上海交通大学博士学位论文

基于非易失性内存的文件系统研究

博士研究生:
学　　号:
导　　师:
申请学位: 工学博士
学　　科: 计算机科学与技术
所在单位: 电子信息与电气工程学院，软件学院
答辩日期:
授予学位单位: 上海交通大学

Dissertation Submitted to Shanghai Jiao Tong University
for the Degree of Doctor

NON-VOLATILE MEMORY FILE SYSTEM RESEARCH

Candidate:

Student ID:

Supervisor:

Academic Degree Applied for: Ph.D. of Engineering

Speciality: Computer Science

Affiliation: School of Software, SEIEE

Date of Defence:

Degree-Conferring-Institution: Shanghai Jiao Tong University

基于非易失性内存的文件系统研究

摘　要

非易失性内存是一种新型存储设备。其既有存储设备持久保存大容量数据的能力，又有内存访问速度块、以字节寻址的特点，正逐步被部署在数据中心的服务器中，成为大数据背景下承担海量、高速数据存储功能的重要存储设备。在系统层面来看，非易失性内存结合了外部存储与内存的优势，正在改变传统的“CPU 缓存–内存–存储”的存储层次。非易失性内存可字节寻址的特性，使得对非易失性内存的访问不必再以普通内存作为“字节粒度”到“块粒度”中间层；同时，由于非易失性内存接近普通内存的性能，对非易失性内存的访问也不再需要使用普通内存作为缓存。这些改变缩短了传统的存储层次，使得“CPU 缓存–存储（非易失性内存）”的存储层次成为可能。而存储层次的坍缩，打破了文件系统等系统软件对存储设备的传统假设，为文件系统的设计提出了新的需求，为进一步提升文件系统的数据存储性能提供了机会。

为了充分发挥非易失性内存的性能优势，研究人员开始研究针对非易失性内存特性的新型文件系统。由于高性能的传统文件系统一般作为内核模块运行在内核空间，对新型非易失性内存文件系统的研究从内核态文件系统开始，研究人员将日志机制、写时复制技术、日志结构等维护文件系统崩溃一致性的技术在非易失性内存文件系统中进行的相应的优化，设计和实现出一系列新型内核态非易失性内存文件系统。然而，由于非易失性内存直接以 CPU 的 LOAD、STORE 等指令进行访问，CPU 对非易失性内存的写入可能会被 CPU 缓存乱序写回到非易失性内存之中，从而以错误的顺序被持久化。因此，为了保证文件系统崩溃一致性，这些新型文件系统需要在处理系统调用时使用同步的 CLFLUSH 等缓存行刷除指令，导致系统调用关键路径上的时延

增长，降低了在高速非易失性内存上的文件系统性能。**如何在保证文件系统崩溃一致性的情况下，避免同步缓存刷除指令对文件系统性能的影响，是进一步提升内核态非易失性内存文件系统性能过程中的重要问题。**

对内核态非易失性内存文件系统的研究，进一步促发了研究人员对极致发挥非易失性内存性能的追求。然而研究人员很快发现，在文件系统的操作中，系统调用造成的上下文切换和内核中虚拟文件系统中复杂的逻辑，逐渐成为发挥非易失性内存性能优势的瓶颈。考虑到非易失性内存具有可字节寻址的特性，其可以在用户态进行访问。因此，用户态非易失性内存文件系统逐渐成为研究热点。然而由于用户态文件系统与应用程序处于同一个内存空间之中，为了防止应用程序有意或无意地修改文件系统的结构，用户态非易失性内存文件系统一般无法直接修改文件系统元数据，以保护文件系统的安全性、完整性和一致性。这种限制导致用户态文件系统的元数据依然需要进入内核空间、或通过通讯由可信的进程进行，影响了用户态文件系统的性能。**如何在保障文件系统安全性和完整性的同时，赋予用户态非易失性内存文件系统对元数据的完全控制，是为了完全发挥非易失性内存性能所需要研究的核心问题。**

虽然性能是文件系统所追求的重要指标之一，但是实际生产生活中的文件系统并非只需要高性能。成熟度、可靠性、兼容性、易用性等因素都是一个文件系统被实际应用时所需要考虑的问题，因此新型非易失性内存文件系统需要多年的积累和使用，才能够逐渐成熟，被应用到实际生产之中。而在此之前，将成熟的传统文件系统为非易失性内存进行针对性的优化，是应用非易失性内存的最可靠方法之一。Ext4、XFS 等成熟的传统文件系统已经针对非易失性内存的特性增加了“直接访问”模式，然而在实际应用中，其性能依然与为非易失性内存设计的新型内核态文件系统有所差距。**如何探索导致性能差距的根本问题，并在尽量少改动的情况下去除现有成熟传统文件系统在非易失性内存上的性能瓶颈，是实际应用非易失性内存时需要解决的关键**

问题。

针对上述的三个问题，本文分别从三个角度对基于非易失性内存的文件系统展开较为深入的研究，包括新型内核态非易失性内存文件系统、新型用户态非易失性内存文件系统、传统文件系统在非易失性内存上的优化。具体来说，本文的研究内容和贡献如下：

第一，针对关键路径上的同步缓存行刷除问题，本文提出**新型内核态异步非易失性内存文件系统 SoupFS**，通过推迟文件系统操作的持久性保证，在保证崩溃一致性的情况下，消除了关键路径上的同步缓存行刷除指令。SoupFS 将软更新技术与非易失性内存的特点相结合，通过软更新技术允许文件系统修改的延迟写入，在关键路径中消除元数据的持久化和同步缓存刷除操作；利用非易失性内存的可字节寻址特性，设计使用更高效的文件系统组织结构，与软更新技术中指针粒度的依赖性追踪相契合，简化了软更新技术中原本非常复杂的依赖追踪和依赖保证。为了解决软更新技术中双视图对页缓存的依赖，SoupFS 还提出了基于指针的双视图，在同一结构上通过不同指针组成两种不同的视图，以支持元数据的延迟持久化、依赖追踪和依赖保证。实验结果表明，与现有的内核态非易失性内存文件系统相比，SoupFS 显著降低了文件系统操作的时延，并提供了更高的吞吐量。

第二，针对用户态文件系统无法直接管理文件系统元数据问题，本文提出**新型用户态非易失性内存文件系统框架 Treasury**。通过对应用程序的数据文件进行调研，本文发现应用程序倾向于以相似的文件权限保存其文件，同时这些文件的权限很少被修改。基于这个发现，本文提出一个新的抽象 Coffer，用于管理一组具有相同访问权限的非易失性内存资源。通过使用 Coffer 抽象，Treasury 框架将非易失性内存的保护和管理分开，内核中的模块 KernFS 负责以 Coffer 为粒度对非易失性内存提供强保护，同时赋予用户态非易失性内存文件系统 μFS 对 Coffer 内的文件系统结构（包括数据和元数据）的完全管理权。本文还通过结合内存保护键等机制，解决由于用户态对 Coffer 的完全管理权带来的潜在误写和恶意攻击问题，进一步增强对文件系统结构的

保护和隔离。Treasury 框架支持动态链接程序不经修改直接运行在高性能的用户态非易失性内存文件系统之上，并为实现各种不同结构的用户态非易失性内存文件系统提供便利。在 Treasury 框架中，本文实现了一个示例用户态非易失性内存文件系统 ZoFS。测试结果表明，使用 Treasury 框架的 ZoFS 在基准测试和现实应用程序中均表现出更优的性能。在部分测试中，其对文件的访问性能达到了非易失性内存访问带宽的上限。

第三，针对成熟文件系统在非易失性内存上的性能瓶颈问题，本文提出**为非易失性内存优化的传统文件系统 Ext4pm**。通过对成熟的传统文件系统 Ext4 在非易失性内存上进行测试和分析，并与新型非易失性内存文件系统 NOVA 进行对比，本文识别出 Ext4 在非易失性内存上的性能问题，并将问题归纳为三个造成性能瓶颈的根本原因：过度抽象、非相称的实现、不可伸缩的设计。针对这些根本原因，本文提出多种优化策略，并根据策略应用补丁或实现优化，解决 Ext4 在非易失性内存上的性能问题。通过不足 1,000 行的优化修改，为非易失性内存优化的文件系统 Ext4pm 在微基准测试集 FxMark、综合基准测试集 Filebench 和真实应用 RocksDB 中的性能超出高度优化过的 Ext4 分别达 7.7 倍、1.12 倍和 22%，性能接近甚至优于 NOVA 等新型非易失性内存文件系统。在真实应用 RocksDB 的性能测试中，Ext4pm 的性能超过 NOVA 达 8%。

本文中的三个研究内容分别从新型内核态文件系统、新型用户态文件系统和成熟文件系统的优化三个角度对非易失性内存在文件系统中的应用进行了较为深入的研究，并针对不同的应用场景有所侧重。内核态异步非易失性内存文件系统 SoupFS 较为均衡。其既通过成熟的系统调用接口实现了完整的文件系统操作，又针对非易失性内存的特点设计了全新的结构，较好地发挥了非易失性内存的性能。针对对性能要求极高的场景，使用用户态非易失性内存文件系统框架 Treasury 可以避免系统调用、上下文切换和虚拟文件系统的开销，最大程度地发挥非易失性内存的优势。同时，使用 Treasury 还可以根据应用程序

的实际需求，定制化用户态非易失性内存文件系统，对文件系统结构进行有针对性的设计，以提升应用程序使用非易失性内存存储数据的效率。对于可靠性、稳定性要求较高，但希望利用非易失性内存的性能优势的场景，可使用基于成熟传统文件系统、为非易失性内存优化的 Ext4pm。其通过少量的代码修改，在不改变 Ext4 的存储布局的情况下，对 Ext4 的文件操作的处理流程进行了优化，实现了在非易失性内存上的性能提升。其兼容 Ext4 的存储格式，可直接使用为 Ext4 设计的各种工具进行管理、调优和监控。此外，本文中使用的基于非易失性内存的文件系统设计和优化思想，同样可以拓展到其他新型存储介质和其他场景中，为大数据时代高速文件系统的设计实现、优化和演化提供了新的方法和思路。

关键词：非易失性内存，文件系统，存储系统

NON-VOLATILE MEMORY FILE SYSTEM RESEARCH

ABSTRACT

Non-volatile memory is a kind of emerging storage device that combines both the durability of traditional high-volume storage devices and the high performance and byte-addressability of DRAM. Non-volatile memory has been being deployed in data center servers as one of the primary storage devices to enable high-volume and high-performance data storage in the big data era. From the perspective of computer systems, non-volatile memory combines the advantages of both external storage and internal runtime DRAM, revolutionizing the traditional storage hierarchy of "CPU cache – DRAM – storage". With the byte-addressability of non-volatile memory, accesses to non-volatile memory drop the necessity of using DRAM as the intermediate layer for transforming between bytes and blocks. Meanwhile, the near-DRAM high performance of non-volatile memory reduces the advantages of DRAM as a data cache for bridging the speed gap between CPU and storage. Both these two changes collapse the storage hierarchy, making the new hierarchy of "CPU cache – storage (non-volatile memory)" possible and practical. The collapse of the storage hierarchy also breaks the traditional assumption of storage devices made by the system software, raising new requirements for the design of file systems and providing opportunities to improve file systems' data storage performance.

