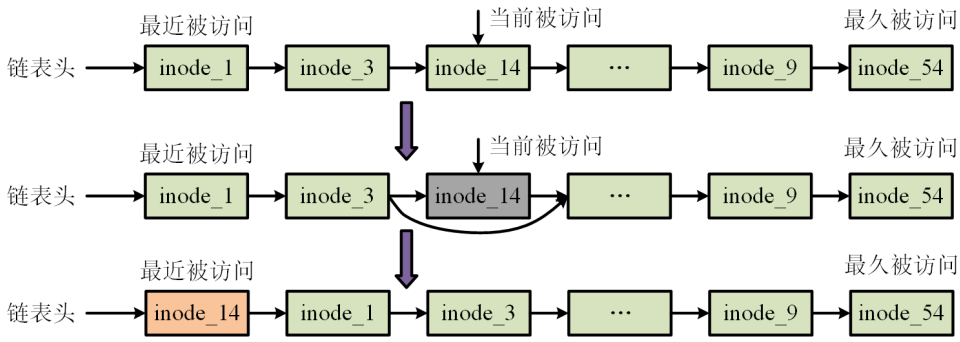

图 2-14　inode 链表时间排序图

第一步，使用简单的 LRU[42]（Least Recently Used，最近最少使用）链表记录 inode 节点最近被访问的时间排序，每次访问以 O(1)时间复杂度进行排序，选取垃圾回收的 inode 节点依然只需 O(1)时间复杂度。如图 2-14 所示，正在被访问的 14 号 inode 节点，被移动到链表头部，而每次垃圾回收从链表尾部开始选取回收节点，跳过无效页面占比低于 30%的 inode。

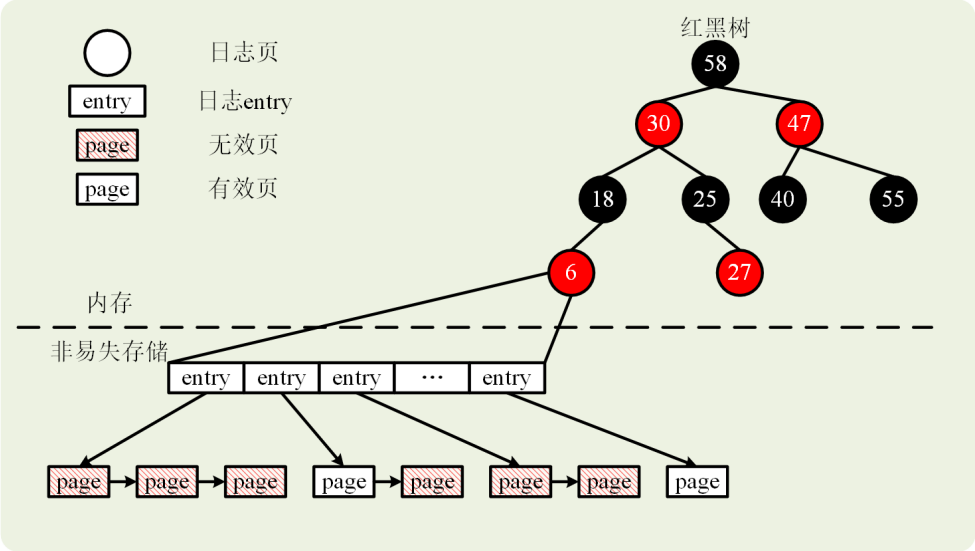

图 2-15　inode 失效数据统计图

第二步，选取好回收的 inode 节点后，由于每个 inode 节点中日志数据块的失效程度不一致，此时需要进行第二轮筛选。日志数据块即日志页面，一个日志页面上有多个日志 entry。如图 2-15 所示，对 inode 节点中日志数据块中失效页数量进行排序统计，从高到低选取日志数据块进行垃圾回收，直至日志数据块中无效数据少于 50%不再进行选取。不对无效页较少的日志数据块进行回收，是因为其回收操作会带来过多的元数据转移的拷贝操作，反而增加 inode 被锁时间，造成用户读写响应慢。本文采用红黑树的方式，以日志数据块中失效数据页数量为权值进行排序。红黑树直接在内存中实现，系统掉电或奔溃，则在后台重新进行恢复处理。

2.7　本章小结

本章详细介绍了用户态文件系统 UnvmFS 的整体结构到细节设计，先从当前的一些文件系统的问题开始分析，提出了 UnvmFS 的设计目标。接着提出 UnvmFS 的

整体结构由用户态库和内核态模块两部分组成，主要文件系统操作功能在用户态完成，内核态仅负责部分辅助性功能。UnvmFS 把整个 NVM 空间映射到用户态地址空间，需要对空间进行分段划分并管理，在组织结构设计这一节描述了空间布局并对文件系统的组织方式进行了介绍，在这一节最后还介绍了 UnvmFS 定制化的前缀树，以加快查找速度。针对单文件或多文件多目录操作的原子性操作流程也进行了详细设计介绍和举例。最后详细说明了整个文件系统的非易失存储空间管理和失效数据垃圾回收的设计。

3 基于 NVM 的用户态文件系统 UnvmFS 实现

本章基于上一章的系统设计，从代码实现的角度对UnvmFS进行更深入的阐述。以用户态库和内核态模块两大部分基础，分别介绍模块内部的主要功能实现。

3.1 用户态文件系统 UnvmFS 整体实现

UnvmFS 的整体结构已经在 2.3 节中进行详细描述，在用户态库和内核模块内部，各自根据功能划分，可将 UnvmFS 划分为多个功能模块，模块组成如图 3-1 所示。

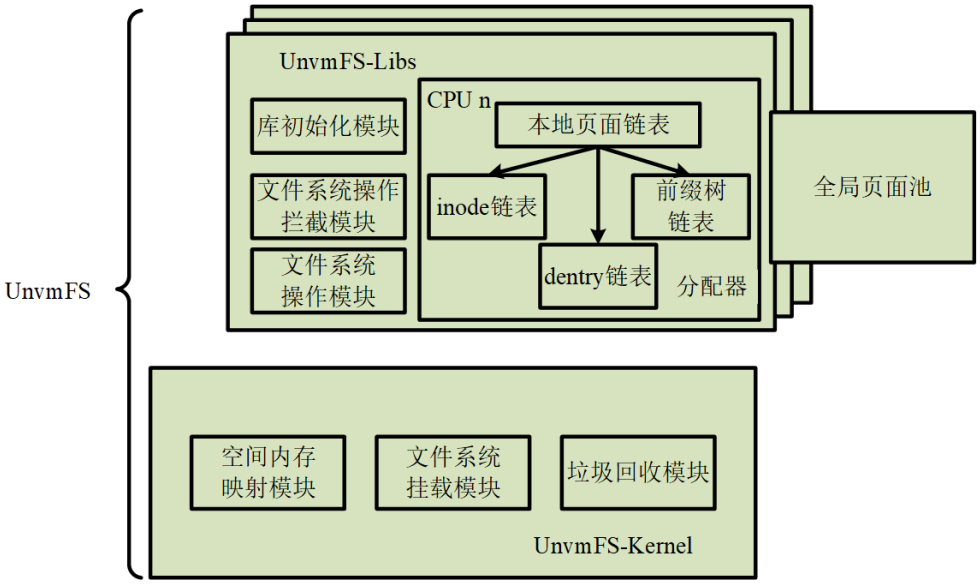

图 3-1 UnvmFS 功能模块图

内核模块主要由三个功能模块组成：

（1）空间内存映射模块。用户态库初始化时调用 mmap 接口，以系统响应的方式向内核模块发送 NVM 空间内存映射到用户态地址空间的请求，内核模块接收来自用户态的内存映射系统响应，将当前 NVM 映射到该用户进程地址空间，返回映射基址给用户态，之后该用户进程可访问整个文件系统空间。用户态文件系统下，原有的 mmap 功能不再需要，内核模块对 mmap 进行了重写，直接映射整个非易失存储器空间到该用户进程地址空间。

（2）文件系统挂载模块。文件系统注册后，用户可执行挂载命令，内核模块接收来自用户的文件系统挂载命令，完成超级块的初始化，以及页面空间管理、inode、

前缀树等资源的初始化等工作，并创建垃圾回收线程。

（3）垃圾回收模块。以后台线程的形式，定期以及在空间使用量达到阈值时，选取近期未被使用的一些文件，对过时的日志及数据进行清理和回收，释放空间。

用户态库主要由四个功能模块组成：

（1）库初始化模块。用户初次调用 lib 库需要向内核模块发送 mmap 系统调用对 NVM 空间进行内存映射，从而获取访问基址。获取基址后，将本地的页面、inode、前缀树等一些数据资源全局变量指针赋值到 lib 库相应全局变量，供后续进程对 lib 库资源的访问使用。执行完初始化后，该用户态进程即可执行正常文件系统操作，而无需与内核模块进行交互和同步。

（2）文件系统操作拦截模块。为了将文件操作在用户态进行拦截，不进入内核 VFS 层处理，而是转向 UnvmFS 直接处理，用户态库实现了一套同名的文件系统操作接口。

（3）文件系统基本操作模块。整个文件系统的组织结构都在用户态实现，在用户态 lib 库实现了多进程可重入的目录及文件等相关操作功能。由于时间关系，本文基本实现了文件的操作和小部分的目录操作。

（4）非易失分配器。为了使地址空间相互独立的多个进程能获取一致的文件系统元数据，实现一个以偏移为基础的非易失分配器，元数据和数据的访问都以基址加偏移的方式获取其存储位置。整个 NVM 空间初始化为一个全局的页面池，以链表的形式挂载。再以 CPU 为单位挂载各自的本地页面链表，本地链表提供页面资源给 inode 节点链表、前缀树链表和日志 entry 链表等元数据结构链表和更新所需的数据页面，本地链表页面不足时，向全局池申请资源。而 lib 库被调用时，根据当前线程所被执行的 CPU 核获取该 CPU 核的本地链表进行资源的分配。

3.2 UnvmFS 内核模块实现

以往的用户态文件系统内核模块主要功能是实现元数据同步，UnvmFS 的内核模块不同，其负责内核模块与文件系统基本操作的解耦，元数据处理直接放在用户态，仅保留最基本的挂载、内存映射和后台垃圾回收功能。本文内核模块的实现，尽

量依托于原有内核文件系统的机制，降低代码实现难度。

3.2.1 文件系统挂载功能

UnvmFS 内核模块加载时注册，模块加载后用户执行命令行挂载文件系统。实现与交互流程如图 3-2 所示，UnvmFS 内核模块安装时，会将 UnvmFS 在 Linux 内核进行文件系统注册，将文件系统的名字和文件系统的挂载函数进行登记注册。

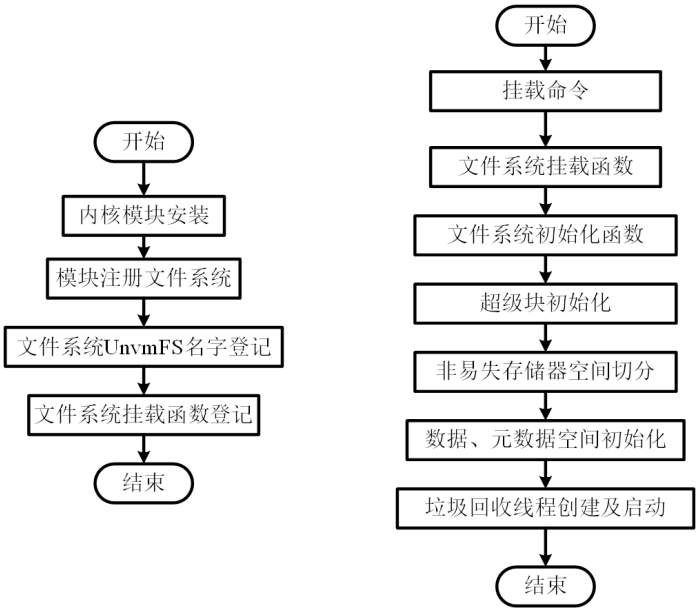

图 3-2 文件系统挂载交互图

在执行挂载命令挂载 UnvmFS 时，内核调用 UnvmFS 文件系统的文件系统挂载函数，进行文件系统初始化操作，包括超级块的初始化、非易失存储空间切分、数据、元数据空间初始化以及垃圾回收线程的创建和启动，至此文件系统的挂载流程结束，用户进程可通过 UnvmFS 用户态库进行文件系统的访问。文件系统注册数据结构如表 3-1 所示。

表 3-1 文件系统注册数据结构 UnvmFS_fs_type

成员名称	默认值
owner	THIS_MODULE
name	"UnvmFS"
mount	UnvmFS_mount
kill_sb	kill_block_super

在内核模块安装时，调用文件系统注册函数 register_filesystem，将文件系统注册数据结构 UnvmFS_fs_type 进行登记注册，即可完成此项功能。