To fully exploit the performance benefit of non-volatile memory, researchers start to study the design of new file systems running on non-volatile memory. As traditional high-performance file systems usually running as kernel modules in the kernel space, research starts from designing new non-volatile file systems in the kernel space. By adapting and optimizing

crash consistency techniques, including journaling, copy-on-writes, and log-structured file systems, on non-volatile memory, a series of new kernel-space non-volatile memory file systems are designed and implemented. However, as non-volatile memory can be directly accessed via CPU `LOAD` and `STORE` instructions, writes issued by CPU can be reordered by the CPU cache when writing back to non-volatile memory. To ensure the file system crash consistency, existing new non-volatile memory file systems need to issue synchronous `CLFLUSH` instructions to enforce that the data is evicted from the cache and reaches the non-volatile memory. These synchronous `CLFLUSH` instructions prolong system calls' critical path and reduce the file system performance on high-performance non-volatile memory. **As a result, in order to further improve the performance of kernel-space non-volatile memory file systems, it becomes one of the key challenges to avoid the costly synchronous cache flushes in the critical path while retaining the file system crash consistency.**

The research on kernel-space non-volatile memory file systems further promotes the pursuit of exploiting the extreme performance of non-volatile memory. Researchers find that within the processing of file system requests, the context switches and the complexity of the kernel virtual file system become the bottlenecks of exploiting non-volatile memory performance. Considering that the byte-addressability feature of non-volatile memory enables its access in user space, user-space non-volatile memory file systems become one hot research topic. However, as user-space file systems and applications share the same memory address space, to prevent applications from altering the file system structures intentionally or unintentionally, user-space non-volatile file systems are usually not allowed to update file system metadata directly in order to preserve file systems' security, integrity and consistency. User-space non-volatile file systems have to update file system metadata by entering the kernel space or via inter-process communications, which in-

evitably affects the file system performance. **In order to fully utilize non-volatile memory performance, it becomes a core challenge to provide the user-space non-volatile memory file system full control over file system metadata while retaining the security and integrity of file systems.**

Although performance is one of the major metrics of file systems, it is not the only one in real life and production. Other factors, such as maturity, reliability, compatibility, and usability, all matter when a file system is to be deployed. Thus, it usually takes years of testing and refinement for new file systems to become mature before being widely deployed in production environments. While before the process is complete, optimizing existing mature traditional file systems with respect to non-volatile memory features is one of the most reliable and practical approaches to take the performance benefits of non-volatile memory. Many mature traditional file systems, such as Ext4 and XFS, are already enhanced with the "direct access" mode to optimize their performance on non-volatile memory. However, in benchmarks and real-world applications, these optimized traditional file systems are still outperformed considerably by new kernel-space file systems designed for non-volatile memory. **Thus, it is a practical and important challenge to identify the culprits of the performance difference between traditional file systems and new file systems on non-volatile memory, and improve traditional file system performance on non-volatile memory with as minimal modifications as possible.**

With respect to the three challenges described above, this thesis conducts in-depth research on non-volatile memory file systems from three perspectives, including new kernel-space non-volatile memory file system designs, new user-space non-volatile memory file system designs, and the optimizations for traditional file systems on non-volatile memory. Specifically, the research content and contributions of this thesis are as follows:

First of all, for the issues of synchronous cache line flushes in the crit-

ical paths, this thesis proposes **a kernel-space asynchronous non-volatile memory file system, SoupFS**, which delays the persistence of file system operations to eliminate the costly synchronous cache line flushes while retaining the crash consistency. The SoupFS design combines both advantages of soft updates and non-volatile memory. It eliminates file system metadata persistence and synchronous cache line flushes in the critical paths by leveraging the delayed persistence of file system operations allowed by soft updates. With the byte-addressability of non-volatile memory, SoupFS also utilizes more efficient data structures to organize the file system content, which matches the pointer-based dependency tracking, simplifies the complicated dependency tracking and enforcement in the original soft updates implementation. To avoid the use of unnecessary page cache in the dual-views, SoupFS proposes pointer-based dual-views, which leverages different pointers in shared structures to present two different views, to support the delayed persistence of file system metadata and simplified dependency tracking and enforcement. Evaluation results show that compared with existing kernel-space non-volatile memory file systems, SoupFS significantly reduces the latency of file system operations and improves the operation throughput.

Second, to address the indirect metadata update issues in user-space non-volatile memory file systems, this thesis proposes **a new user-space non-volatile memory file system framework, Treasury**. Surveys on file permissions show that applications tend to store their files with similar permissions, and the permissions are seldom changed. According to this observation, this thesis proposes a new abstraction, Coffer, to manage a collection of non-volatile memory pages with the same access permission. With the Coffer abstraction, Treasury separates the non-volatile memory management from its protection. The kernel module, KernFS, provides strong protection of non-volatile memory at the Coffer granularity while enabling user-space

file systems, μFS, the full control over file system structures (including file data and metadata) within Coffers. With mechanisms such as memory protection keys, this thesis also addresses the potential stray writes and malicious attacks induced by the full control to Coffers of the user-space programs, which enhances the protection and isolation of file system structures. The Treasury framework supports unmodified dynamic-linked applications to run atop high-performance user-space non-volatile memory file systems and facilitates file systems development with several utilities. Within the Treasury framework, we implemented a user-space non-volatile memory file system, ZoFS, as an example μFS. Evaluation results show the ZoFS with the Treasury framework outperforms other file systems in both benchmarks and real-world applications.

Third, to eliminate the bottlenecks and improve the performance of mature traditional file systems on non-volatile memory, this thesis proposes **a further optimized traditional file system for non-volatile memory, Ext4pm**. Through evaluating and analyzing the performance of the traditional file system, Ext4, running on non-volatile memory and comparing it with the new non-volatile memory file system, NOVA, this thesis identifies several performance issues in Ext4 that limit the exploitation of non-volatile memory. These issues are further summarized into three culprits that cause the performance issues, including excessive abstractions, incommensurate implementations and non-scalable designs. This thesis proposes several optimization strategies for the culprits and optimizes Ext4 on non-volatile memory by implementing specific optimizations or applying 3rd-party patches. With less than 1,000 lines of modifications, our optimized Ext4pm outperforms highly-optimized Ext4, i.e., Ext4 with all 3rd-party patches, in microbenchmark FxMark, synthesized benchmark Filebench, and the real-world application RocksDB, by 7.7×, 1.12× and 22%, respectively. In the real-world application RocksDB, Ext4pm also outperforms NOVA by

up to 8%.

The three file systems proposed or optimized in this thesis have conducted a more in-depth study on the application of non-volatile memory in file systems from three perspectives, including new kernel-space file system designs, new user-space file system designs and mature file system optimizations. These three file systems take different trade-offs and focus on different application scenarios. The kernel-space non-volatile memory file system SoupFS is relatively balanced. It implements the file system operations through the mature system call interfaces and designs new file system structures that match the characteristics of non-volatile memory to maximize the performance advantages of non-volatile memory. For scenarios with extremely high performance requirements, using the user-space non-volatile memory file system framework Treasury can avoid the overhead of system calls, context switches, and virtual file systems. Thus, it is possible to exploit non-volatile memory performance to the extreme. Meanwhile, Treasury allows the customization of user-space file system designs to meet the application's actual needs and further improves the efficiency of non-volatile memory data storage. For scenarios that require high reliability and stability but want to utilize non-volatile memory as much as possible, Ext4pm is more appropriate since it is based on the mature traditional file system and optimized for non-volatile memory. Through a small amount of code modifications, without changing the storage layout of Ext4, the processing flow of Ext4 file operations has been optimized to improve its performance of non-volatile memory. Ext4pm is completely compatible with Ext4 and can be directly used by various existing tools designed to manage, tune and monitor Ext4. The design principle and optimization strategies used in this thesis can also be adopted by other new storage media and different scenarios, providing new insights and methods to design, implement, optimize, and evolve high-speed file systems in the era of big data.

KEY WORDS: Non-volatile Memory, File System, Storage System

目 录

插图索引

表格索引

第一章　绪论

存储是现代计算机系统中最重要的功能之一。随着计算机网络等系统的发展和大数据时代的到来，越来越多的数据在不断地产生并通过计算机系统被持久地保存在存储设备之中。同时，存储设备和存储介质也在不断发生变革。从最古老的打孔纸带到磁鼓，再到磁带、机械硬盘和高速固态硬盘，存储介质和设备在访问速度、存储密度、制作工艺、功耗和数据持久性等方面不断地发展。

非易失性内存（Non-volatile Memory，NVM）是一种新型存储介质。如图 1–1 所示，非易失性内存结合了传统大容量存储设备持久保存数据的能力（即非易失）和普通内存的高性能与字节粒度寻址，模糊了内存与存储（外存）之间的边界，为计算机存储系统带来了新的挑战与机会，在计算机存储体系中正掀起一场新的变革。

作为计算机存储体系中最常用的系统之一，文件系统负责管理不同的存储设备，并以统一的文件接口向应用程序提供存储功能。面对革命性的新型非易失性内存存储设备，文件系统需要根据非易失性内存的特性，进行有针对性的优化和设计，才能充分发挥非易失性内存的性能优势，为用户提供更加快速的存储服务。

在本章接下来的三部分中，将首先介绍几种比较常见的非易失性内存技术和存储设备，并对非易失性内存的特性和其为存储系统带来的挑战与机遇进行介绍。此后，将介绍基于非易失性内存的文件系统方面的研究工作，重点关注不同文件

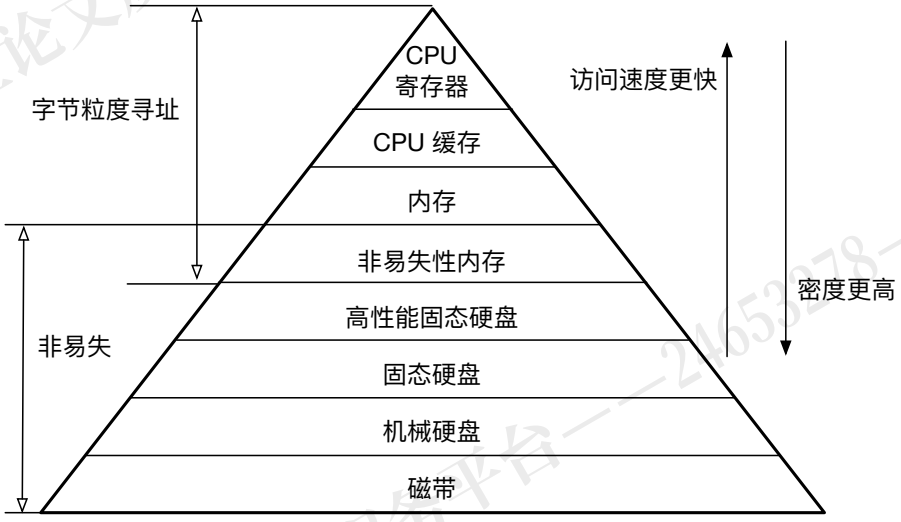


图 1–1　非易失性内存在计算机存储层次结构中的位置。非易失性内存结合了内存的字节粒度寻址与存储（外存）的非易失特性，模糊了传统内存与存储之间的界限

系统如何为非易失性内存进行优化和设计。最后，将简要介绍本文在基于非易失性内存文件系统研究中做出的工作和主要贡献，并介绍本文的组织结构以结束本章。

1.1 非易失性内存的发展与现状

本节将首先简要介绍比较常见的非易失性内存技术，并根据商用非易失性内存存储设备对非易失性内存的特点进行总结。最后将讨论非易失性内存为存储系统，特别是文件系统带来的挑战与机遇。

1.1.1 非易失性内存技术介绍

非易失性内存是一种新型存储设备。正如固态硬盘（Solid-state Disk，SSD）大多基于闪存（Flash）技术一样，非易失性内存存储设备也基于更底层的非易失性内存技术，目前有多种原理可以实现非易失性内存技术，如相变内存（Phase-change Memory，PCM）[1-2]、自旋转移矩磁阻式内存（Spin-transfer Torque MRAM，STT-MRAM）[3-5] 、忆阻器（Memristor）[6-7]、3D-XPoint 技术[8-10]。

相变内存（Phase-change Memory，PCM）[1-2] 是一种非易失性内存技术。其利用了一种或者多种硫族氧化物制造出特殊玻璃作为存储材料。这种特殊材料，通过快速加热和慢速加热的方式可在晶态和非晶态之间进行转换。而该材料处于晶态和非晶态情况下的导电率不同，因此其可以表示不同的比特数值，从而进行数据的存储。IBM、英特尔和三星等公司[11-13] 一直在从事相变内存的研究工作。

自旋转移矩磁阻式内存（Spin-transfer Torque MRAM，STT-MRAM）[3-5] 是另一种新兴的非易失性内存技术。其利用了自旋转移矩效应，通过改变三层结构磁隧道结（Magnetic Tunnel Junction，MTJ）中的自由层的磁矩方向，使得磁隧道结表现出不同的阻抗状态，从而区分 0 和 1，进行数据的保存。自旋转移矩磁阻式内存具有能耗低、高可擦写次数、高密度、高性能的优势，是下一代存储技术的有力竞争者。英特尔、三星、海力士、IBM、EverSpin 等公司[14-15] 在不断地对自旋转移矩磁阻式内存进行研究。

忆阻器（Memristor）[6-7] 是除了电容、电阻、电感之外的第四种基本非线性电路元件。忆阻器的阻值可以通过控制流经忆阻器的电流进行改变，因此其也可以用来制成非易失性内存。忆阻器体积小且易堆叠，可以用来支撑大容量的非易失性内存设备（通常称为 ReRAM）。此外，通过合理设计忆阻器使用方式，可以从结构上模拟逻辑与、或、非等门电路。这使得直接使用忆阻器进行计算成为可能。惠普、三星、Crossbar 等公司[16] 在不断进行忆阻器非易失性内存设备的研究。

2015 年 7 月，英特尔与镁光共同对外发布了新型 3D-XPoint 技术[8-10]，可用于制作非易失性内存存储设备。虽然英特尔和镁光并未公布 3D-XPoint 技术的细节，但其宣称 3D-XPoint 技术与传统闪存（Flash）技术相比，有一千倍以上的性能提升。同时，3D-XPoint 介质具有更高的密度，允许写入更多的次数，且具有更低的能耗。在 2017 年，英特尔公司发布了使用 3D-XPoint 技术的固态硬盘产品[17-19]，性能和可靠性相较基于闪存的固态硬盘产品有极高的提升。在 2019 年，英特尔公司发布了使用 3D-XPoint 技术的持久内存产品英特尔傲腾持久内存[20-21]。其使用 DDR-T 接口，直接插在内存插槽中使用，是真正意义上的商用非易失性内存产品。英特尔通过与华为[22]、阿里巴巴[23]、腾讯[24]、HPE[25]、Lenovo[26]、DELL EMC[27] 等公司进行紧密合作，推广傲腾持久内存在数据中心中的使用。如今，傲腾持久内存已经走入了数据中心和云服务平台之中，在 SAP HANA[28]、Google Cloud[29] 和 Microsoft Azure[30] 服务中进行了部署和使用。

1.1.2 非易失性内存的特点

简单来说，非易失性内存结合了普通内存与存储设备各自的特点：既拥有内存一样的访问性能和访问方式，又拥有存储设备的大容量和持久保存数据的能力。综合不同种类的非易失性内存技术和商用的非易失性内存存储设备，非易失性内存对比普通内存和传统存储设备，具有以下特点：

- **非易失性**。相对于普通内存来说，非易失性内存可以持久地保存数据。断开电源装置之后，数据可以保存在非易失性内存中。当再次通电之后，计算机依然可以从非易失性内存中读取之前保存的数据，并可继续进行修改。
- **可字节寻址**。与传统存储设备一般使用 512 字节或者 4 KB 的块粒度进行访问不同，非易失性内存直接接入到内存总线之上，可以以字节粒度进行寻址和访问。同时，访问传统存储设备一般需要内核的帮助，使用特定的 I/O 接口和指令才可以访问存储设备上的数据。而非易失性内存由于具有与内存相同的访问方式，其可以使用 `LOAD` 和 `STORE` 等 CPU 指令直接进行访问。这使得其可以在用户态被直接使用，无需内核的参与，因而使得在用户态实现高性能非易失性内存文件系统成为可能。
- **高性能（高吞吐、低时延）**。得益于内存结构的高效性和非易失性内存介质的高性能，非易失性内存的读写时延在几十到几百纳秒级别，访问速度远高于需经过 PCIe 等 I/O 协议的传统存储设备。另一方面，非易失性内存的访问时延已经非常接近普通内存，但依然有一定的差距。这种差距使得非易失性内存在短时间内无法彻底替换掉普通内存在计算机系统中的地

位。同时在带宽方面，非易失性内存的带宽与普通内存同样存在差距。不过随着新型非易失性内存技术的不断研究和应用，非易失性内存在这些方面依然还存在很大的进步空间。

- **高密度**。目前最常见的普通内存为单条 8 GB 或者 16 GB。也有一些能够达到 32 GB 的内存条。然而相对于传统设备 TB 级别的容量，普通内存的存储密度较低。非易失性内存单条的容量可以做到几百 GB 甚至更高。在英特尔发布的傲腾持久内存产品中，有单条 128 GB、256 GB 和 512 GB 三种规格。这使得非易失性内存不仅仅可以作为一个存储设备的替代品，还可以被认为是普通内存在容量上的拓展。在内存计算等应用场景中，大容量的非易失性内存将能够发挥出巨大作用。
- **低能耗**。普通内存一般需要不断地进行刷新和充能以保证其中的数据不会发生变化。然而频繁的刷新操作需要消耗比较多的电能。相比来说，由于非易失性内存其存储介质本身就可以保证数据的持久性，因此其不需要进行频繁的刷新操作，运行时的能耗要低于普通内存。
- **高耐磨度**。存储介质一般都有最大写入次数。当写入次数超过固定的次数时存储介质会发生损坏，届时将不能保证保存在其中的数据依然能够被正确地读取。基于闪存技术的存储设备是一个比较著名的例子，因而闪存存储一般会非常注意在使用过程中控制磨损均衡。非易失性内存在耐磨度方面，比闪存存储有更高的最大写入次数，从而在理论上有更长久的使用寿命。然而考虑到非易失性内存的性能要远远高于闪存存储，其实际使用寿命还应与实际使用情况相结合而确定。如果频繁地在相同位置写入数据，拥有更高耐磨度的非易失性内存也可能会在更短的时间内发生写穿现象。因而在英特尔傲腾持久内存的固件中，也被加入了磨损均衡功能，防止因频繁在相同位置写入而过早产生写穿问题。

在上述这些非易失性内存的特点中，非易失性、可字节寻址和高性能是将非易失性内存作为存储设备时，以及设计基于非易失性内存文件系统时最关注的三个特点。非易失性让非易失性内存有了存储设备最基本的功能，而可字节寻址和高性能让非易失性内存在使用方式上与传统存储设备有了本质上的不同。同时，虽然非易失性内存具有高耐磨度，且在固件中会进行磨损均衡考虑，但其依然有被写穿的风险，因此磨损均衡也是一些非易失性内存文件系统在设计时进行考虑的要点。

非易失性内存的另外一种使用场景是将其作为大容量的普通内存，用于保存不需要持久化的临时数据，从而可以进行大规模内存计算等任务。在这种使用场

景下，非易失性内存的可字节寻址、高性能和高密度的特点更加重要。前两个特点让非易失性内存可以作为普通内存使用，而高密度的特点，允许用户使用有限的内存插槽组建 TB 甚至十几 TB 级别内存的单机系统。同时低功耗的特点也可以降低机器的总体功耗，满足对高性能计算等场景中功耗的要求[31]。由于本文主要关注对基于非易失性内存的文件系统的研究，故对此使用场景不再进行展开。

1.1.3 非易失性内存带来的机遇与挑战

非易失性内存为计算机存储系统带来了巨大变化。如果说计算机存储领域中的上一次变革，即从机械硬盘到固态硬盘的变革，为一次量的变革，那么非易失性内存带来的新变革，则是一场质的变革。本节将从非易失性内存如何改变存储层次结构出发，进而讨论非易失性内存为文件系统设计带来的机遇和挑战。