3.2.2 非易失存储空间内存映射

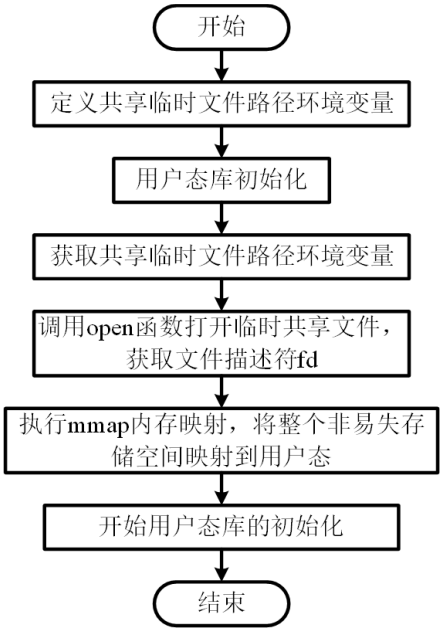

图 3-3 非易失存储器内存映射图

在每个用户进程初次使用 UnvmFS 用户态库时，需要进入内核获取非易失存储器的整个空间的内存映射。完成这个操作需要一些辅助，具体流程如图 3-3 所示。首先定义一个挂载目录下的共享的临时文件路径环境变量，之后调用 open 函数以系统响应的方式进入内核，该步骤进入内核后直接调用内核最原始的 generic_file_open 函数执行文件打开操作即可，这步主要是为后续操作获取文件描述符；然后使用获取的文件描述符执行 mmap 函数同样以系统调用的方式进入内核模块，在内核模块自定义修改的 mmap 执行函数调用 remap_pfn_range 将整个 NVM 空间进行内存映射；最后用户态库进入自己的初始化流程，以内存映射的方式直接对整个非易失存储空间进行访问。

3.2.3 文件系统垃圾回收

垃圾回收相关数据结构，多依赖于超级块、inode 和日志数据页，在其中设置垃

圾回收相关的成员结构。主要成员结构如表 3-2 所示：

表 3-2　垃圾回收主要结构成员

数据成员	类型	说明
s_list	struct list_head	LRU 链表头部
l_node	struct list_head	LRU 链表节点
rb_tree	struct rb_root	红黑树根节点
invalid_cnt	unsigned int	无效数据页数量
pages_cnt	unsigned int	总数据页数量

超级块作为 inode 节点中心，每次文件被访问时，更新 LRU 链表，将最近被访问的 inode 节点放入 LRU 链表首部，从尾部开始选取无效页面高于 30%的 inode 进行回收。inode 节点以红黑树形式排序日志数据页，每次选出无效数据页最多的日志数据页进行回收，且不回收无效数据页低于 50%的日志数据页，减少不必要的拷贝。

在后台创建垃圾回收线程。过度频繁的垃圾回收会造成系统操作的阻塞，垃圾回收时间间隔过长又会造成页面不足的情况。于是要设定垃圾回收时机，精准回收。垃圾回收设定的时间、临界值等如表 3-3 和

表 3-4 所示，以两种方式两个线程进行垃圾回收。阈值回收机制，线程每 1 秒判断一次是否达到开始回收阈值，达到则开始进行垃圾回收，反之则睡眠 1 秒再进行判断；读写比例计数器每睡眠 10 次进行重置。定时回收机制，采用随机函数确定睡眠时间，未达到回收阈值时，每次回收一个 inode 节点中的无效数据。

表 3-3　垃圾阈值回收时机表

读写比例	开始回收阈值	停止回收阈值
读 70%	90%	70%
写 70%	70%	50%
其它	80%	70%

表 3-4　垃圾定时回收时机表

定时时间（s）	回收数量
1~6	一个 inode

阈值回收机制和定时回收机制的差别在于回收的判定条件和回收的数据量。如图 3-4 所示为阈值回收线程流程，每睡眠 1 秒，进行一次读写比例计算，进而根据近期读写比例判断读写场景，根据对应场景，判断当前是否执行垃圾回收且是否达到设定的回收阈值；若是未执行垃圾回收且达到设定的回收阈值则开始进行垃圾回收，第一步从超级块 LRU 链表尾选择最近最久未被访问且无效页高于 30%的 inode 节点进行垃圾回收。

第二步，从 inode 节点中逐个选择最多无效数据页的日志页面进行垃圾回收，无效页面低于 50%时，不再对该 inode 进行垃圾回收。然后判断回收后，是否达到停止垃圾回收阈值，若是未达到，则继续从 LRU 链表尾选取 inode 进行垃圾回收；若是达到停止阈值，则返回睡眠。在睡眠后，进入垃圾回收与否，最后都会有个计数器判定是否已经进行了超过 10 次的判定，若是则重置读写比例计数器和判定计数器。

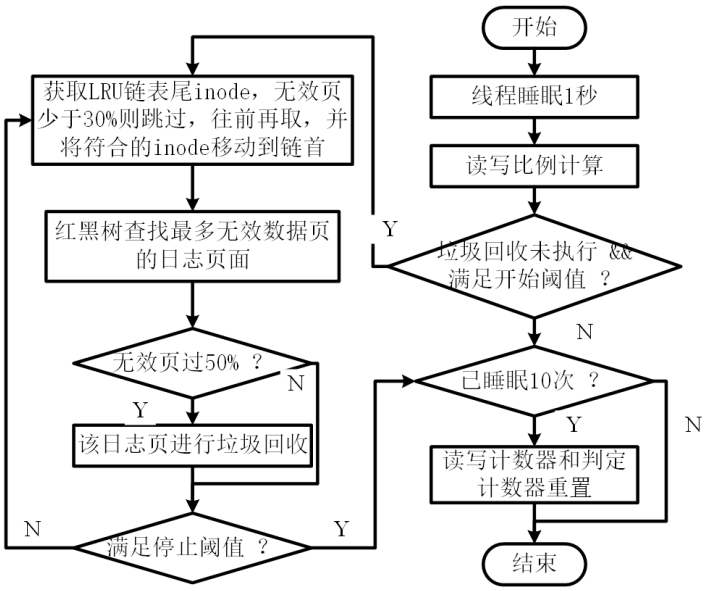

图 3-4　阈值回收机制流程图

相比于阈值回收，定时回收的判定和处理机制要简单很多，如图 3-5 所示，获取随机睡眠参数，按获取值进行睡眠；睡眠完成后，再判定当前阈值回收是否在执行，若是则继续睡眠；否则同样选取一个文件开始垃圾回收，回收完继续睡眠。定时回收主要起辅助性作用，能在一些较空闲时刻。

提前完成一些数据回收，减轻在数据页用多后，且用户读写密集状态下的批量阈

值回收。前面主要介绍了垃圾回收的回收机制，接下来以伪代码的形式，详细讲解日志页面具体的页面回收和日志重写。

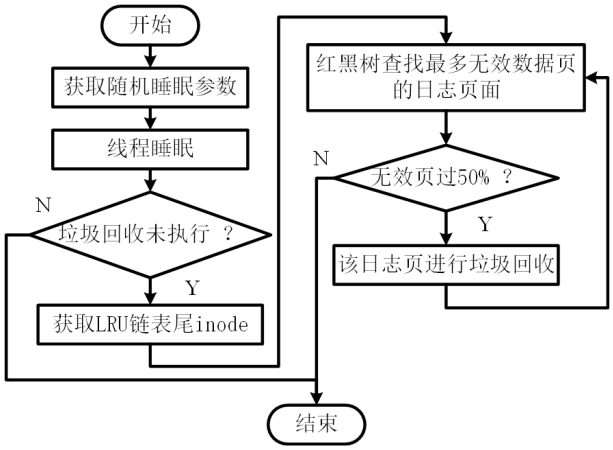

图 3-5　定时回收机制流程图

伪代码 3-1　日志页面回收

Inputs: Log Page, Data Pages
Outputs: New Log Entries

```
1.    log_tail ← inode->log_tail
2.    num_entries ← page_node->obj_cnt
3.    for i 0 to num_entries by 1 do
4.        log_entry ← (struct file_log_entry *)(log_page + (i * entry_size))
5.        j ← 0
6.        while j < log_entry->page_nums do
7.            new_entry ← get_log_entry(&log_tail)
8.            while j < log_entry->page_nums && !page_invalid(j) do
9.                add valid data pages to new_entry
10.               ++j
11.           end
12.           while j < log_entry->page_nums && page_invalid(j) do
13.               add page j to invalid data page list
14.               ++j
15.           end
16.       end
17.   free log page invalid data pages
19.   update inode log tail
20    end
```

如伪代码 3-1 所示，根据依次遍历日志页面上的日志 entry，再依次遍历日志 entry 上的数据页面，将有效的连续数据页面加入到新日志 entry，将无效数据页面加入到链表，供后续统一回收。遍历完后，释放该日志页面，并一次性释放无效数据页面，最后更新日志尾。

3.3 UnvmFS 用户态库实现

3.3.1 用户态库初始化

在用户进程第一次执行文件操作时，需要对用户态库进行初始化操作，来获取 NVM 空间的操作方式；如图 3-6 所示，初始化操作之前先将整个非易失存储器空间映射到内存地址。将非易失化的页面链表、目录链表、inode 链表、前缀树链表和 journal 链表的内存地址全部初始化在用户库的全局变量数组，供之后文件系统操作期间直接调用，避免每次计算偏移来获取内存地址。

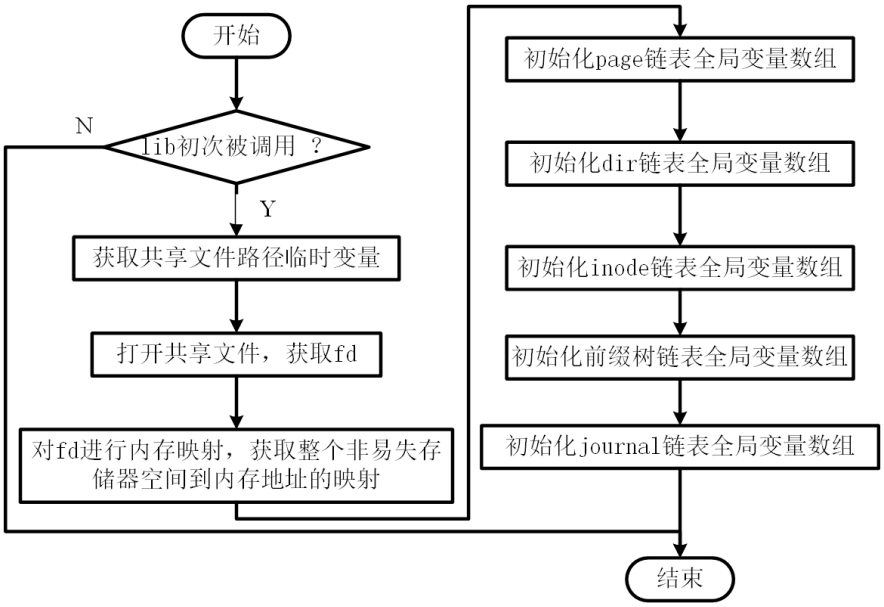

图 3-6　用户态库初始化流程图

3.3.2 文件系统组织结构

（1）超级块数据结构

超级块的基本结构如表 3-5 所示，包括基本通用的名称、时间、大小等信息；然后是用于垃圾回收的 LRU 链表和用于文件系统目录和文件索引的前缀树根节点。在

内核模块挂载时，将 cpu 核数量记录在超级块，一方面各级资源链表需要依据此参数进行初始化，另一方面为之后用户态库初始化提供参数。

表 3-5　超级块主要结构成员

数据成员	类型	说明
s_volume_name[64]	char	volume 名称
s_mtime	unsigned int	挂载时间
s_wtime	unsigned int	修改时间
s_block_size	unsigned int	块大小
s_volume_size	unsigned int	volume 大小
s_list	struct list_head	LRU 链表头部，垃圾回收用
s_tree_root	radixtree_t	前缀树根节点，目录、文件索引用
s_cpu_nums	unsigned int	cpu 核数量
s_rwlockp	pthread_rwlock_t	读写锁

（2）目录数据结构

表 3-6　目录主要结构成员

数据成员	类型	说明
d_flags	unsigned int	目录 flags
d_atime	unsigned int	访问时间
d_mode	unsigned int	目录模式
d_parent	unsigned long long	父亲目录
d_fd	unsigned int	目录描述符
d_in_page	unsigned long long	所在 NVM 页面
d_tree_root	radixtree_t	前缀树根节点，目录、文件索引用
l_dir	struct list_head	dir 链表用
j_node	struct list_head	journal 链表挂载
d_rwlockp	pthread_rwlock_t	读写锁

如表 3-6 所示，目录结构包括基本通用的 flag、时间、用户 id 和模式等信息，然后再是本模块特有的一些数据成员。d_parent 用于回溯查找父亲目录，使用偏移进行