1.1.3.1 非易失性内存改变存储层次结构

固态硬盘相对于机械硬盘来说，虽然消除了机械寻道时间并带来了更高的并发性和巨大的性能提升，但固态硬盘和机械硬盘依然都使用块设备接口，应用程序依然间接地对存储设备进行访问。因而固态硬盘为计算机存储系统带来的更多是速度（量）上的改变。

相比于传统的块接口存储设备，非易失性内存却提供了与普通内存一样的可字节寻址接口。从块设备接口到字节粒度内存接口，非易失性内存将存储与普通内存相结合，提高了存储性能的同时，还从根本上改变现有的计算机系统软件和应用程序对存储的使用方式。因而非易失性内存带来的变革是性能和存储访问方式（质）上的变化。

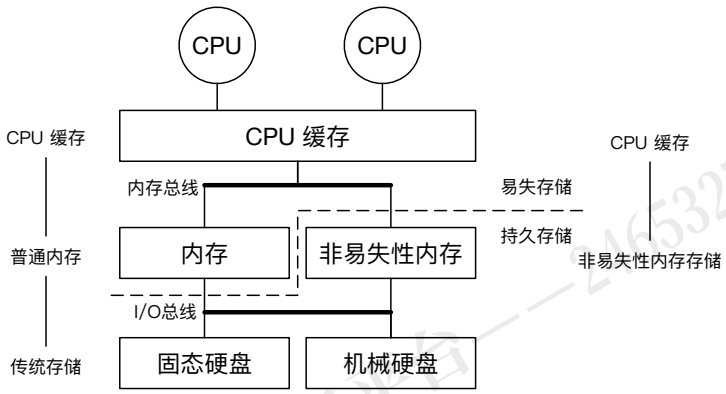


图 1–2 非易失性内存在计算机体系结构中的位置。非易失性内存改变了传统计算机体系结构中存储设备的位置，其接入到内存总线中，可以被 CPU 以字节粒度直接访问

图 1–2 中展示了包含非易失性内存的计算机存储层次结构。具体来说，在非易失性内存出现之前，系统软件和应用程序需要通过“CPU 缓存–内存–存储”的存储层次结构对存储设备进行访问（图 1–2 中左方的路径）。普通内存作为一个“中间层”在这个存储层次结构中发挥着非常重要的作用：首先，由于传统的存储设备需要通过块设备接口进行访问，而系统软件和应用程序一般以字节粒度修改数据和结构。因此，普通内存作为一个中间层，允许系统软件和应用程序在普通内存中细粒度地访问数据，并以较粗的块粒度与传统存储设备交换数据。其次，传统存储设备的访问性能与 CPU 缓存的访问性能相差多个数量级，如果每次数据访问都与存储设备进行交互，会极大地影响应用程序的性能。因此，系统软件和应用程序需要使用性能处于两者之间的普通内存作为数据缓存，以缓解两者访问性能差距带来的系统性性能问题。

在非易失性内存出现之后，其结合了外部存储与内存的优势，直接接入在内存总线之上，一改传统的“CPU 缓存–内存–存储”的存储层次，使得“CPU 缓存–存储（非易失性内存）”的新存储层次成为可能（图 1–2 中右方的路径）。非易失性内存可字节寻址的特性，使得对非易失性内存的访问不再依赖于以普通内存作为“字节粒度”与“块粒度”之间的中间层；同时，由于非易失性内存具有接近普通内存的性能，普通内存作为 CPU 与存储之间的缓存的重要程度也极大减小。“CPU 缓存–存储（非易失性内存）”的新存储层次，将普通内存从存储软件栈中去除，简化了传统的存储软件栈的同时，打破了原有存储系统中的“内存快、存储慢”的假设，为进一步提升文件系统的数据存储性能带来了新的机遇，同时也为文件系统的设计提出了许多新的需求，带来了诸多挑战。

1.1.3.2 非易失性内存为文件系统发展带来的机遇与挑战

文件系统是计算机系统中最常用的存储系统之一，负责管理不同的存储设备，并以统一的文件接口向应用程序提供存储功能。非易失性内存作为一种新型存储设备，为文件系统设计提供一个新的发展契机和方向，为进一步提升高性能文件系统带来了新的机遇。而机遇同样伴随着挑战。

高性能带来新的性能权衡，系统软件栈易成为瓶颈。 由于传统存储设备的访问速度较慢，文件系统通常会使用额外的操作确保对存储设备的访问性能是最优的。例如，为了避免机械硬盘漫长的寻道时间影响存储性能，文件系统以及相关系统软件会通过额外操作尽量保证对存储设备的访问是顺序的；甚至一些文件系统本身的结构就是为了顺序访问存储设备而设计（如日志结构文件系统[32]）。当存储

设备的访问速度是整个系统的性能瓶颈时，这些操作和设计是有效的，它们使用非瓶颈处的很小性能开销，换取了性能瓶颈处极大的性能提升。

非易失性内存将存储设备的吞吐量和时延提升到普通内存级别，原有的“存储设备访问速度慢”的假设不再成立，此前文件系统中使用的额外操作，不仅带来的性能提升有限，还会由于这些操作本身而成为系统中的新的瓶颈。不仅如此，由于高性能文件系统通常作为操作系统内核的一部分运行在内核中，应用程序需要通过系统调用的方式才能对文件系统发起请求。而一个请求需要经过系统调用层、虚拟文件系统层之后才能到达具体的文件系统。而文件系统在处理请求时，数据又需要经过页缓存、块设备层、I/O 调度、设备驱动，最终到达存储设备。考虑到非易失性内存的高性能，系统存储栈中这些过多的层次抽象增加了不必要的性能开销。

存储设备性能的改变，为文件系统设计带来了新的性能权衡，文件系统需要精简现有文件系统操作并重新审视此前的各项优化，甚至需要全新的设计，才能充分发挥出非易失性内存的性能。同时，由于非易失性内存的高性能，文件系统或操作系统内核中的软件栈非常容易成为限制存储性能的瓶颈，这对文件系统和相关存储软件栈的设计和优化提出了非常高的要求。

字节粒度寻址推动新的访问方式，更小的原子持久化粒度影响一致性保证。 非易失性内存的字节粒度寻址对文件系统来说是一把双刃剑。一方面，文件系统中有众多粒度较小的更新，如更新一个文件的访问时间，更新一个文件结构中的一个指针，更新分配信息中的某个比特位等。传统存储设备最小的更新单元（即存储块）都要大于这些修改，导致在传统存储设备中有写放大的问题。而非易失性内存的字节粒度更新，恰好可以满足这些小规模、细粒度的数据更新请求，可以提升文件系统元数据的更新性能。

然而另一方面，由于非易失性内存的更新粒度小，在修改较大量的数据时，需要跨越多个最小更新单元，而非易失性内存通常无法保证多个原子更新单元上数据写入的原子性。因此，更小的原子持久化粒度，使得文件系统原有的一致性保护方法不再有效。例如，在传统存储设备上写入一个 inode 结构时，由于 inode 结构大小小于一个存储块，此 inode 的写入（即持久化）是被原子执行的；然而，在可字节寻址的非易失性内存上，写入一个 inode 结构时，若 inode 结构超出了 CPU 原子指令的大小（通常为 8 字节，一些指令允许 32 或 64 字节的原子写入），则非易失性内存无法保证此写入被原子持久化。因此，文件系统不得不设计新的一致性保证方法保证原子更新，这反而让文件系统的一致性保证变得更加复杂。

非易失性内存的可字节寻址对文件系统来说既是一种机遇也是一种挑战。利

用好非易失性内存可以极大提升文件系统的性能，甚至由于可字节寻址允许应用程序在用户态直接访问非易失性内存，文件系统可以利用这一点为应用程序提供独特而高效的访问接口；然而同时，可字节寻址也使得文件系统在一些一致性保证操作上变得复杂，且由于易失性 CPU 缓存有可能将 CPU 的写入操作以乱序进行持久化，文件系统需要注意使用特殊的机制来保证写入的原子性以及不同写入操作的顺序，这对于文件系统的正确性和可靠性带来了比较大的挑战。

1.2 相关研究概述

自非易失性内存技术出现以来，基于非易失性内存上的文件系统研究层出不穷。按类别可以主要分为三类：(1) 新型内核态非易失性内存文件系统，如 BPFS[33]、PMFS[34]、HiNFS[35]、NOVA[36-37] 等；(2) 新型用户态非易失性内存文件系统，如 Aerie[38] 和 Strata[39]；(3) 传统文件系统在非易失性内存上的优化，如在 Ext4 和 XFS 中为支持非易失性内存提供的新的操作模式——直接访问（Direct Access，DAX）[40-41]。本节将对这些文件系统进行简要介绍。

1.2.1 内核态非易失性内存文件系统

高性能文件系统一般作为操作系统内核的一部分运行在内核态。因此，在非易失性内存出现之后，首先出现了一批内核态非易失性内存文件系统。这些文件系统结合非易失性内存的特点，将日志机制（Journaling）、写时复制技术（Copy-on-Write）、日志结构（Log Structuring）等传统的文件系统技术进行相应的优化后应用在非易失性内存之中。

1.2.1.1 可字节寻址持久内存文件系统 BPFS

BPFS[33] 是一个特意为可字节寻址的非易失性内存而设计的文件系统。其利用树形结构来管理文件系统中所有的数据和元数据，并使用写时复制技术（Copy-on-Write，CoW）保证文件系统的崩溃一致性。在写时复制技术中，更新并不直接修改在原有数据上，而是先将原有数据拷贝到新的内存地址中，随后在新的拷贝上进行数据修改。最后，通过原子地修改指针，将旧的数据替换为新的数据。BPFS 的作者指出，传统使用写时复制技术的文件系统有递归更新（Wondering Tree）问题：即使文件系统仅需要更改树形结构中的一处数据，其依然需要递归地使用写时复制技术修改该节点所有的祖先节点，直至树根。为了优化这个问题，BPFS 提出支持短路的影子页机制。支持短路的影子页机制在原有写实复制技术基础上增加了短路机制：当一个修改可以通过一次原子操作完成时，则可以在原地进行修

改，而无需进行拷贝，同时也不需要继续向父节点递归进行修改。结合非易失性内存可字节寻址的特性，支持短路的影子页机制可以将不超过 8 个字节的修改进行原子更新。BPFS 的整个文件系统被组织成一颗树，因此文件系统中的任何修改（包括空间分配和文件重命名等）都可以通过支持短路的影子页机制保证原子更新。除了支持短路的影子页机制之外，BPFS 还提出了一系列其他优化。如当对文件进行追加操作时，可以先将文件数据直接写入到文件末尾。此时无需考虑这些数据写入的一致性。在保证所有数据均已经被持久化完毕后，通过原子更新文件大小，来使得这些新写入的数据一次性生效，从而保证了这个数据追加操作的原子性。又如对于不超过原子更新粒度的修改，可以直接原地完成，而无需进行数据拷贝。BPFS 使用用户态文件系统框架 FUSE 进行实现，因而在性能方面受到 FUSE 框架的影响较大。准确来说，BPFS 是一个基于 FUSE 的用户态文件系统，但由于在使用过程中，所有文件系统请求依然需要通过系统调用进入到内核的 FUSE 模块中，因此将其列在此处。

1.2.1.2 持久内存文件系统 PMFS

PMFS[34] 是另外一个为非易失性内存而设计的文件系统。PMFS 的作者首次提出了新的硬件原语——`pm_wbarrier`。通过回顾当时 CPU 和内存系统的架构，作者指出新原语在保证数据持久化和一致性方面的重要性：在非易失性内存中，当数据从 CPU 缓存中被刷出后的一段时间内，数据还未被写入到非易失性内存中。而如果此时发生断电，这些数据会丢失。现有的 `CLFLUSH` 缓存刷除指令，只能保证数据从 CPU 缓存中被刷出，却并不能保证数据已经被持久化在非易失性内存之中。此时就需要在 `CLFLUSH` 之后使用 `pm_wbarrier` 这个新的原语来进行数据持久化的保证。在新的原语的基础上，PMFS 利用了细粒度的日志（Fine-grained Journaling）机制，结合了非易失性内存上的原子更新操作高效地保证元数据的一致性。同时 PMFS 还支持使用写时复制技术保证数据更新的原子性和一致性。PMFS 进一步提出允许应用程序在用户态通过文件内存映射（Memory Map）机制，直接访问非易失性内存。PMFS 同时还提供了透明大页（Transparent Huge Page）机制，使非易失性内存上的文件内存映射访问变得更加高效。最后，PMFS 指出，由于非易失性内存的访问可以通过简单的 `LOAD` 和 `STORE` 等 CPU 指令进行，一个有软件缺陷（Bug）的程序很容易对文件系统中的数据造成损坏。这个问题被称作误写（Stray Write）问题。为了保护 PMFS 文件系统中的数据不受到误写问题的影响。PMFS 的作者组合使用了多种软硬件机制，包括内存的分页机制，CPU 的权限级机制，CPU 平台的 SMAP 机制和一个软件实现的写窗口机制。对于

写窗口机制，非易失性内存被以只读的方式映射到内核的地址空间中。当 PMFS 想要访问文件系统的数据时，其会首先将 CR0 寄存器中的 WP 比特位关闭掉。此时，当前的 CPU 可以直接修改被只读映射的非易失性内存区域而不触发任何的异常。当 PMFS 完成了对非易失性内存区域的修改后，其会重新打开 CR0 寄存器中的 WP 比特位，以保证有缺陷的代码对只读映射的非易失性内存区域的修改会访问失败并触发异常。PMFS 被实现为 Linux 的一个内核模块，运行在内核空间中。相较于 BPFS 来说，PMFS 不受到 FUSE 框架的性能影响，拥有更好的性能。

1.2.1.3 使用缓存的高性能文件系统 HiNFS

HiNFS[35] 是一个利用普通内存来隐藏非易失性内存较长写时延的非易失性内存文件系统。为非易失性内存设计的文件系统一般会避免使用页缓存。由于非易失性内存的高性能，直接在非易失性内存中进行数据的更新，与在普通内存中进行更新的性能相差较小。而若继续使用页缓存机制，每个文件系统的数据写入都要首先写入到页缓存（即普通内存）中，再被拷贝到非易失性内存中保证持久性。页缓存带来的两次写入（一次在普通内存上，另一次在非易失性内存上）会影响文件系统的性能。这也是大多数非易失性内存避免使用页缓存的原因。而 HiNFS 指出，页缓存在一些情况下，依然对非易失性内存文件系统的性能有所帮助。如当一个文件的某部分被频繁地更新时，若这些更新并不要求立即持久化，则可以将这些更新暂存在页缓存中。此后在同一页上的后续更新可以在页缓存中被合并，最终文件系统通过一次非易失性内存写入来保证持久化。页缓存在此过程中充当了一个写缓冲区的作用，其可以合并在同一个页面上的多次写入，从而减少耗时的非易失性内存写入操作。因此，HiNFS 将页缓存机制带回到非易失性内存文件系统中，并进行了诸多优化。HiNFS 根据非易失性内存的字节寻址特性，设计了细粒度的缓存行写缓冲区。这个缓冲区以缓存行为粒度，能够合并相同缓存行上的多个写入操作，最终一次性写入到非易失性内存中。HiNFS 将不需要立刻持久化的写入称为惰性持久化写（Lazy-persistence Write），而将那些需要立刻持久化的写入称为急性持久化写（Eager-persistence Write）。其中惰性持久化写操作可以通过细粒度缓存行写缓冲区进行优化；而对于急性持久化写操作，HiNFS 让它们跳过写缓冲区，直接写入到非易失性内存中，从而避免两次写入问题造成的性能开销。HiNFS 使用了一个计算模型区分惰性持久化写和急性持久化写，进而采用不同的方法进行处理。为了保证能够快速地定位和读取到最新的数据，HiNFS 还在普通内存中维护了块索引（DRAM Block Index）和缓存行位图（Cacheline Bitmap）。HiNFS 的实现基于 PMFS，但由于其避免了惰性持久化写操作中大量的非易失性

内存写入操作，因此相对于 PMFS 表现出了明显的性能提升。

1.2.1.4 日志结构文件系统 NOVA

NOVA[36] 同样利用了普通内存和非易失性内存各自的优势，但其采取了日志结构[32] 来设计和实现非易失性内存文件系统。一般来说，机械硬盘等块设备的顺序访问性能远高于随机访问性能。因此高性能的文件系统通常顺序地访问存储设备。日志结构文件系统的设计是为了达到一种理想情况：文件系统中的所有读取操作均可以在内存缓存中完成，而所有的文件系统更新以日志形式顺序地追加到存储设备之上。NOVA 针对非易失性内存的特点，重新设计了日志结构文件系统。NOVA 将多个非易失性内存页，通过链表的方式串联起来，形成一个逻辑上的连续日志结构。NOVA 在每个非易失性内存页内部以追加的方式顺序写入文件修改，因此其依然可以享受到顺序写入带来的性能优势。同时，由于 NOVA 使用链表结构组织日志空间，其可以通过简单的链表插入和删除等操作灵活地对日志空间进行扩大、清理和回收。解决了传统日志结构文件系统中复杂的空间管理问题。NOVA 为每个文件建立了一个上述日志结构，用于保存该文件中的修改。如在常规文件的日志中，NOVA 会记录该常规文件中数据块的位置变化；在目录文件的日志中记录该目录中目录项的增加和删除。因此，在 NOVA 中进行文件操作的持久化，只需要在该文件对应的日志中追加内容即可。日志结构虽然利于顺序写入，但对读取的支持较差。为了加快文件系统读取的性能，NOVA 使用了普通内存中的高性能索引结构。例如，为了快速定位和访问文件中的数据，NOVA 通过基数树保存了一个常规文件中每个块号对应的非易失性内存页位置；又如，NOVA 为了加快目录内容的访问，使用红黑树索引了目录中所有的目录项。通过文件名的哈希值，NOVA 可以快速在红黑树中找到其对应的目录项在非易失性内存日志中的位置。多核可伸缩性是 NOVA 设计的重点之一，NOVA 中每个文件使用一个单独日志空间的设计，允许其在不同文件上的更新操作并发进行。同时，为了提升空间分配的可伸缩性，NOVA 将所有空闲的非易失性内存空间划分给不同的 CPU 核心。在进行空间分配时，执行 NOVA 逻辑的线程只需要从其所在 CPU 核心的私有空闲空间中分配非易失性内存即可，避免了多 CPU 核心之间的数据竞争。在数据持久化方面，NOVA 将所有的更改保存以追加日志的形式保存在非易失性内存中。无论是否发生崩溃，在进行文件系统的挂载操作时，NOVA 会读取非易失性内存中保存的日志，并根据其中记录的内容重建出普通内存中的所有索引结构。

1.2.2 用户态非易失性内存文件系统

虽然在内核中实现非易失性内存文件系统是一个比较传统的做法，但由于系统调用造成的上下文切换和内核中虚拟文件系统中复杂的逻辑，内核系统软件栈开销逐渐成为阻碍非易失性内存性能优势的瓶颈。另一方面，非易失性内存可字节寻址的特性，使其可以在用户态进行访问，因此，用户态非易失性内存文件系统逐渐成为研究热点。非易失性内存设备由内核管理，因此严格来说，用户态文件系统需要与内核中的特殊模块或内核中现有的文件系统配合，由内核中的模块将非易失性内存暴露给用户态文件系统进行使用。但由于这类文件系统直接在用户态处理应用程序的文件系统请求，因此本文中将其统称为用户态非易失性内存文件系统。

1.2.2.1 灵活的非易失性内存文件框架 Aerie

Aerie[38] 是一个用户态非易失性内存文件系统框架。其目的是提高文件系统的在应用使用中的灵活性。Aerie 利用非易失性内存高性能、可字节寻址等特性，对传统的存储栈进行了精简，如去掉了对于非易失性内存来说不必要的块设备层、I/O 调度器和设备驱动等。Aerie 认为，在高性能的非易失性内存上，继续使用现有的文件系统接口保存和访问数据可能并非是一种高效的做法。文件系统所提供的文件抽象和接口，限制了非易失性内存性能和特性的发挥，对于许多新型非易失性内存应用程序来说效率较低。为此，论文作者提出了 Aerie 文件系统架构，将非易失性内存以更加灵活的方式暴露给用户态的应用程序。Aerie 由三部分组成：一个非易失性内存管理器（NVM Manager）、一个可信的文件系统服务（Trusted File System，TFS）和文件系统库（File System Libraries）。非易失性内存管理器工作在内核态，负责进行非易失性内存空间的分配和保护。其以较大的内存块为粒度进行非易失性内存管理和分配，并通过内存管理单元的分页机制对非易失性内存进行保护。可信文件系统服务是一个运行在用户态的进程，负责处理文件系统中所有元数据以及受保护的文件数据的更改请求。可信文件系统服务通过向内核中的非易失性内存管理器发起请求，进行非易失性内存的空间分配。Aerie 的最后一部分是文件系统库。文件系统库与应用程序的代码链接在一起，为应用程序提供灵活的文件系统服务。这些文件系统库在将非易失性内存映射到内存空间后，可以直接读取和写入文件数据。而对于元数据修改，例如创建文件或者修改文件的权限，文件系统库需要通过进程间通信（Inter-Process Communication，IPC）的方法，发送请求给可信文件系统服务。可信文件系统服务在检查权限之后会对元数据修改请求进行处理并完成操作。应用程序可以根据其需求，选择或设计使用不