记录。目录描述符 fd 由目录路径哈希而来，用于前缀树的查找。d_in_page 参数用于记录当前目录 dentry 所在 NVM 页面，该参数与 NVM 分配池相关。前缀树成员自然是用于下一级目录和文件的查找。dentry 结构是 NVM 页面分配而非内存，于是 l_dir 用于目录结构分配和释放使用。j_node 是在多目录或文件操作时，在 journal 机制的链表挂载使用。

（3）文件数据结构

表 3-7　inode 主要结构成员

数据成员	类型	说明
i_flags	unsigned int	inode flags
i_atime	unsigned int	访问时间
i_mode	unsigned int	文件模式
i_fd	unsigned int	文件描述符
log_head	unsigned long long	日志头部
log_tail	unsigned long long	日志尾部
file_offset	unsigned long long	文件当前偏移
i_in_page	unsigned long long	所在 NVM 页面
i_tree_root	radixtree_t	前缀树根节点，数据页面索引用
l_node	struct list_head	inode 链表用
rb_tree	struct rb_root	垃圾回收用红黑树
j_node	struct list_head	journal 链表挂载
i_rwlockp	pthread_rwlock_t	读写锁

inode 结构成员大多与目录结构成员相似，如表 3-7 所示。这里主要对 inode 特有的成员结构进行介绍，log_head 和 log_tail 指针在前面章节已经多次提到，即写日志的头尾指针偏移，新日志从尾指针开始写。inode 的前缀树与超级块和目录的前缀树有所不同，它是用来对数据页面进行查找定位的。红黑树结构以日志页面无效页作为 key 进行排序查找，在垃圾回收时使用。

3.3.3　文件系统基本操作

本节旨在详细描述文件的打开和读写流程，以及多目录或文件时的原子性操作

journal 机制的实现。

（1）文件系统基本操作

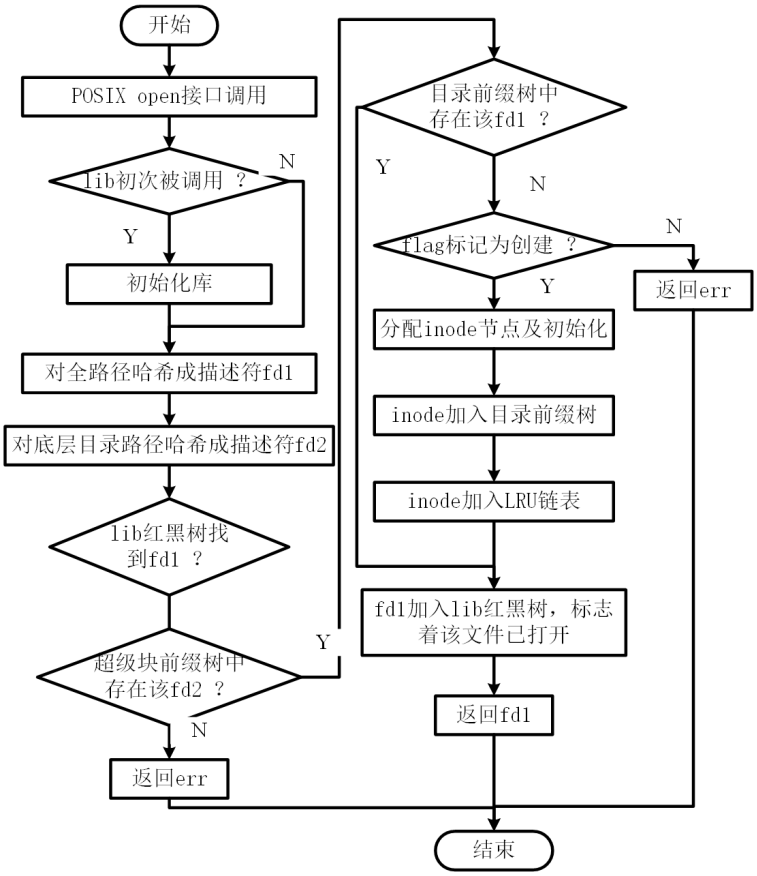

图 3-7　文件打开流程实现图

UnvmFS 会拦截 POSIX 标准的 open 系统调用，转移到 UnvmFS 用户态库直接处理，不进入内核 VFS 层，详细实现流程如图 3-7 所示。若是初次调用 lib 库，则会先执行一遍初始化流程，详细请参考 3.3.1 小节。之后会对路径有两次哈希，一次是整个路径的哈希用于文件索引，另一次是对文件底层目录的哈希用于目录索引。lib 库使用内存红黑树记录该进程已经打开的文件。用户进程调用 lib 库相关函数，该函数使用红黑树查找文件路径哈希成的 fd1 是否已存在，如果存在则表明该进程已打开该文件，直接返回文件描述符 fd1 即表示完成文件打开操作。若是文件不存在，则在超级块下的前缀树中索引底层目录 fd2，若是不存在该文件且底层目录也不存在，返回错误；若是目录存在，接着在目录前缀树索引文件 inode，若是 inode 找到，则加入 lib 红黑树，返回文件描述符 fd1，打开成功；若是未找到 inode 且 flag 未标记

为创建，则返回错误；若是未找到 inode 且标记为创建，则分配新 inode 节点并初始化，随后加入目录前缀树、LRU 链表和 lib 库红黑树，返回描述符 fd1，打开成功。

（2）文件读流程

文件读流程相较于写流程要简单很多，如图 3-8 所示。

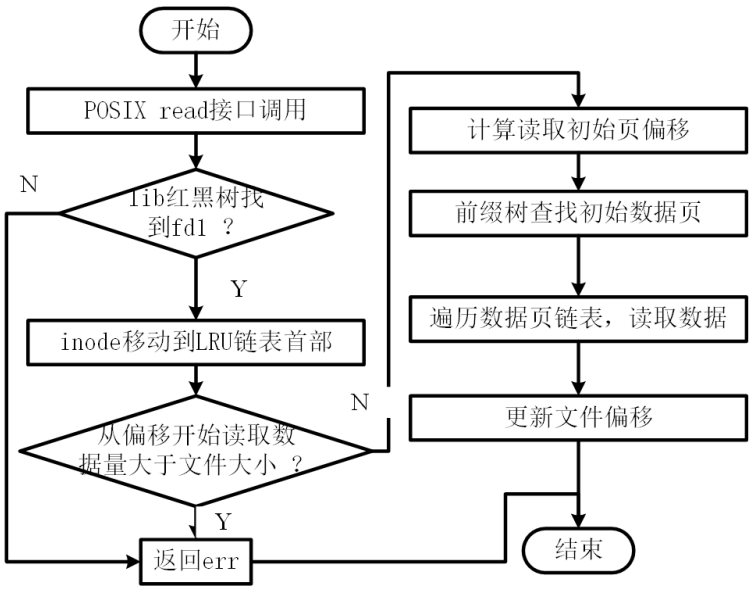

图 3-8　读流程实现图

首先在 lib 库红黑树查找文件是否打开，打开则从红黑树获取文件 inode 节点，未打开则返回错误；若是文件已打开，开始读流程，将 inode 节点移动到 LRU 链表首部，代表最近访问过，不进行垃圾回收。需要判断当前文件偏移加上文件数据读取的字节数是否超过文件大小，若是超过，也直接返回错误。然后开始真正的文件读操作，首先计算读取的文件初始页偏移，在文件前缀树找到初始页，之后遍历数据页面链表，读取数据即可。最后把文件偏移位置更新到最后读取的位置。

（3）文件写流程

如图 3-9，依旧是先查找 lib 红黑树确认文件已经被打开，获取 inode 信息。将 inode 节点移动到 LRU 链表首部，代表该 inode 节点最近被访问过，至此与读流程是相似的。

计算出总写入的页面数和初始写页面偏移，然后开始进入写流程。判断当前写页面数是否大于 0，若是，则判断总的写页面数是否超过 256 个页，一个日志 entry 最多支持 256 个页面即 1 MB 数据的写入，超过则拆分成多个日志 entry 写入；一次性

分配此次日志 entry 需要的数据页面数和分配一个新的日志 entry。

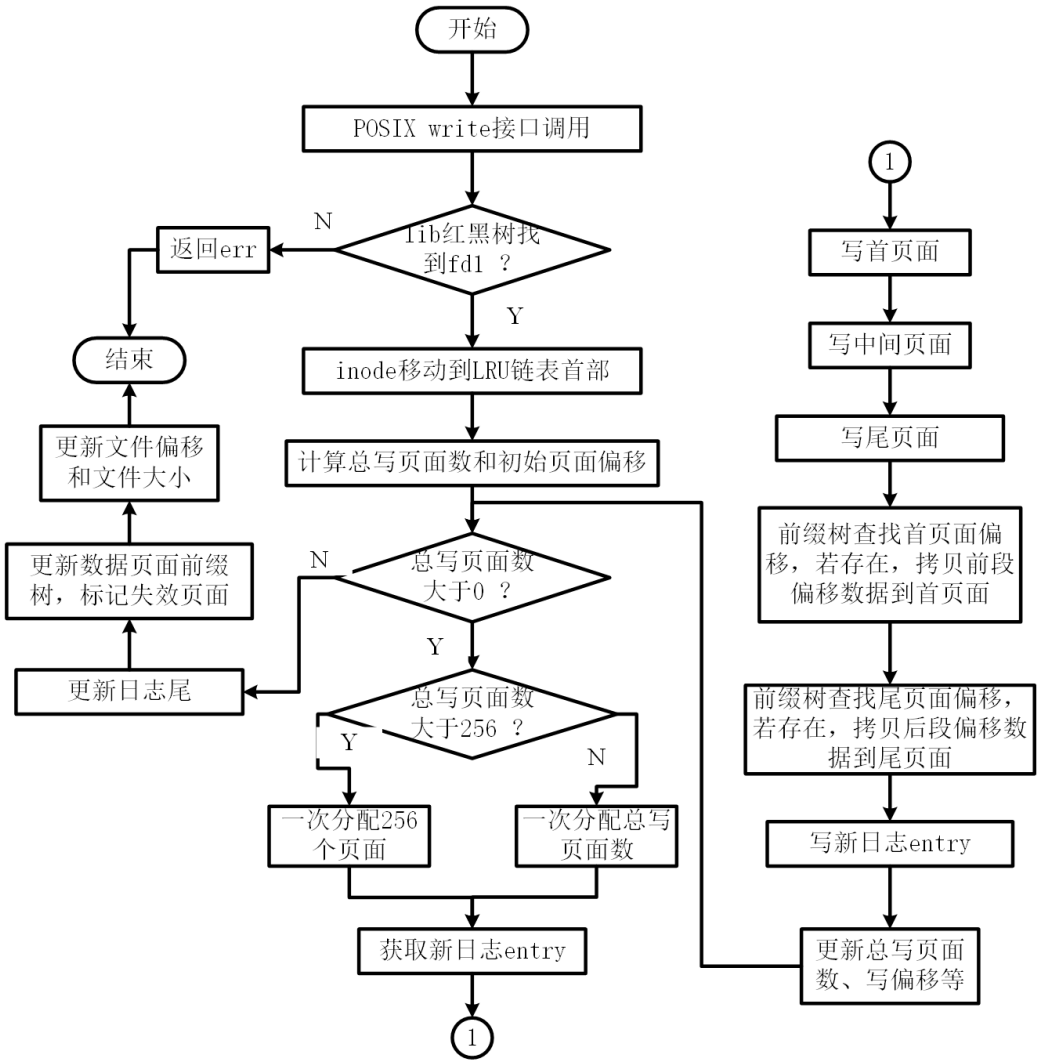

图 3-9　写流程实现图

写入数据分三个步骤：第一步，首页和尾页可能写不满一页，单独拎出来计算页内偏移和写入字节数进行写入；第二步，中间页面都是直接满页写；第三步，之后要对新的首页和尾页进行数据页面的补全，于是在文件内的前缀树查找旧的页面拷贝数据到新页面。完成数据拷贝后，对新的日志 entry 的写入和总页面数、写偏移等相关信息更新，然后继续进入循环中总写页面数判断；若是还大于 0，则重复刚才的写入操作；否则，跳出循环，更新日志尾部指针，更新文件前缀树，置旧数据页面为无效，更新文件偏移和文件大小，完成此次写操作全部流程。

在完成写操作之前需要先完成前缀树的更新，遍历旧日志尾和新日志尾之间的

日志 entry，再逐个遍历日志 entry 中数据页偏移，无效前缀树中该页偏移对应的旧数据页，然后将新数据页插入到前缀树中，完成前缀树的更新，如伪代码 3-2 所示为前缀树更新的伪代码记录了详细步骤。

伪代码 3-2　前缀树更新伪代码

```
Inputs:  inode, old_log_tail
Outputs: radix tree
1.      log_tail ← inode->log_tail
2.      cur_off ← old_log_tail
3.      while cur_off != log_tail do
4.          if is_last_entry(curr_off) then
5.              curr_off ← next_log_page(curr_off);
6.          entry ← get_entry(cur_off)
7.          for j 0 to entry->num_pages by 1 do
8.          old_data_page ← get data page from radix tree
9.          if data_page != NULL then
10.             old_entry ← get old log entry
11.             invalid old_data_page in log_entry
12.         new_data_page ← get from entry
13.         set new_data_page to radix tree
14.         end
15.     end
```

（4）目录原子操作流程

在执行多目录或文件的操作时，journal 机制下拷贝 inode 和 dentry 结构即可完成副本备份，以最小的代价，提供操作的原子语义。

如图 3-10 所示，先获取当前执行的 cpu 核的 ID 号，找到对应该 ID 的 journal 指针对是否为空；若非空，则此 cpu 还有其它的 journal 操作未完成，于是先调用线程调度函数 schedule()暂时让出 cpu；若为空，则开始拷贝参与操作的 dentry 和 inode，并将链表首尾指针拷贝到 journal 指针对。链表搭建完后，开始遍历链表逐个执行操作，若是中途失败或断电后重启，因为保留了原始 dentry 和 inode 副本，于是开始进行逆操作回滚；逆操作完成或正常操作完成后，逐个释放拷贝的链表副本，再重新将 journal 段置为空，完成操作回退或完成此次原子操作。

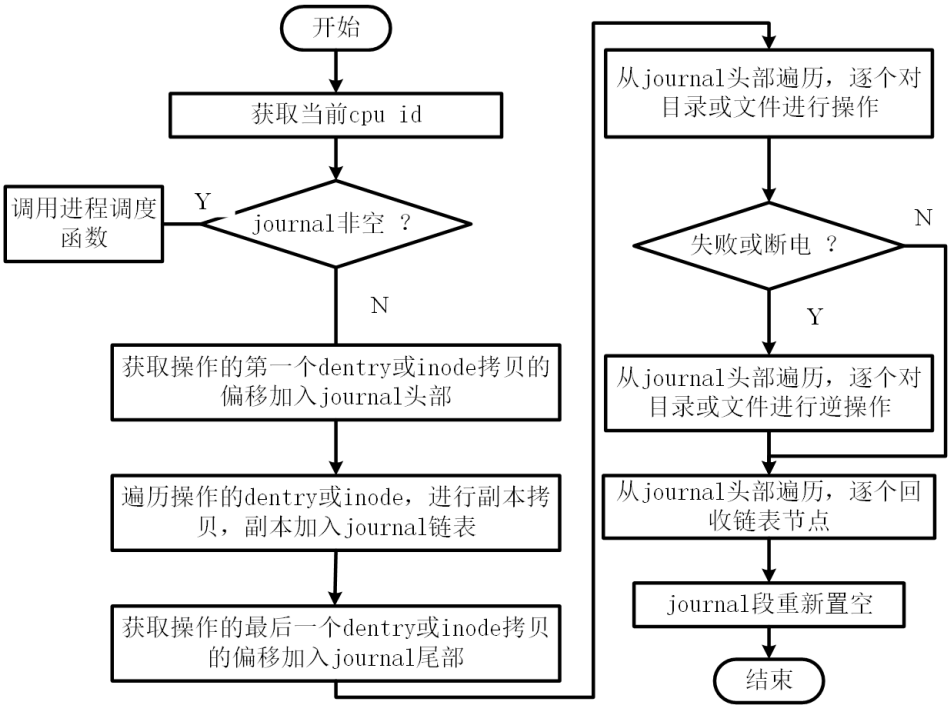

图 3-10　journal 机制流程图

3.3.4　非易失分配器

（1）非易失存储空间分配管理

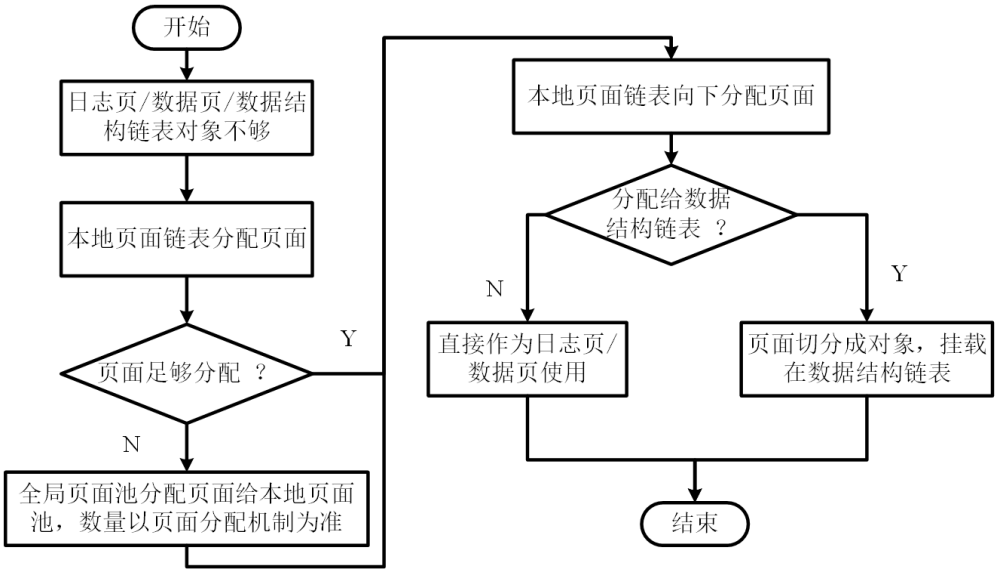

图 3-11　非易失存储空间管理流程图

最下层的非易失对象包括日志页、数据页和元数据对象。非易失存储空间管理流程如图 3-11 所示。日志页和数据页不够时直接面向本地页面链表申请所需要的页面

数量。元数据对象主要包括 inode、dentry 和前缀树节点，这些对象分别挂载在对应的数据结构链表上。文件系统向这些链表上申请需要的元数据对象，链表中对象不够时，才会进一步向本地页面链表申请页面，将页面划分成一个个元数据对象挂载相应的数据结构链表上。本地页面链表为空或者不足以分配给下一级时，向全局页面池申请页面，具体数量以页面分配机制计算为准。本地页面链表不加锁，而全局页面池在访问时，会提前加上互斥锁，防止多线程间的争抢。

（2）页面分配流程

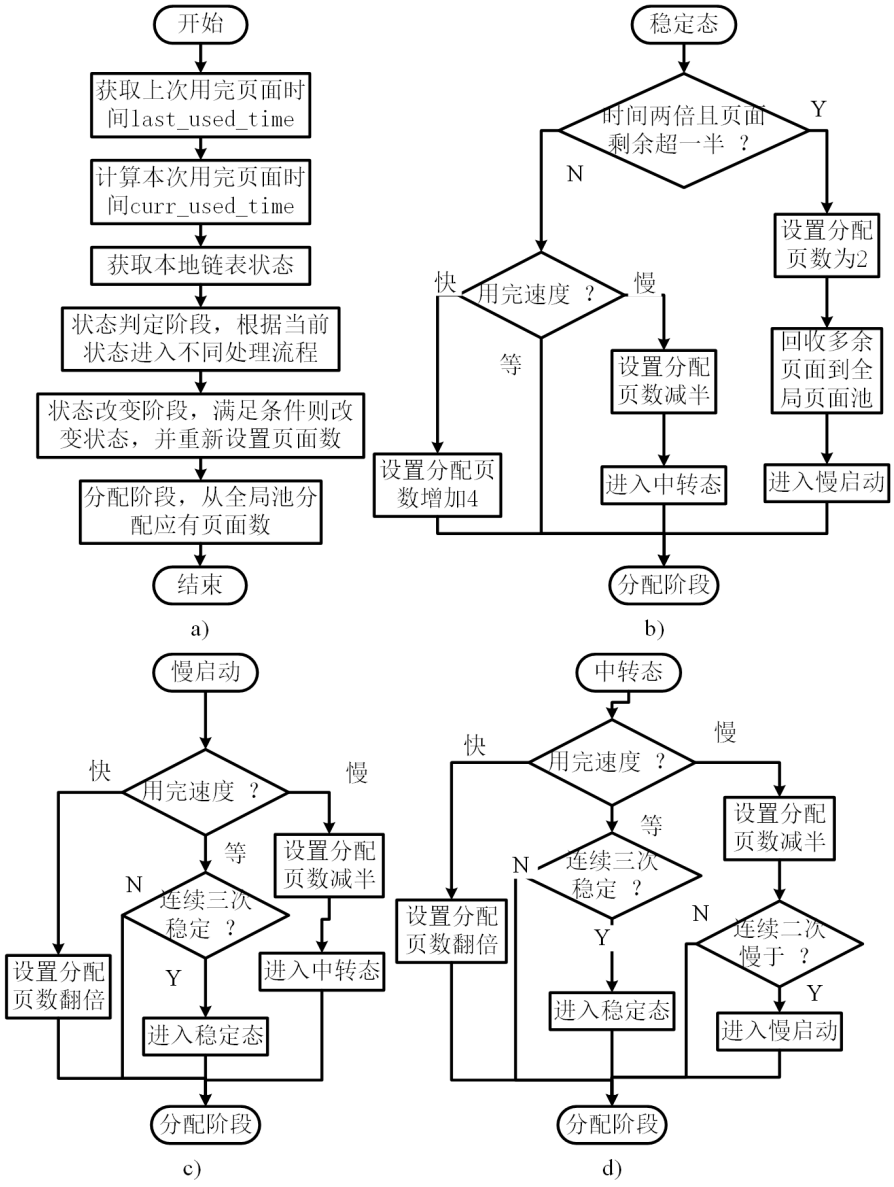

图 3-12　页面分配流程图

整个非易失存储器的自由空闲空间都以页面链表的方式，挂载在全局页面池。本地页面链表在使用完或页面不够时，会向全局页面池申请需要的页面。

如图 3-12 a)所示，描述页面分配的基本流程。在本地页面链表会用成员 last_used_time 记录上次页面使用完的时间,以及本次申请页面时的时间戳 last_alloc_time，再获取当前时间戳，得到此次页面用完花费的时间 curr_used_time；然后开始获取当前链表状态，进入状态判定阶段判断当前本地链表所处的状态，根据不同状态（慢启动、中转态、稳定态）进入不同处理流程，在状态改变阶段设置新的状态和分配页面数；最后，进入页面分配阶段，从全局池返回页面给本地链表。

如图 3-12 b)所示，若是当前在稳定态，两倍时间于上一次用完速度，且还有过半页面未分配出去，则释放多余页面，转入慢启动，重新开始获取稳定状态；若是用完速度变慢，则分配页面数减半，进入中转态，等待分配稳定。

如图 3-12 c)所示，若是当前在慢启动态，在用完速度变快时，快速对分配页面数翻倍（最高 2 MB）；等连续达到三次稳定状态时，进入到稳定态，缓慢每次增加 4 个页面分配量，防止翻倍时数量巨大造成浪费；若出现使用速度变慢的情况，则减半分配的页面数，进入中转态，等待分配稳定。

如图 3-12 d)所示，若是当前在中转态，用完速度变快则分配页面数翻倍，用完速度变慢则分配页面数减半，且连续两次变慢则进入慢启动，重新找寻新的稳定分配状态；若是连续三次用完时间稳定，则进入稳定态。

表 3-8　本地页面链表数据结构

数据成员	类型	说明
head	unsigned long long	链表头部
count	unsigned int	页面数量
same_stat_nums	unsigned int	连续同一状态次数
alloc_nums	alloc_page_num_type_t	每次页面分配次数
alloc_stat	alloc_page_stat_type_t	当前分配状态
last_used_time	unsigned long long	上次用完页面花费的时间
last_alloc_time	unsigned long long	上次分配的时间戳
mutex	pthread_mutex_t	互斥锁

如表 3-8 所示，本地页面链表的数据结构，有链表页面挂载的头部和页面数量计数器，也有多次分配连续同一状态次数的计数器。同时还会以枚举变量的形式记录当前链表所在的状态以及一次该分配的页面数量。

last_used_time 和 last_alloc_time 分别记录上一次分配的页面用完时间和上一次分配页面时的时间戳，配合下一次分配的时间戳，计算出页面用完速度的快慢。

3.4 本章小结

本章是对 UnvmFS 系统实现过程与细节的描述，以整个系统的主要结构模块开始介绍，内核模块和用户态库两个大模块，以及其各自内部的小功能模块。内核模块主要包括对用户挂载命令响应的文件系统挂载功能模块、对用户态库初始化映射响应的 NVM 空间内存映射功能模块和及时回收旧数据页释放空间的文件系统垃圾回收功能模块。而用户态库包括内存映射整个非易失存储空间以及初始化基本数据结构的库初始化功能模块、实现文件系统操作拦截的接口模块、实现目录和文件查找索引和文件系统基本操作的功能模块和非易失分配器功能模块。