同的文件系统，以最符合应用程序需求的方式在非易失性内存中保存数据。换句话说，应用程序可以在 Aerie 框架中定制自己的文件系统库，这为应用程序对存储资源的使用带来极大的灵活性。例如，在 Aerie 的架构中，一个键值存储应用程序可以直接使用具有键值对接口的文件系统库；传统应用程序也可使用一个提供了传统文件系统接口的文件系统库。Aerie 文件系统的设计与微内核架构[42-47] 相似，其通过可信的用户态文件系统服务对文件系统进行管理。Aerie 同样也具有微内核架构的缺点——进程间通讯的性能问题。当应用程序频繁访问文件系统的元数据时，其会产生大量的进程间通讯请求，带来较大的性能开销。

1.2.2.2 基于非易失性内存混合存储文件系统 Strata

Strata[39] 是一个用户态的混合存储文件系统，其利用了非易失性内存可字节寻址的特性，实现了高效的用户态文件系统访问。Strata 的设计，分为用户态文件系统和内核态文件系统两部分。用户态文件系统同样以文件系统库的形式，与应用程序动态链接在一起。Strata 利用了非易失性内存可以在用户态被直接以字节粒度访问的特性，为每个用户态进程提供了一个可以在用户态访问的非易失性内存日志空间。应用程序可以将对文件系统的修改，以日志的形式保存在其非易失性内存日志空间中。在保证写入此日志空间中的日志已经被成功持久化之后，此操作便可以认为完成。内核态文件系统会定期对用户态文件系统写入到日志空间中的日志进行消化（Digest），即将日志中的修改应用到全局可见的非易失性内存区域中。同时内核态文件系统会根据存储数据的冷热程度和存储容量的情况，将数据在非易失性内存与机械硬盘和固态硬盘等传统存储设备之间进行迁移，以充分利用不同存储设备等优势。为了读取到传统存储设备上的数据，Strata 的用户态文件系统库通过使用 SPDK[48] 在用户态对传统存储设备上的数据进行访问。对于访问一致性问题，由于进程拥有自己的非易失性内存日志空间，写入该空间的数据对其他进程不可见。因此，当需要进行多进程同步时，内核态文件系统需要进行消化操作，以保证每个进行对文件系统的修改可以被其他进程观察到，兼容现有文件系统的语义。

1.2.3 传统文件系统对非易失性内存的支持与优化

前文中介绍的文件系统均为针对非易失性内存所设计的新型文件系统。然而文件系统从设计到实现再到大规模使用，并非是一朝一夕的事情。新的非易失性内存文件系统需要经过较长时间的验证才能可靠地投入到实际的生产中。在这种情况下，为传统的文件系统提供非易失性内存支持，是另外一个很重要的研究方

向。在传统的文件系统，如 Ext2、Ext4 和 XFS 中，均增加了对非易失性内存的支持。这些文件系统中为非易失性内存增加了一种新的模式和挂载选项，称为直接访问（Direct Access，DAX）。当这些文件系统以直接访问模式进行挂载，或以直接访问模式打开一个文件时，对文件的修改会直接进入到非易失性内存中，跳过普通内存中的页缓存机制，从而避免两次写入的问题。另外一方面，这些传统的文件系统还提供了直接访问模式的文件内存映射。通过 `mmap` 的文件内存映射接口，应用程序可以直接将物理非易失性内存映射到自己的虚拟内存空间中。此后，应用程序可以直接使用 `LOAD` 和 `STORE` 等 CPU 指令在非易失性内存上进行操作。

1.3 本文主要贡献

1.3.1 现有研究工作的不足

虽然研究人员此前对新型内核态非易失性内存文件系统、新型用户态非易失性内存文件系统和传统文件系统在非易失性内存上的优化均有所研究，但在这三个不同场景中，现有研究工作均有所不足。

- 对于现有的新型内核态非易失性内存文件系统来说，为了保证文件系统崩溃一致性，防止 CPU 对非易失性内存的写入被 CPU 缓存乱序写回到非易失性内存之中，这些新型文件系统需要在处理系统调用时使用同步的 `CLFLUSH` 等缓存行刷除指令，导致系统调用关键路径上的时延增长，降低了在高速非易失性内存上的文件系统性能。**现有研究工作对如何在保证文件系统崩溃一致性的情况下，避免同步缓存刷除指令对文件系统性能的影响未进行深入研究**。
- 对于新型用户态非易失性内存文件系统，由于其与应用程序处于同一个内存空间之中，为了防止应用程序有意或无意地修改文件系统的关键结构，用户态非易失性内存文件系统一般无法直接修改文件系统元数据，以保护文件系统的安全性、完整性和一致性。这种限制导致用户态文件系统的元数据依然需要进入内核空间、或通过通讯由可信的进程进行，影响了用户态文件系统的性能。**现有研究工作未曾涉及将对元数据的控制完全赋予用户态非易失性内存文件系统，并同时保障文件系统的安全性和完整性**。
- 虽然 Ext4、XFS 等成熟的传统文件系统已经针对非易失性内存的特性增加了“直接访问”（Direct Access，DAX）模式，然而在实际应用中，其性能依然与为非易失性内存设计的新型内核态文件系统有所差距。**现有的研究工作对导致性能差距的原因未进行深入分析，同时对如何在尽量少改动现有成熟文件系统的情况下解决性能问题，提升成熟的传统文件系统在非易失**

性内存上的性能，未曾有深入研究和探索。

1.3.2 本文的主要贡献

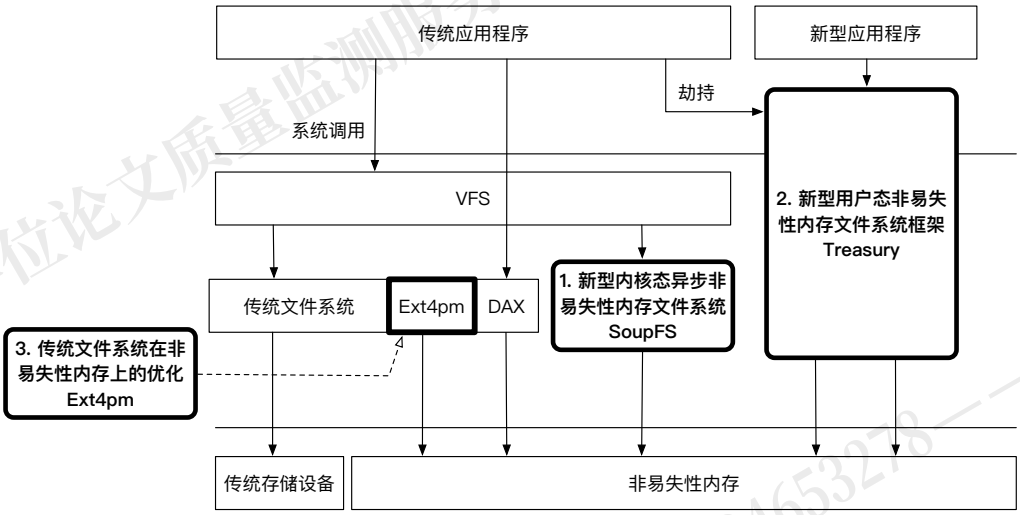


图 1–3 本文的主要贡献总览，包括新型内核态异步非易失性内存文件系统 SoupFS、新型用户态非易失性内存文件系统框架 Treasury 和传统文件系统在非易失性内存上的优化 Ext4pm

针对现有工作的三个不足之处，本文分别从三个角度对基于非易失性内存的文件系统展开较为深入的研究（如图 1–3），包括新型内核态非易失性内存文件系统研究、新型用户态非易失性内存文件系统框架研究和传统文件系统在非易失性内存上的优化研究。具体来说，本文的研究内容和贡献如下：

第一，针对关键路径上的同步缓存行刷除问题，本文提出**新型内核态异步非易失性内存文件系统 SoupFS**，通过推迟文件系统操作的持久性保证，在保证崩溃一致性的情况下，消除了关键路径上的同步缓存行刷除指令。SoupFS 结合了软更新技术与非易失性内存的特点，通过软更新技术允许文件系统修改的延迟持久化，在关键路径中消除了元数据的持久化和同步缓存刷除操作。SoupFS 还利用非易失性内存可字节寻址的特性，设计使用了更高效的文件系统组织结构，与软更新技术中的指针粒度的依赖性追踪相契合，简化了传统软更新技术中复杂的依赖追踪和依赖保证。SoupFS 还提出了基于指针的双视图，在同一结构上通过不同指针形成两种不同的视图，在不依赖于页缓存的情况下，支持元数据的延迟持久化和依赖追踪与保证。在性能测试实验中，与现有的内核态非易失性内存文件系统相比，SoupFS 显著降低了文件系统操作的时延，提高了文件系统操作的吞吐量。

第二，针对用户态文件系统无法直接管理文件系统元数据问题，本文提出**新型用户态非易失性内存文件系统架构 Treasury**。通过对应用程序的数据文件进行

调研，本文发现应用程序倾向于以相似的文件权限保存其文件，同时这些文件的权限很少被修改。基于此发现，本文提出一个新的抽象 Coffer，用于表示一组具有相同访问权限的非易失性内存资源。借助 Coffer 抽象，本文设计了 Treasury 框架，将非易失性内存的保护和管理分离。Treasury 在内核中的模块 KernFS 以 Coffer 为粒度为非易失性内存提供强保护。同时，Treasury 赋予了用户态非易失性内存文件系统 μFS 对 Coffer 内的文件系统数据和元数据的完全管理权。通过结合内存保护键等机制，本文还解决因用户态对 Coffer 的完全管理权而造成的潜在误写和恶意攻击问题，进一步增强对文件系统结构的保护和隔离。Treasury 框架支持动态链接程序不经修改直接运行在高性能的用户态非易失性内存文件系统之上，并为设计和实现不同的新型用户态非易失性内存文件系统提供了便利。在 Treasury 框架之上，本文设计并实现了一个用户态非易失性内存文件系统 ZoFS。其在基准测试和现实应用程序测试中，比现有非易失性内存文件系统表现出更优的性能。在部分测试中，ZoFS 对文件的访问性能能够达到非易失性内存访问带宽的上限。