4 系统测试与结果分析

本章是对 UnvmFS 系统性能的验证，在真实的 AEP 服务器环境下，用文件系统基准工具 Filebench，选取基本操作负载和真实应用环境负载，与 Ext4-DAX 和 Libnvmmio 测试对比，并对结果进行分析，证明本文方案的可行性。

4.1 测试环境

实验测试环境基本配置如表 4-1 所示，AEP 设备只支持 4.15.0 以上版本的 Linux 内核，本文在英特尔最新商用的 AEP 服务器上安装基于 5.10.1 内核版本的 Linux64 位系统 Centos7.6。服务器上有两条 128 GB 的 Optane 设备，以交错应用直接访问模式（App Direct Mode），/dev/pmem0 的分区形式供用户使用。

表 4-1 测试环境配置

名称	参数
操作系统	Centos 7.6 x86_64
内核版本	Linux 5.10.1
处理器	18 * Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz
主存	DDR4REG 64 GB
NVM	(Apache Pass) 2 Socket * 128 GB Optane DIMM
测试工具	Filebench 1.4.9

本文的性能测试使用 Filebench[43]工具，Filebench 工具能模拟真实的应用环境负载来测试文件系统的性能，不同于传统的测试基准工具，Filebench 作为一款自动化测试工具，不仅可以模拟文件系统基本操作顺序读写、随机读写、文件创建删除等，Filebench 还提供了负载模型语言（Workload Model Language，WML）调节测试参数和灵活的模拟真实应用的 I/O 行为。

测试对比方案选取一般内核态文件系统 Ext4、基于非易失存储器的内核态文件系统 Nova 和 2020 年 ATC 会议新发表论文的用户态文件系统 Libnvmmio。一方面体现出用户态实现文件系统相较内核态文件系统的性能优势，另一方面与最新性能最

优的 Libnvmmio 比较能表现出本文 UnvmFS 的设计下性能的提升。Libnvmmio 在 Microbenchmark 和 Macrobenchmark 测试时选取预分配 32 KB 空间作为对比测试，32 KB 接近于大部分测试负载的平均文件大小和用户态写请求大小，且不会有较大空间浪费，测试更满足客观条件。

4.2 额外内核态操作时延影响测试

在 2.1 节讨论过现在已有的用户态文件系统如 libnvmmio 元数据操作放在内核态，会在文件大小变化或映射空间不够等产生一些额外的内核态操作，增加部分或大部分 I/O 操作的额外处理时延，反而降低了系统性能。本节通过测试单线程单文件下对 1 GB 数据的 I/O 总时延，以此分析额外内核态操作对追加写、顺序写和顺序读三种基本操作的访问时延影响。该测试主要比较不同软件栈下文件系统优化对比，具体测试结果如图 4-1 所示，从测试结果得出以下几个结论：

（1）UnvmFS 的单文件追加写时延比 libnvmmio-1GB 降低 25.27%，libnvmmio-1 GB 在 1 GB 追加写时是没有额外的内核态附加操作的，其性能差异在于 libnvmmio 默认的 UNDO 日志策略多出的一次拷贝操作。而 UnvmFS 追加写时延比 libnvmmio-4 KB 降低 75.87%，libnvmmio-4KB 每次追加写不仅多出一次拷贝，而且会增加进入内核态的内存映射撤销与重新映射、文件预留等操作开销，比传统内核态文件系统 ext4-dax 的软件时延开销更高。UnvmFS 分别比 ext4-dax 和 nova 时延降低 69.85%和 45.96%，UnvmFS 追加写时延比 ext4-dax 精简了内核态切换开销以及复杂的软件栈开销，比 nova 主要缩小了内核态切换开销。

（2）在顺序读性能上 UnvmFS 和 libnvmmio 基本持平，libnvmmio 在各级预留空间时顺序读时延持平，说明预留空间只会对写操作产生影响，读操作可以直接在用户态完成且不会产生额外操作。UnvmFS 顺序读时延分别比 ext4-dax 和 nova 时延降低 44.89%和 35.96%，开销差异与写操作类似，在于内核态切换与更深的软件栈路径。

（3）libnvmmio 各级预留空间下顺序写时延基本持平，原因在于此测试的顺序写是建立在文件大小已经是 1 GB 之后再开始进行的顺序写时延测试，此时 libnvmmio 顺序写时延接近 libnvmmio-1GB 的追加写时延。因此 UnvmFS 在顺序写时延比

libnvmmio 降低 25%左右。UnvmFS 顺序写时延分别比 ext4-dax 和 nova 时延降低 64%和 50%左右，由此可知 ext4-dax 和 nova 单线程单文件的顺序写花费的时延与追加写时延基本接近，同一个文件系统顺序写和追加写性能差异较小，广义都可称为顺序写。

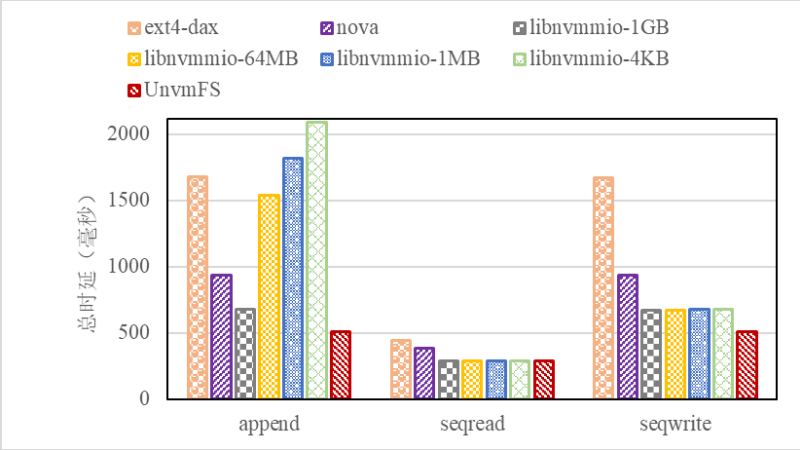

图 4-1　额外内核态操作时延影响对比图

UnvmFS 全程在用户态分配空间，拷贝数据，以日志方式更新数据，以尽可能精简的软件栈尽量降低响应时延，各组测试结果也表明 UnvmFS 在追加写和顺序写都能实现快速响应，顺序读性能与同是用户态文件系统的 libnvmmio 相当。

4.3　Filebench 不同负载测试

4.3.1　测试负载

本文使用的 Filebench 测试负载包括 Microbenchmark 和 Macrobenchmark 两部分。

表 4-2　Microbenchmark 测试负载配置

读写模式	访问模式	I/O 大小	文件数	线程数	文件大小
读	顺序	4 KB	10000	1、2、4、8、12、16、20	64 KB
	随机	4 KB	10000	1、2、4、8、12、16、20	64 KB
写	顺序	4 KB	10000	1、2、4、8、12、16、20	64 KB
	随机	4 KB	10000	1、2、4、8、12、16、20	64 KB
追加	/	4 KB	10000	1、2、4、8、12、16、20	64 KB

Microbenchmark 测试文件基本读写性能，如表 4-2 所示，对基本的读、写和追加操作进行测试，访问模式设置为顺序和随机两种，I/O 大小统一为页面大小 4 KB，文件数量为 10000 个，平均文件大小 64 KB，测试数据量为 10 GB。测试线程数 1 至 20 范围内，不同并发访问强度，各文件系统基本操作性能对比。

Macrobenchmark 是对真实的应用访问模式的模拟负载，选取文件服务器、网页代理服务器、网页服务器和邮件服务器负载进行对比测试。各负载具体配置如表 4-3 所示，文件服务器负载是写多读少，顺序读写为主，I/O 粒度较大的负载，网页代理服务器负载和网页服务器负载是读多写少的负载，而邮件服务器负载则是 I/O 粒度较小的负载模式。各负载读写操作外，夹杂了一些文件打开、关闭、设置文件指针位置等元数据操作，更贴近于实际访问。设置多组不同线程数的负载并发度，验证并发度对系统性能影响。

表 4-3　Macrobenchmark 测试负载配置

负载	文件大小	线程数	读写比例	文件数
Fileserver	2 MB	5、10、20、30、50	1:2	10000
Webproxy	32 KB	5、10、20、30、50	5:1	10000
Webserver	512 KB	5、10、20、30、50	10:1	10000
Varmail	128 KB	5、10、20、30、50	1:1	10000

4.3.2　基本操作测试

（1）追加写

文件系统在很多负载场景上都会有批量的追加写操作，比如文件服务器或者邮件服务器都会有对整个文件从零开始的写操作，完成对数据的存储。追加写作为性能测试的重要指标之一，如图 4-2 所示，测试在单线程多线程下，UnvmFS、libnvmmio、nova 和 ext4-dax 系统带宽差异。在线程数量 20 范围内，UnvmFS 始终提供最高的带宽，比 libnvmmio 性能高 15%至 35%，比 nova 高出 52%至 73%，比起 ext4-dax 则高出 60%至 104%。

UnvmFS 在追加写场景下，各种线程并发下都能保持最高的性能，离不开其以文

件为单位的日志结构。UnvmFS 与同为用户态文件系统的 libnvmmio 追加写性能差异的原因主要在于，libnvmmio 刚开始默认的日志策略是 UNDO 日志，在文件大小预分配变动范围内会一直采用默认策略，UNDO 日志策略会先将原文件旧数据拷贝到日志，再将新数据写到原文件，尽管是以内存映射的方式进行写操作，两次的拷贝，也会带来成倍的时延；之后超过预分配变动范围后，在判断出读写比例为全写，进行日志模式的切换，转为 REDO 日志，此时本应该能维持用户态较高性能的写操作；但因为追加写导致频繁的文件大小改变，需要重复对文件的内存映射与撤销等系统调用操作，本来对数据用户态的更新，反而增加了多次的元数据操作，在小粒度写操作下反而比拷贝数据花的时间更多，导致写性能的变差。

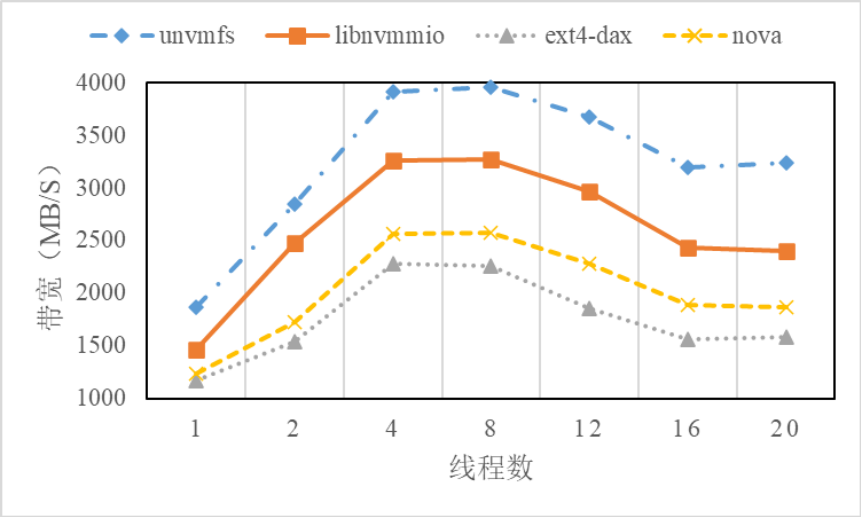

图 4-2　追加写性能测试图

libnvmmio 将元数据交给内核文件系统处理，文件创建时会预留指定大小空间的文件，每次打开时会将文件整个映射到内存，在文件的大小超过预分配阈值，需要同步对原文件进行新的预留，进行重新的内存映射，导致整个文件的暂时无法访问，影响并发。线程数量越多，元数据操作越频繁，进而影响 libnvmmio 性能。UnvmFS 只在系统初始化时执行内存映射，之后的操作全在用户态执行，日志结构的特点使得新写数据只需要拷贝到新分配数据页面，也只有一次拷贝，且处理软件栈很简洁，性能自然要提升很多。

nova 与 UnvmFS 同为日志结构型文件系统，最大差异在于实现在用户态或内核态，软件栈深度不同。nova 和 ext4-dax 经过系统响应切换线程进入内核，跳过了页

面缓存，但本身还经过复杂的 VFS 层。

（2）顺序写

顺序写性能测试结果如图 4-3 所示，UnvmFS 系统带宽比 libnvmmio 高出 22%至 35%，比 nova 高出 53%至 69%，带宽比 ext4-dax 则高出 57%至 115%。

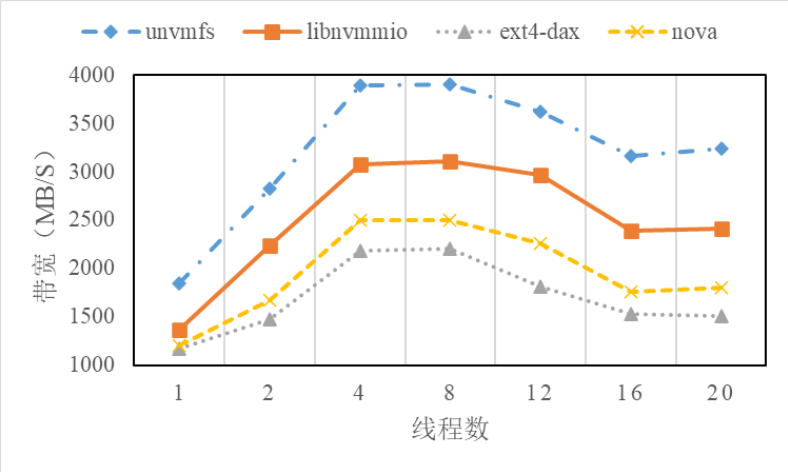

图 4-3　顺序写性能测试

随着线程数并发提高，UnvmFS 与其它系统性能差距拉大，得益于 UnvmFS 本地资源分配机制以及动态页面分配机制，能加大并发访问减少冲突。顺序写时，libnvmmio 同样默认 UNDO 策略会导致同步的日志和文件两次拷贝操作，且文件大小变动范围内会一直保持 UNDO 的日志策略。当文件大小超过预分配变动范围时，带来文件重新内存映射等操作，也会造成多次进入内核态处理，暂停用户对文件的写操作。文件重新映射过后，改变为 REDO 日志策略，不过比起追加写每次文件大小的改变，在文件大小变动范围内，增加的只是一次额外的拷贝操作开销，比起追加写时性能会略好。libnvmmio 还对每个文件创建了一个后台日志下刷线程，在大量打开文件时，占用了很多的系统资源；而且在多线程并发时，因频繁下刷数据加锁，总会与用户请求相互等待，造成响应时延变长。

而 UnvmFS 则没有上述烦恼，利用自己全局和局部页面分配池，快速给顺序写申请到数据页面，一次拷贝完成顺序写操作，置位旧数据页面失效，待后台清理回收即可。UnvmFS 使用定时回收和阈值回收两种策略，定时回收回收量小对用户进程影响小，而阈值回收发生次数少，比 libnvmmio 的后台策略要精简和高效很多。

nova 在顺序写性能表现略优于追加写，但是曲线轨迹与追加写时比较接近。而 ext4-dax 模式在顺序写和追加写时的曲线是基本一致的，想必经历的软件栈处理结构也是相同的，进入内核和额外的复杂软件栈是其性能低的主要原因。

（3）随机写

用户随机写场景下，UnvmFS 也是以追加日志的方式作为顺序写执行，但是会多出一些额外的处理操作，以及比较频繁的后台垃圾回收工作，性能优势会相对弱势一点。如图 4-4 所示，UnvmFS 相较于 libnvmmio 能够提供 17%至 32%的性能提升，比 nova 高出 45%至 71%，比 ext4-dax 性能则高出 52%至 101%。UnvmFS 峰值带宽略微降低，猜测随机写带来的写冲突有增加。从测试结果可知，UnvmFS 在随机写场景性能比 libnvmmio 提高较追加写和顺序写提高接近，两者皆是日志更新的文件系统。

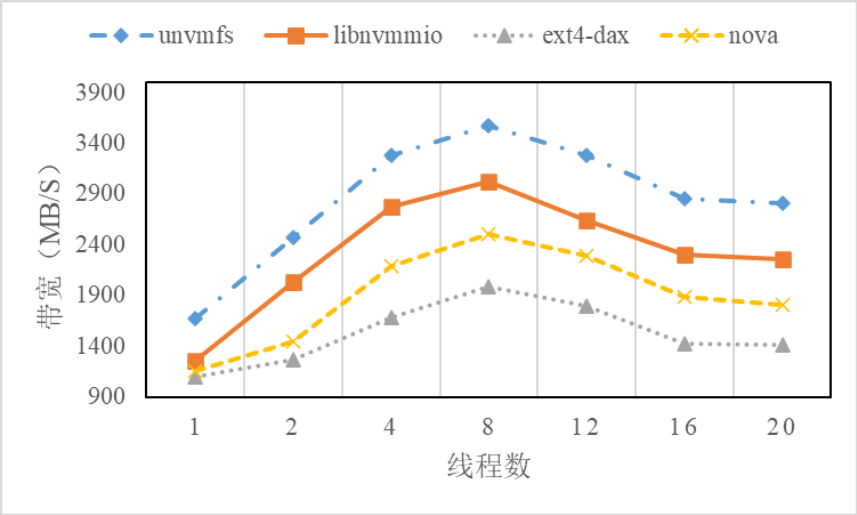

图 4-4　随机写性能测试图

分析原因，libnvmmio 在随机写场景下，涉及到文件大小改动较少，大部分被访问的文件，总是保持默认的 UNDO 日志策略，存在二次拷贝的问题。少部分被访问的文件，则偶有大小扩展和重新内存映射的情况发生。于是在随机写测试下，libnvmmio 和 UnvmFS 的主要性能差异在于 libnvmmio 多了一次数据拷贝，这本是较大的性能差异。但是 libnvmmio 有 UnvmFS 在随机写下没有的优势，libnvmmio 的并发粒度为文件内的数据块，在对同一个文件不同块的随机写下，在一定程度能保持较高的多线程并发，这也是其随机写性能与顺序写差距不大的原因之一。

nova 同样也是日志更新的文件系统，顺序与随机写性能差不多。ext4-dax 的性能

比起在追加写和顺序写时，在线程并发较大时与前三种文件系统差距要大一些。

（4）顺序读

顺序读性能比较如图 4-5 所示，可以看到 libnvmmio 相较于 UnvmFS 在大部分线程数性能略好，基本可以视为等同。UnvmFS 带宽比 nova 高出 27%至 95%，比起 ext4-dax 要高出更多，带宽高出 43%至 97%。

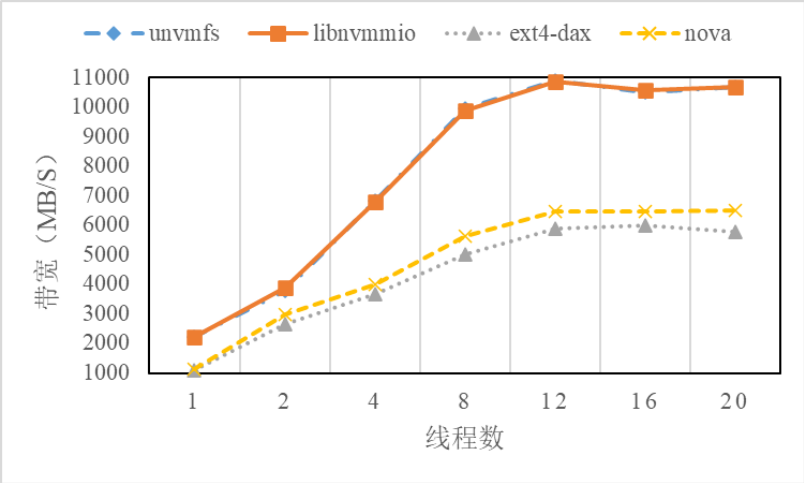

图 4-5　顺序读性能测试图

UnvmFS 与 libnvmmio 在顺序读下的性能差异几乎可以忽略不计。虽然原理上 libnvmmio 的顺序读的并发性较高，libnvmmio 在日志策略 UNDO 下读内存映射上来的文件内容，在 REDO 策略下读日志内容同样是内存映射的方式，都没有对单独文件的加锁，可以并发的执行单个文件的多个读操作。

UnvmFS 的日志是以文件为粒度的加锁以及查找前缀树的读取，但是加的读写锁，因此在完全的读负载下两者的并发度可谓是一样的，同一个文件允许多个线程同时读取。既然并发度是一样的，UnvmFS 与 libnvmmio 本质上都是在用户态空间以内存映射的方式读取文件的数据内容，性能上也不会有大的差异。而且因为测试的是 4 KB 大小的读取，UnvmFS 针对大的读取的一次前缀树查找，连续链表访问的读取优化的特性没有体现出来，若是对于大粒度顺序读 UnvmFS 的性能应该会略优于 libnvmmio。nova 和 ext4-dax 则由于自身软件栈的原因，性能远低于用户态文件系统。

（5）随机读

随机读性能测试如图 4-6 所示，UnvmFS 和 libnvmmio 的曲线比顺序读时贴合的

还是同样紧密，性能基本持平。UnvmFS 比 nova 带宽高 52%至 88%，比 ext4-dax 随机读带宽高出 72%至 182%。

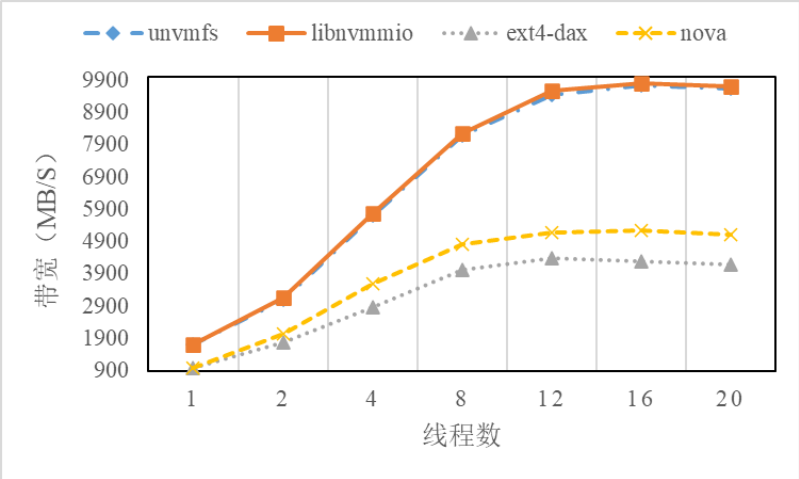

图 4-6　随机读性能测试图

随机读在各种文件系统下带宽都略低于顺序读，这主要是硬件 NVM 的机制原因而非软件原因。在 4 KB 大小的读取操作下，四种文件系统随机读性能测试效果与顺序读时的测试效果基本相同，可以看出随机读性能和顺序读性能图中随着线程数增加带宽曲线走向一致。

主要原因在于操作系统页面大小一般是 4 KB，于是很多文件系统沿用传统，使用 4 KB 作为基本的数据分配与使用页面，例如 UnvmFS 自行设计的内存池基本粒度是 4 KB 页面大小，nova 的数据页面也是 4 KB；libnvmmio 虽说实现了多种粒度的数据块，其范围也是在 4 KB 与 2 MB 之间成倍递增的，更不用说已经使用多年的 ext4 文件系统了。究其原因可能是因为在全读场景下，单个文件可以多线程并发读，而测试 I/O 大小又是 4 KB 页，所以顺序读和随机读在此效果类似。

4.3.3　混合负载测试

（1）文件服务器负载

文件服务器负载的特点是大块的读写操作且写比例占据主导位置，可以从图 4-7 看出在线程数为 20 时，几种文件系统在并发线程数达到 20 性能带宽开始较快下降，高比例写负载下在较低并发数下已经达到 NVM 访问的带宽，没法进一步提高性能。UnvmFS 比 nova 带宽高出 33%至 111%，内核态文件系统能达到的带宽与用户态文

件系统差距较大。UnvmFS 比 libnvmmio 带宽高出 13%至 22%，原因是 libnvmmio 在写占主导的负载 I/O 下，在执行 sync 操作或因文件大小超过变化范围重映射后，会有对日志策略转换的判断，从默认的 UNDO 策略转为 REDO 策略，此时写请求只需对日志写即可返回响应，后续后台线程会刷新到底层文件，且对于大块的数据 libnvmmio 有大块页面的优化处理。

UnvmFS 在数据更新上没有多大优势，但是 libnvmmio 以底层内核文件系统管理元数据有个比较致命的缺点，在真实负载下，打开、关闭、创建文件等元数据操作会比较频繁，而这会给 libnvmmio 带来反复的内存映射和取消映射操作，还有 libnvmmio 以文件为粒度的同步线程的不断创建和撤销，对性能产生影响。

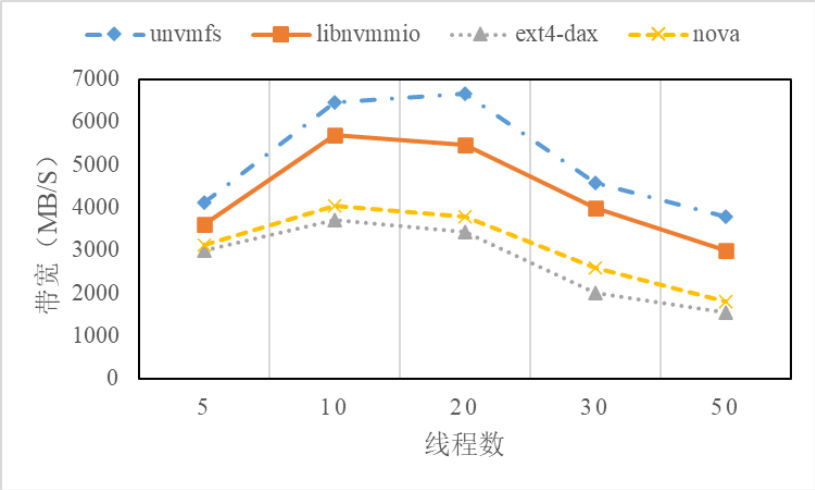

图 4-7　FILESERVER 性能测试图

UnvmFS 在用户态进行元数据处理的优势就体现出来了，libnvmmio 反复的内存映射和日志同步底层文件的拷贝操作严重增加了延时，UnvmFS 全程在用户态一次拷贝的方式，以最精简的软件栈实现最高的性能。nova 和 ext4-dax 此负载下带宽接近。UnvmFS 比 ext4-dax 的带宽高出 38%至 146%，高并发下更能考验出系统性能优势。