第三，针对成熟文件系统在非易失性内存上的性能瓶颈问题，本文提出**为非易失性内存优化的传统文件系统 Ext4pm**。通过对成熟的传统文件系统 Ext4 在非易失性内存上进行测试和分析，并与新型非易失性内存文件系统 NOVA 进行对比，本文发现了 Ext4 在使用非易失性内存时的 10 个性能问题，并将这些问题归纳为三个根本原因：过度抽象、非相称的实现、不可伸缩的设计。针对这三个根本原因，本文提出了多种优化策略，并根据策略应用补丁或实现优化，解决 Ext4 在非易失性内存上的性能问题。在进行了不足 1,000 行的优化修改后，优化后的文件系统 Ext4pm 在基准测试中的性能提高最多 7.7 倍，性能接近甚至优于 NOVA 等新型非易失性内存文件系统。在真实应用 RocksDB 的性能测试中，Ext4pm 的性能最多超过 NOVA 达 8%。

本文所包含的三个研究内容分别从新型内核态文件系统、新型用户态文件系统和成熟的传统文件系统的优化三个角度对非易失性内存在文件系统中的应用进行了较为深入的研究。提出的三个文件系统适用于不同的使用场景。内核态非易失性内存文件系统 SoupFS 较为均衡。其既通过成熟的系统调用接口实现了完整的文件系统操作，又针对非易失性内存的特点设计了全新的结构，能较好地发挥非易失性内存的性能。用户态非易失性内存文件系统框架 Treasury 针对于对性能要求极高的场景，其设计避免了系统调用、上下文切换和虚拟文件系统的开销，能够最大程度地发挥非易失性内存的性能优势。同时，应用程序可以根据自身需求，在 Treasury 框架中定制化文件系统。应用程序通过对文件系统结构进行有针对性的设计，以提升其使用非易失性内存的效率。基于成熟传统文件系统、为非易失

性内存优化的 Ext4pm 适用于对可靠性、稳定性要求较高的场景。其应用了极少量代码修改，在不改变 Ext4 的存储布局的情况下，对 Ext4 的文件操作的处理流程进行了优化，实现了 Ext4 在非易失性内存上的性能提升。Ext4pm 完全兼容 Ext4 的存储格式，可直接使用为 Ext4 设计的各种工具进行管理、调优和监控。

本文在设计和优化基于非易失性内存的文件系统时所使用的思想和技巧，可以拓展到其他新型存储介质和其他场景中，为新型存储介质的使用、新型文件系统的设计实现和演进提供了新的方法和思路。

1.4 本文组织结构

本文内容共分为五章。第一章绪论部分至此已基本结束，后续章节组织如下：

第二章将介绍内核态异步非易失性内存文件系统 SoupFS 如何将软更新技术与非易失性内存技术相结合，消除关键路径上的同步缓存行刷除操作，缩短文件系统操作时延，提高内核态非易失性内存文件系统性能。

第三章将介绍新型用户态非易失性内存文件系统框架 Treasury 如何通过新的抽象赋予用户态非易失性内存文件系统对文件系统数据和元数据的完全管理权，以发挥非易失性内存的极致性能，并介绍 Treasury 如何对文件系统的安全性和隔离性进行保证。

第四章将对成熟的传统文件系统在非易失性内存上的性能进行详细分析，揭示导致性能问题的根本原因，并提出相应优化策略，让传统文件系统的性能接近甚至优于新型非易失性内存文件系统。

第五章将对全文进行总结，并对非易失性内存文件系统上的未来研究工作进行展望。

第二章　新型内核态异步非易失性内存文件系统设计

在 Linux、Windows 等常见的操作系统中，文件系统通常是操作系统内核的一部分，或以可加载的内核模块的形式出现和被使用。由于这些文件系统在内核态处理文件请求，本文称这些文件系统为内核态文件系统，如 Linux 中的 Ext4、XFS、F2FS 等文件系统。在内核中实现文件系统，具有诸多好处。首先，在内核中实现文件系统，便于对存储设备进行直接操控：无论是机械硬盘还是固态硬盘，存储设备通常需要使用特权指令进行操作，而特权指令需要在内核态（通常为操作系统内核）中才能使用。其次，在内核中实现文件系统，可以保护文件系统的安全性。应用程序通常运行在用户态空间，而内核文件系统运行在内核态空间，内存空间的隔离使得文件系统的数据和元数据安全性得到充分保障。再次，内核文件系统不需要为应用程序进行特殊的适配，即可兼容旧有的应用程序。使用不同语言编写的应用程序可能会使用不同的上层接口访问文件系统，如 C 程序可能会使用 GNU C Library[49] 中提供的 `open`、`read`、`write`、`close` 等函数访问文件，Java 程序则使用 `java.io`[50]、`java.nio.file`[51] 等包中的对象对文件进行访问。但最终，这些访问方式均会通过成熟稳定的系统调用接口向内核发起文件请求。因此在内核中的文件系统只需要考虑实现系统调用所需的功能，而不需要对应用程序进行特殊的适配。同时，内核中的文件系统框架，如 Linux 中的虚拟文件系统（Virtual File System，VFS），使用成熟的代码实现了文件路径解析等通用流程，且为具体内核文件系统提供了大量的辅助函数与机制，减轻了内核文件系统设计和实现的负担。

与机械硬盘、固态硬盘等存储设备相同，非易失性内存作为一种硬件存储资源，首先由操作系统内核进行管理。因此在内核中实现管理非易失性内存的文件系统是一个比较直接的想法。本章将介绍一种新的内核态异步非易失性内存文件系统 SoupFS 的设计和实现。如图 2–1，SoupFS 是一个典型的内核文件系统，其运行在内核态空间之中，处理来自虚拟文件系统 VFS 的请求，管理非易失性内存并以文件的形式保存用户数据，同时，SoupFS 通过更新页表信息，利用内存管理模块（Memory Management Unit，MMU）硬件机制，允许应用程序通过文件内存映射的方式直接访问保存在非易失性内存上的文件数据，实现了类似于传统文件系统中直接访问（Direct Access，DAX）的特性。

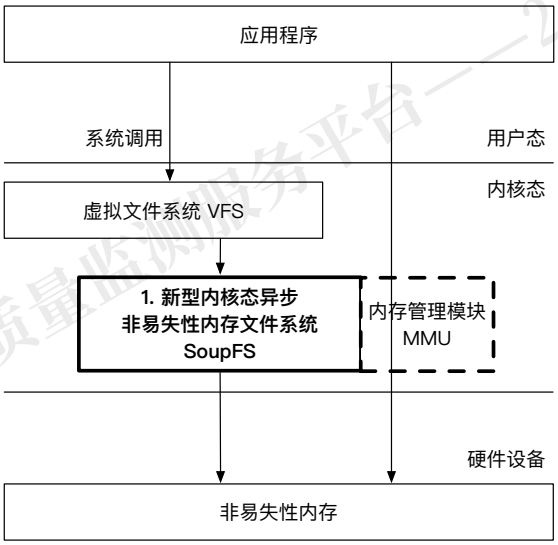


图 2–1 新型内核态异步非易失性内存文件系统 SoupFS 在应用和内核架构中的位置

2.1 研究背景与动机

2.1.1 CPU 缓存与缓存刷除

由于非易失性内存的可字节寻址特性，应用程序可以直接使用 LOAD 和 STORE 等指令访问非易失性内存。然而在 CPU 与非易失性内存之间，还有一层 CPU 缓存。在最经常使用的写回（Write-back）模式下，CPU 对非易失性内存的数据修改，首先写入到 CPU 缓存中，由 CPU 缓存的替换机制决定何时将数据修改写入到非易失性内存之中。这导致 CPU 发出的多个数据修改，会以不同的顺序（即乱序）写入非易失性内存中被持久化，从而有可能违背应用程序原有的语义，破坏元数据的一致性。

图 2–2 中给出了一个 CPU 缓存可能会导致乱序持久化的例子。在此例子中，data 和 flag 是保存在内存中的两个不同变量。其中 data 表示要保存的数据，flag 则表示在 data 中保存的数据是否有效。两个变量所在的内存地址相差较远，在 CPU 缓存中占用不同的缓存行。两个变量的初始值均为 0，因此此时 data 中的数据是无效的。

CPU 首先将 data 赋值为 2021，之后将 flag 置为 1，表示写入的 2021 是有效的数据。CPU 缓存使用写回（Write-back）模式，CPU 对内存的写入首先进入了 CPU 缓存。而 CPU 何时将缓存中的数据写回到非易失性内存中，一般是由 CPU 的替换策略来决定的。由于 data 和 flag 保存在不同的缓存行中，CPU 缓存在将这两个变量的修改写回到非易失性内存中时，有可能产生两种写回顺序：一种是 data 的修改先被写回，另给一种是 flag 的修改先被写回。由于数据在写入非易