（2）邮件服务器负载

邮件服务器是读写均衡的负载，用户总是对整个邮件的读写，读写操作总是以顺序方式为主。UnvmFS 能支持小于等于 2 MB 的写操作，更大的写操作会被拆分成多个写操作执行，而在此负载下，单个邮件的写操作在文件系统侧也无需拆分。在细粒

度频繁的元数据文件操作，UnvmFS 都可以在用户态直接完成，整个 UnvmFS 运行期间的所有操作都不用经过内核。

libnvmmio 在读写均衡的负载下，基本使用的是 UNDO 日志策略，在写操作时会有二次拷贝的问题存在，而在小文件频繁的文件的一些打开、关闭操作，也会带来频繁的内核态操作，增加用户响应时延。小文件频繁的读写，以及小文件频繁的元数据操作，对于 ext4-dax 的影响更大，频繁的用户态和内核态切换本身就是对性能的影响。

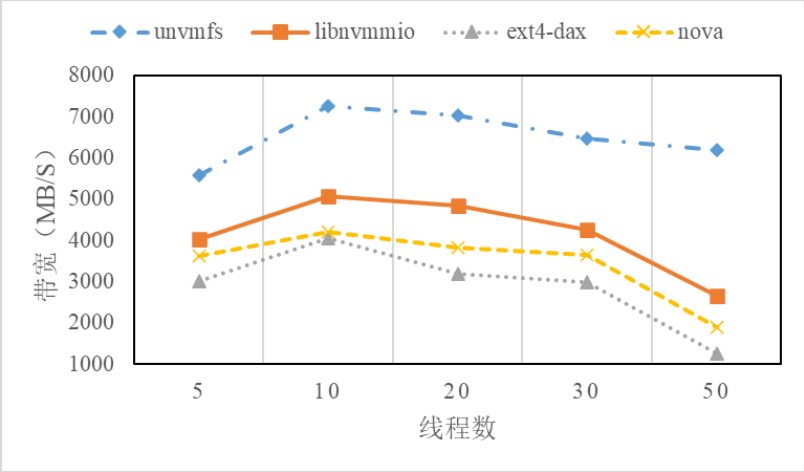

图 4-8　VARMAIL 性能测试图

测试性能对比如图 4-8 所示，UnvmFS 的带宽比 libnvmmio 高出 43%至 134%，UnvmFS 的带宽比 nova 高出 53%至 214%，比 ext4-dax 高出 78%至 389%。UnvmFS 在小粒度负载下性能要远好于 nova 和 ext4-dax，从侧面也可看出小粒度的文件读写对内核态的文件系统性能影响很大。libnvmmio 在某些线程数并发下虽比 ext4-dax 和 nova 性能高出接近一倍，但三者虽线程数并发的曲线走向基本一致，且越来越接近，说明并发越高，进入内核态处理的次数也越多，性能也下降变快。

（3）网页代理服务器负载

网页代理服务器是典型的存储的数据是半结构化的，因此读写数据的粒度是一样的，且数据写入后一般不会有改动，多是对数据的读取操作。webproxy 负载测试结果如图 4-9 所示，UnvmFS 相较 libnvmmio 的带宽高出 16%至 39%，在读操作占据约 80%时，因为 webproxy 的读写粒度未超过兆字节，每次读取时，在 UnvmFS 和

libnvmmio 都是对文件加一次读写锁就可开始读取操作，都是在用户态执行，读取性能不会相差多少，这也可以从前面的顺序和随机读取操作的性能测试中看出。而写操作占据 20%左右时，libnvmmio 默认使用的 UNDO 策略带来很大的性能差，影响到性能的输出。

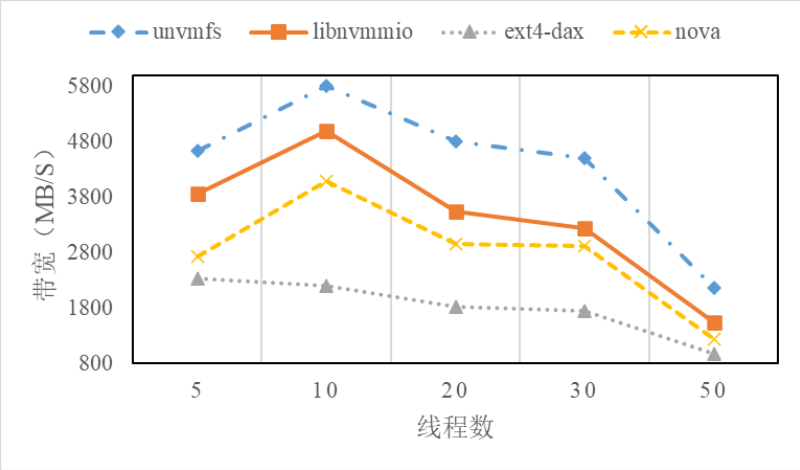

图 4-9　WEBPROXY 性能测试图

UnvmFS 比 nova 带宽高出 42%至 74%，nova 在线程数 5 的低并发时性能较差，其它并发时能接近 libnvmmio 的性能，此负载下 UnvmFS 和 nova 的差异主要在于内核切换的开销。UnvmFS 的带宽与 ext4-dax 相比高出 100%至 164%，同为日志结构型文件系统，精简的软件栈在非易失存储器下才能充分发挥出性能优势。webproxy 负载下的测试有一个特点，从线程并发度 10 开始，随着线程数增加并发度增加，各个文件系统性能都是直接在下降的，说明在 webproxy 负载下较低并发度就达到了文件系统性能峰值，估计是因为对较集中文件的频繁访问所导致的并发度无法提高。

（4）网页服务器负载

在图 4-10 中所示，UnvmFS 带宽高出 libnvmmio 约 3.4%至 6%，UnvmFS 比 ext4-dax 带宽高 17%至 31%，比 nova 高约 13%至 18%。网页服务器负载是读取比例最高的负载高达 90%，且读操作的粒度较大达到了 256 KB，从该特点也可看出 UnvmFS 比起 libnvmmio 带宽高出比例较邮件服务器负载较低的原因，UnvmFS 比起 libnvmmio 主要在于元数据的改动和数据写操作，在读操作下是没有多大优势的。ext4-dax 和 nova 的性能比起 UnvmFS 和 libnvmmio 一如既往的差一些，不过在

webserver 负载下带宽要高出在其它负载下不少，最低带宽都达到了 7000 MB 以上，这也说明对于大块的读操作文件系统的性能是很好的。

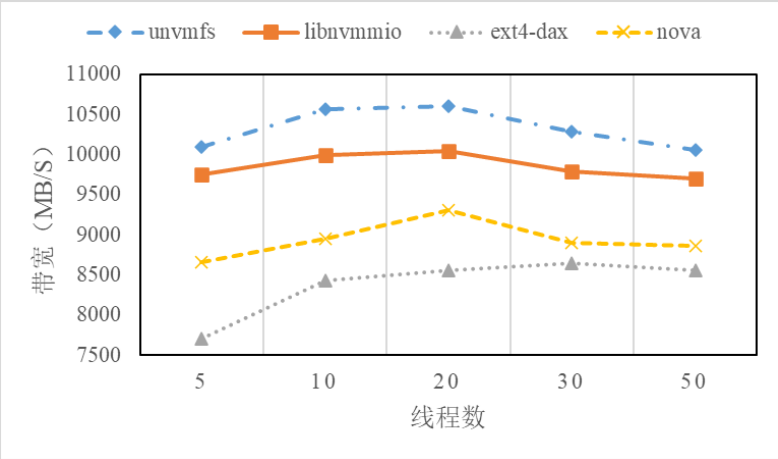

图 4-10　WEBSERVER 性能测试图

该负载下曲线还有一个特点，所有文件系统带宽没有随着线程数的增加，而导致带宽急剧的降低，而是并发度能随着线程数的增加维持一个较平衡的状态。nova 带宽与 ext4-dax 差距较小，猜测原因是在大粒度读操作时，nova 需要频繁的对日志索引进行访问查找数据增加开销，降低并发。webserver 负载下各文件系统的带宽都高于其它类型负载，该负载的访问模式应该是最为适合文件系统的访问模式。

4.4　本章小结

本章是对整个论文 UnvmFS 文件系统设计与实现可行性的验证。开始小节先给出了文件系统的测试环境以及对比方案，接着介绍了文件系统权威测试工具 Filebench，拟定了基本操作的 microbenchmark 和模拟真实负载的 macrobenchmark 测试内容。在测试了各种负载后，对测试的结果进行了分析与说明，在绝大多数场景下 UnvmFS 都能保持比其它文件系统更高的性能，UnvmFS 基于非易失存储设备的用户态文件系统的优势得到了有效验证。

5 总结与展望

5.1 本文主要内容及创新点

云计算、边缘计算、大数据、5G 等新兴概念和技术的兴起，对数据中心的存储提出了更高的需求，尽管现在是分布式存储的时代，依然离不开单机服务器的存储，文件系统是数据访问重要的一环。近两年英特尔商业化的非易失内存设备 AEP，接近于 DRAM 的访问速度和接近于硬盘的存储容量。将 NVM 与文件系统结合，将给已经成熟的文件系统带来更高性能的可能性，而如何充分挖掘 NVM 的性能和特点，适配文件系统语义，是本文的重要研究工作。

传统文件系统软件栈的时延开销对于高访问性能的 NVM 是不可忽略的，简化文件系统软件栈是性能大幅提升的关键。基于 NVM 的文件系统依旧应该保证 POSIX 标准的文件系统操作原子语义，从而在掉电等特殊情况维护文件系统数据的一致性。针对现有的这些问题，再结合分析了 Aerie、Splitfs、Libnvmmio 等新的用户态文件系统工作的优缺点后，本文提出了一个基于 NVM 的用户态文件系统 UnvmFS，UnvmFS 实现了内存映射的方式在用户态访问数据和元数据，以 inode 节点为粒度的高并发访问，以及以日志结构的方式实现文件系统保证系统掉电数据不丢失，维护数据一致性。

本文主要有如下工作成果：

（1）对现有的一些基于非易失存储设备的内核态和用户态文件系统的研究工作进行了详细剖析，分析了现有系统存在的不完整性、非原子更新、性能还能继续挖掘等问题，也借鉴了日志结构、内存映射、原子更新等先进思路和技术。

（2）设计了一套用户态文件系统 UnvmFS，整个非易失存储空间映射到用户态地址空间，借此文件系统元数据的访问也放到用户态，进一步精简软件栈，完全没有进入内核的开销，实现性能最优化；兼容 POSIX 文件系统标准，以日志结构的方式实现文件系统，从而实现访问操作的原子性。

（3）在 Linux 系统下实现了 UnvmFS，不仅实现了文件系统基本操作的访问流程，还实现了高速的垃圾回收功能和自动动态的存储页面分配机制。

（4）最后使用了文件系统标准测试工具 Filebench 来对 UnvmFS 的性能进行了衡量，与基于非易失存储设备的内核态文件系统 Ext4-Dax、Nova 和用户态文件系统 Libnvmmio 进行了测试对比，测试结果表明，UnvmFS 的性能有很大优势。

本文创新点总结如下：

（1）实现文件系统操作全用户态访问，精简软件栈；通过自设计的非易失分配器，分配和管理文件系统必需的元数据，解决用户态下进程间地址空间独立难以共享数据的问题，消除文件系统元数据同步开销。

（2）以日志结构方式实现文件系统文件操作原子性，并以轻量级写前日志拷贝元数据结构实现目录的原子性，以高性能的方式维护文件系统操作的原子语义。

（3）持久化元数据，消除文件系统卸载后重新挂载的重构开销，也最大限度降低系统掉电或崩溃后的恢复开销。

5.2 未来工作展望

UnvmFS 重心在于完整实现一个元数据和数据操作都不经过内核态，消除冗余软件层直接访问 NVM 的文件系统，有充分发挥出非易失存储设备的性能优势。但由于时间关系，想要实现一个可靠能实际应用的非易失存储设备的文件系统，后续还有一些完善和改进的工作，大致如下：

（1）现代处理器会重排序指令来提升性能，CPU 缓存的一致性模型不提供对 NVM 设备更新的一致性保护，掉电可能会造成数据不一致状态，在之后的工作中需要借助 clflush、mfence 指令等指令强制下刷数据并保证 CPU 缓存写顺序。

（2）文件系统实现在用户态后，拥有高权限内核态的进程乱写或者恶意用户窃取数据可能引起文件系统数据错误或暴露。

（3）UnvmFS 只实现了自己的一套简单的非易失内存池，供文件系统的一些数据结构持久化分配使用，比起内核的内存分配算法相差甚远，需要大力改进。

致 谢

光阴飞逝，2014 年秋天踏入华中科技大学进行本科学习，到 2018 年秋天继续留在本校攻读研究生学位，在华中科技大学的七年时间里，从对自己以后的人生路线毫无规划，到现在坚定地走向计算机存储领域继续深耕，为自己热爱的方向不断前进。能有现在扎实的基础和坚定的目标，离不开母校浓厚的学术氛围，离不开老师们的悉心指导，也离不开同学们的大力帮助，是他们让我的人生更丰富多彩，是他们让我的目标更加高远，也是他们让我拥有了更多的可能性。

良师是我科研道路上的指路明灯。首先，感谢我的导师谭志虎教授，在我读研的三年时间，在学术上给我提供前沿的科研方向、良好的科研环境和耐心的交流指导，培养我严谨细致的态度和敢于实践、乐于创新的科研精神；在生活上给予我无微不至的关心。感谢万继光教授在论文开题和撰写中对我的指导与帮助，万老师专业积累深厚、平易近人，与万老师的学术交流和探讨中受益匪浅。感谢实验室谢长生、吴非、曹强、肖亮、胡迪青等老师在我科研道路上的指导和帮助。

益友是我在学校的生活助力。感谢朱承浩学长，带领我踏入实验室科研项目的第一步，感谢张艺文和汤陈蕾学长在我论文设计实现当中的帮助，感谢李大平、徐鹏学长、盛涛涛、刘志文、鲁凯同学、张鑫、费长红、杨豪迈学弟在实验室科研以及生活上的支持与帮助。感谢实验室的所有同学，一起营造的浓厚科研氛围和互帮互助的生活环境。感谢与我一起相伴七年的同学王子轲和卢浩迪，是你们在我低落不顺时的鼓励，生活乐事的分享和一起约玩的欢乐，让我觉得交到如此挚友，我的本科和研究生生活不留遗憾。

父母是我朝目标前进的动力。能有现在的我，离不开父母背后的支持与鼓励，他们总在背后默默倾听我的喜怒哀乐，尊重和毫无私心地支持我的每一个决定，父母的爱是永远无法用言语表达足够的，感谢你们让我来到了这个世界。

参考文献

[1] Z. Liu, A. Zhao, M. Liang. A Port-Based Forwarding Load-balancing Scheduling Approach for Cloud Datacenter Networks. Journal of Cloud Computing, 2021, 10(1): 1-14

[2] H. He, Y. Zhao, S. Pang. Stochastic Modeling and Performance Analysis of Energy-aware Cloud Data Center Based on Dynamic Scalable Stochastic Petri Net. Computing and Informatics, 2020, 39(1-2): 28-50

[3] Y. Tanimura, S. Takizawa, H. Ogawa, T. Hamanishi. Building and Evaluation of Cloud Storage and Datasets Services on AI and HPC Converged Infrastructure In: 2020 IEEE International Conference on Big Data (BigData 2020), Atlanta, GA, USA, December 10-13, 2020, IEEE, 2020: 1992-2001

[4] K. Benzina. Cloud Infrastructure-as-a-Service as an Essential Facility: Market Structure, Competition, and the Need for Industry and Regulatory Solutons. Berkeley Technology Law Journal, 2019, 34(1): 3-3

[5] 袁夫, 郭红. 从"摩尔定律"看 CPU 工艺发展之殇. 人文之友, 2019, 000(009): 13-13

[6] 赵晓菲. 略谈 NVM 用于内存计算缓冲器的几种设计. 新一代：理论版, 2019, 000(012): 129-129

[7] S. Yu. Neuro-inspired Computing with Emerging Nonvolatile Memorys. Proceedings of the IEEE, 2018, 106(2): 260-285

[8] K. Oe, T. Nanri. Non-volatile Memory Driver to Drastically Reduce Input-output Response Rime and Maintain Linux Device-mapper Framework. International Journal of Networking and Computing, 2020, 10(2): 127-143

[9] A. V. Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, et al. Managing Non-volatile Memory in Database Systems. In: Proceedings of the 2018 International Conference on Management of Data (SIGMOD 2018), Houston, TX, USA, June 10-15, 2018, ACM, 2018: 1541-1555

[10] J. Kong, H. Zhou. Improving Privacy and Lifetime of PCM-based Main Memory. In: Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems

& Networks (DSN 2010), Chicago, IL, USA, June 28 - July 1, IEEE, 2010: 333-342

[11] Y. Du, M. Zhou, R. Bruce, M. Daniel, G. Rami. R. Melhem. Bit Mapping for Balanced PCM Cell Programming. Computer Architecture News, 2013, 41(3): 428-439

[12] R. Sbiaa, H. Meng, S. N. Piramanayagam. Materials with Perpendicular Magnetic Anisotropy for Magnetic Random Access Memory. Physica Status Solidi- Rapid Research Letters, 2011, 5(12): 413-419

[13] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, et al. Spin-Transfer Torque Magnetic Random Access Memory (STT-MRAM). ACM Journal on Emerging Technologies in Computing Systems, 2013, 9(2): 1-35

[14] 赵巍胜, 王昭昊, 彭守仲. STT-MRAM 存储器的研究进展. 中国科学:物理学 力学 天文学, 2016, 46(10): 70-90

[15] S. Bagheri, A. A. Asadi, W. Kinsner, N. Sepehri. Ferroelectric Random Access Memory (FRAM) Fatigue Test with Arduino and Raspberry Pi. In: Proceedings of International Conference on Electro Information Technology (EIT 2016), Grand Forks, ND, USA, May 19-21, 2016, IEEE, 2016: 313-318

[16] 宋玲. RRAM 的阻变特性研究. 微处理机, 2014, 35(04): 24-25

[17] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, et al. Better I/O Through Byte-Addressable, Persistent Memory. In: Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP 2009), Big Sky, Montana, USA, October 11-14, 2009, ACM, 2009: 133-146

[18] D. S. Rao, S. Kumar, A. S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, et al. System Software for Persistent Memory. In: Proceedings of the 9th European Conference on Computer Systems (EuroSys 2014), Amsterdam, The Netherlands, April 13-16, 2014, ACM, 2014: 1-15

[19] A. Sánchez-Macián, L. A. Aranda, P. Reviriego, V. Kiani, J. A. Maestro. Enhancing Instruction TLB Resilience to Soft Errors. IEEE Transactions on Computers, 2019, 68(2): 214-224

[20] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In: Proceedings of the Linux Symposium, Ottawa, Ontario, Canada, June 27th–30th, 2007, ACM, 2007: 21-34

[21] X. Wu, S. Qiu, A. L. N. Reddy. SCMFS: A File System for Storage Class Memory and its Extensions. Acm Transactions on Storage, 2013, 9(3): 1-23

[22] A. R. Cunha, C. N. Ribeiro, J. A. Marques. The Architecture of a Memory Management Unit for Object-oriented Systems. Acm Sigarch Computer Architecture News, 1991, 19(4): 109-116

[23] J. Xu, S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 2016), Santa Clara, CA, USA, February 22-25, 2016, USENIX Association, 2016: 323-338

[24] J. Xu, L. Zhang, A. S. Memaripour, A. Gangadharaiah, A. Borase, T. B. D. Silva, et al. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In: Proceedings of th 26th Symposium on Operating Systems Principles (SOSP 2016), Shanghai, China, October 28-31, 2017, ACM, 2017: 478-496

[25] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, M. M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In: Proceedings of the 9th European Conference on Computer Systems (EuroSys 2014), Amsterdam, The Netherlands, April 13-16, 2014, ACM, 2014: 1-14

[26] M. Dong, H. Bu, J. Yi, B. Dong, H. Chen. Performance and Protection in the ZoFS User-space NVM File System. In: Proceedings of the 27th Symposium on Operating Systems Principles (SOSP 2019), Huntsville, ON, Canada, October 27-30, 2019, ACM, 2019: 478-493

[27] R, Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, V. Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In: Proceedings of th 27th Symposium on Operating Systems Principles (SOSP 2019), Huntsville, ON, Canada, October 27-30, 2019, ACM, 2019: 494-508

[28] J. Choi, J. Hong, Y. Kwon, H. Han. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In: Proceedings of the 2020 USENIX Annual Technical Conference (ATC 2020), Virtual Event, July 15-17, 2020, USENIX Association, 2020: 1-16

[29] F. T. Hady, A. P. Foong, B. Veal, D. Williams. Platform Storage Performance with 3D XPoint Technology. Proceedings of the IEEE, 2017, 105(9): 1822-1833

[30] J. Yang, B. Li, D. J. Lilja. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. ACM Transactions on Modeling and Performance Evaluation of Computing Systems, 2020, 5(1): 1-28

[31] 郝嘉, 候梦清. NVMe 存储协议浅论. 信息系统工程, 2020, 315(03): 162-163

[32] Z. Mao, S. Zheng, L. Huang, Y. Shen. A DAX-enabled Mmap Mechanism for Log-structured In-memory File Systems. In: Proceedings of th 36th International Performance Computing & Communications Conference (IPCCC 2017), San Diego, CA, USA, December 10-12, 2017, IEEE Computer Society, 2017: 1-8

[33] B. Nesterenko, X. Liu, Q. Yi, J. Zhao, J. Zhang. Transitioning Scientific Applications to Using Non-volatile Memory for Resilience. In: Proceedings of the International Symposium on Memory Systems (MEMSYS 2019), Washington, DC, USA, September 30 - October 03, 2019, ACM, 2019: 114-125

[34] W. Wang, S. Diestelhorst. Quantify the Performance Overheads of PMDK. In: Proceedings of the International Symposium on Memory Systems (MEMSYS 2018), Old Town Alexandria, VA, USA, October 01-04, 2018, ACM, 2018: 50-52

[35] S. Scargall. Introducing the Persistent Memory Development Kit. Apress, Berkeley, CA: Programming Persistent Memory, 2020: 63-72

[36] K. Munegowda, G. T. Raju. Avoidance Techniques for Snowball Effect of Wandering Tree in Flash Memory Based File Systems. International Journal of Applied Engineering Research, 2015, 10(86): 120-120

[37] H. Kim, H. Y. Yeom, Y. Son. An Efficient Database Backup and Recovery Scheme Using Write-Ahead Logging. In: Proceedings of the 13th IEEE International Conference on Cloud Computing (CLOUD 2020), Virtual Event, 18-24 October 2020, IEEE, 2020: 405-413

[38] K. Han, H. Kim, D. Shin. WAL-SSD: Address Remapping-Based Write-Ahead-Logging Solid-State Disks. IEEE Transactions on Computers, 2019, 69(2): 260-273

[39] M. Rosenblum, J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems, 1992, 10(1): 26-52

[40] Z. Zhang, D. Feng, Z. Tan, L. T. Yang, J. Zheng. A Light-weight Log-based Hybrid Storage System. Journal of Parallel and Distributed Computing, 2018, 118: 307-315

[41] M. K. McKusick. The Virtual Filesystem Interface in 4.4BSD. Computing Systems, 1995, 8(1): 3-25

[42] J. Boyar, M. R. Ehmsen, J. S. Kohrt, K. S. Larsen. A Theoretical Comparison of LRU and LRU-K. Acta Informatica, 2010, 47(7): 359-374

[43] N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. in: Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 2009), San Francisco, CA, USA, February 24-27, 2009, USENIX, 2009: 125-138

附录 1　攻读硕士学位期间取得的研究成果

发表与接收论文

[1] Daping Li, Jiguang Wan, **Duo Wen**, Chao Zhang, Nannan Zhao, Fei Wu and Changsheng Xie. PreMatch: An Adaptive Cost-Effective Energy Scheduling System for Data Centers. In Proceedings of the 36th IEEE Symposium on Mass Storage Systems and Technologies. MSST 2020（CCF 推荐 B 类国际会议）

附录 2　攻读硕士学位期间参加的科研项目

1. 华为技术有限公司杭州研究所合作项目

项目名称：S2-RAID 和 SSD 冷热数据分层技术研究

项目编号：YBN9105082

起止时间：2019 年 11 月至 2020 年 11 月

担任角色：方案设计、代码开发、结项文档撰写

2. 北京平凯星辰科技发展有限公司合作项目

项目名称：基于 AEP 与 SSD 的混合 KV 系统软件研发

项目编号：20181512

起止时间：2018 年 08 月至 2019 年 07 月

担任角色：方案设计、代码开发