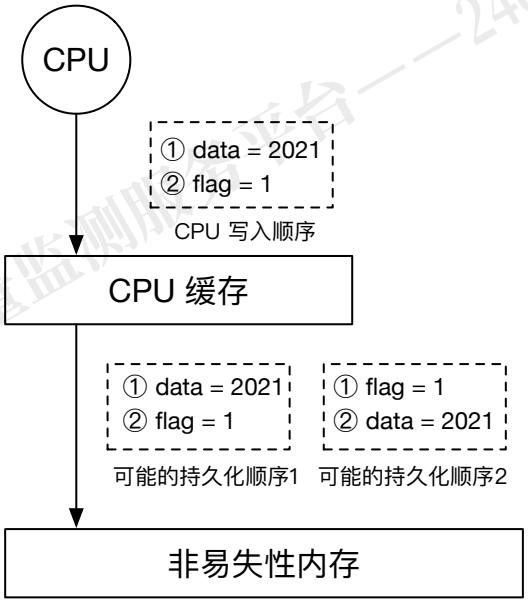
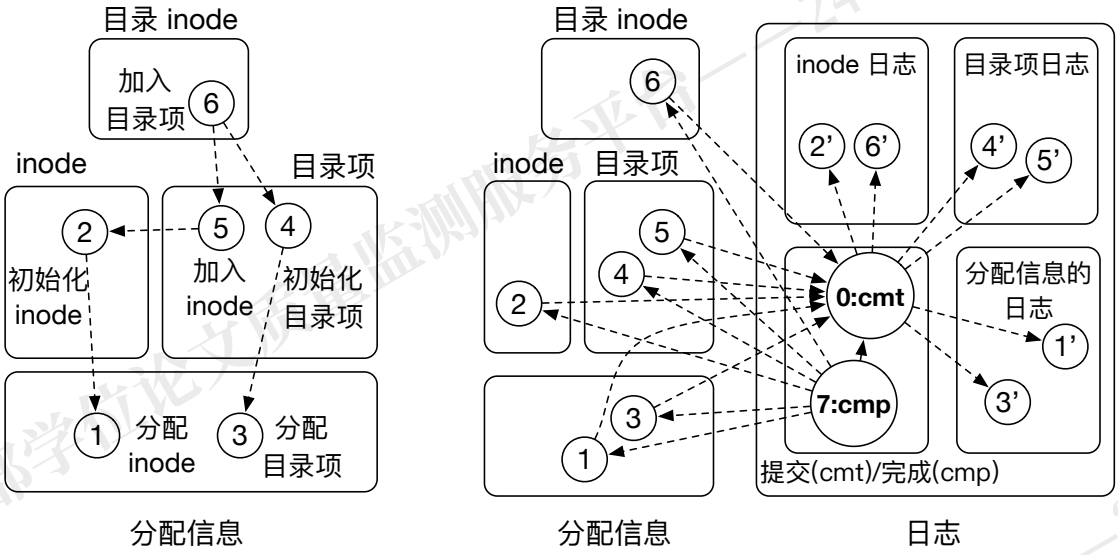


图 2–2 CPU 缓存导致持久化乱序的示例。CPU 缓存可能会将 CPU 的写入以不同的顺序写回非易失性内存（即乱序持久化）

失性内存之后才被持久化，因此写回顺序也表示了修改的持久化顺序。因此，如果写回顺序（即持久化顺序）是 flag 先于 data，且在 flag 刚刚被写入到非易失性内存之后，系统发生了断电，则在 CPU 缓存中还未来得及写回的 data 数据会丢失。若在通电之后读取非易失性内存中的 data 和 flag，会发现 data 的内容为 0，而 flag 的内容为 1，表示 data 中保存的 0 是有效数据，这与应用程序最初的语义是相违背的。因此，此例可以说明 CPU 缓存可能会导致数据的乱序持久化，从而破坏应用程序原因的语义。

为了解决这个问题，在使用非易失性内存时，应用程序需要主动使用 CLFLUSH 等指令来保证数据从 CPU 缓存中逐出，再进行后续的操作。具体来说，应用程序在将 data 赋值为 2021 之后，先使用 CLFLUSH 指令强制将 CPU 缓存中 data 的修改写回非易失性内存。在 CLFLUSH 指令完成之后，再将 flag 变量修改为 1。这样无论何时发生断电，从非易失性内存中读取数据时，要么 flag 为 0，表示保存的数据无效；要么 flag 为 1 且 data 为 2021，符合应用程序原有的语义。

使用 CLFLUSH 等缓存行刷除指令，可以保证数据写入非易失性内存的顺序。然而由于其需要等待数据写回非易失性内存才能返回，使用 CLFLUSH 等指令会延迟更新操作的时延，影响应用程序的性能。



(a) 创建一个文件所需的操作步骤和其持久化顺序

(b) 在使用日志机制创建一个文件时所需的操作步骤和持久化顺序

图 2–3 创建一个文件时的各个步骤以及步骤间的持久化顺序要求

2.1.2 文件系统中的写入操作顺序

为了避免 CPU 缓存导致的乱序持久化，现有的非易失性内存文件系统在处理文件系统请求时，一般会在关键路径中使用缓存行刷除指令保证请求中不同步骤之间的持久化顺序，以防止文件系统一致性被破坏。图 2–3 中给出了在文件系统中创建一个空文件时，不同的步骤以及其之间的持久化顺序（即依赖关系）。例如在图 2–3(a) 中表示的创建文件过程中，共有 6 个步骤：

① 分配 inode 结构；
② 初始化 inode 结构中的内容；
③ 分配目录项；
④ 初始化目录项中的内容；
⑤ 将 inode 信息写入到目录项中；
⑥ 将目录项写入到父目录的目录结构中；

图中的虚线表示该步骤持久化时所应遵循的顺序。如在将目录项写入到父目录的目录结构中之前，应保证目录项已经被初始化，且 inode 信息已经写入到了该目录项。

在 PMFS 等使用日志机制的文件系统中，日志是保证多个步骤原子持久化的主要手段，但其让文件系统操作中的依赖关系更加复杂。图 2–3(b) 中给出了在使

用日志机制时的操作步骤和持久化顺序，共包括大概 19 个持久化顺序要求，需要使用约 14 个 CLFLUSH 指令。

2.1.3 软更新技术

在文件系统中进行更新时，一般会使用多种技术保证文件系统的一致性。一些较常用的技术包括：日志（Journaling），写时复制（Copy-on-Write）和日志结构（Log Structuring）等。软更新（Soft Updates）技术[52-58] 也是用于保证文件系统一致性的技术之一。软更新技术利用异步技术，允许文件系统以内存的速度处理文件系统请求，并通过保证操作的持久化顺序保证文件系统的一致性，被实现和应用在一些基于 BSD 的 UFS 等文件系统中[59-61]。具体来说，软更新技术由三个关键部分组成：双视图、更新依赖的追踪和更新依赖的保证。下面，本节将从这三方面进行简要介绍。

2.1.3.1 双视图

在使用软更新技术的文件系统中，文件系统中具有两个视图：最新视图和一致视图。这两个视图代表了从不同角度所看到的文件系统的不同状态。最新视图通常由内存和存储设备上的数据共同组成，表示文件系统将所有操作均已执行完毕时，所呈现出来的最新状态，是应用程序当前所能观察到的状态。而一致视图则通常在存储设备之上，表现为：若当前发生崩溃，在重启之后，用户或应用程序所能看到的文件系统中所保存的文件系统状态。换句话说，一致视图中的文件系统状态是持久且一致的状态。而一些刚刚进行，但尚未（完全）持久化的操作，可能会在最新视图中却并不在一致视图中。

双视图是软更新技术能够以内存操作的性能完成文件系统操作的关键之一。通过将文件系统的最新视图和一致视图分开，使用软更新技术的文件系统，可以在内存中的最新视图上进行更新操作，并将这些更新操作之间的依赖关系记录下来。此后，文件系统可以在后台异步地将这些更新操作按照记录的顺序，持久化到一致视图之中，以完成数据在一致视图的更新，保证数据操作的持久化。

图 2–4中展示了一个软更新文件系统中双视图的示例。其中在一致视图中有三个目录和三个文件，而在最新视图中，文件“成绩.TXT”被创建，同时，日志目录中的“成绩修改.LOG”文件被删除。这两个操作刚刚在内存中执行完毕，并没有持久化到存储设备之中。因此这两个操作的效果只在最新视图中存在，在一致视图中没有体现。此示例只展示了文件粒度的操作，而在实际的软更新双视图中所能拆分的操作粒度更小，文件元数据、文件结构中的修改均会展现在双视图中。

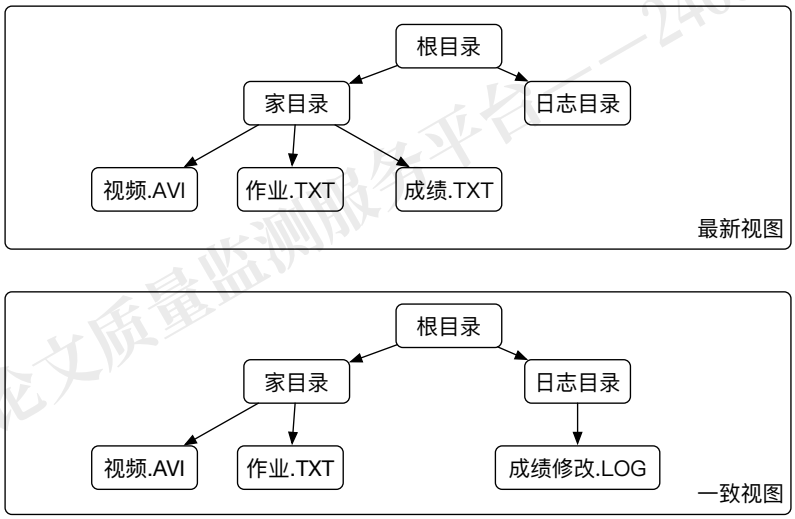


图 2–4 软更新技术中文件系统双视图的示例

2.1.3.2 依赖追踪

依赖追踪是通过软更新保证文件系统一致性的重要一环。由于机械硬盘的访问速度比内存较慢，如果每次文件系统操作均将更新应用到机械硬盘中，则每次操作均需要等待缓慢的机械硬盘操作完毕，影响文件系统和应用程序的性能。为了能够以内存的高性能完成文件系统操作，软更新技术利用双视图中的最新视图处理文件系统请求，以允许文件系统快速地处理请求并返回到应用程序继续运行。而为了保证这些文件系统修改能够最终持久化到存储设备之上，软更新技术需要将这些文件系统请求，以及这些请求的先后顺序记录下来。

在进行依赖追踪时，软更新技术将每个文件系统请求拆分成更小的步骤，并根据每个步骤所修改的不同数据结构，追踪记录不同请求间的前后顺序。

图 2–5是软更新论文[53] 中对新增一个目录项操作的依赖追踪结构示例图。图中有三种结构：inode 的依赖追踪结构 `inodedep`、目录数据页的依赖追踪结构 `pagedep`、表示增加目录项的操作结构 `diradd`。图中表示了两个增加目录项的操作，分别增加了 `foo` 和 `bar` 两个目录项。图中的箭头表示依赖关系。在目录中增加目录项只是一个文件系统请求中的一个小步骤。对于这个步骤，软更新技术需要使用图 2–5 中所示的大量辅助结构来记录复杂的依赖关系。对于一个完整的、更加复杂的请求来说，软更新技术需要记录的辅助结构和依赖关系更加复杂。

在进行依赖追踪时软更新技术遵循三条规则：

1. 在一个结构被初始化之前，永远不要（使用指针）指向该结构；
2. 只要有指针还指向某结构，永远不要将该结构所占用的资源作为他用；

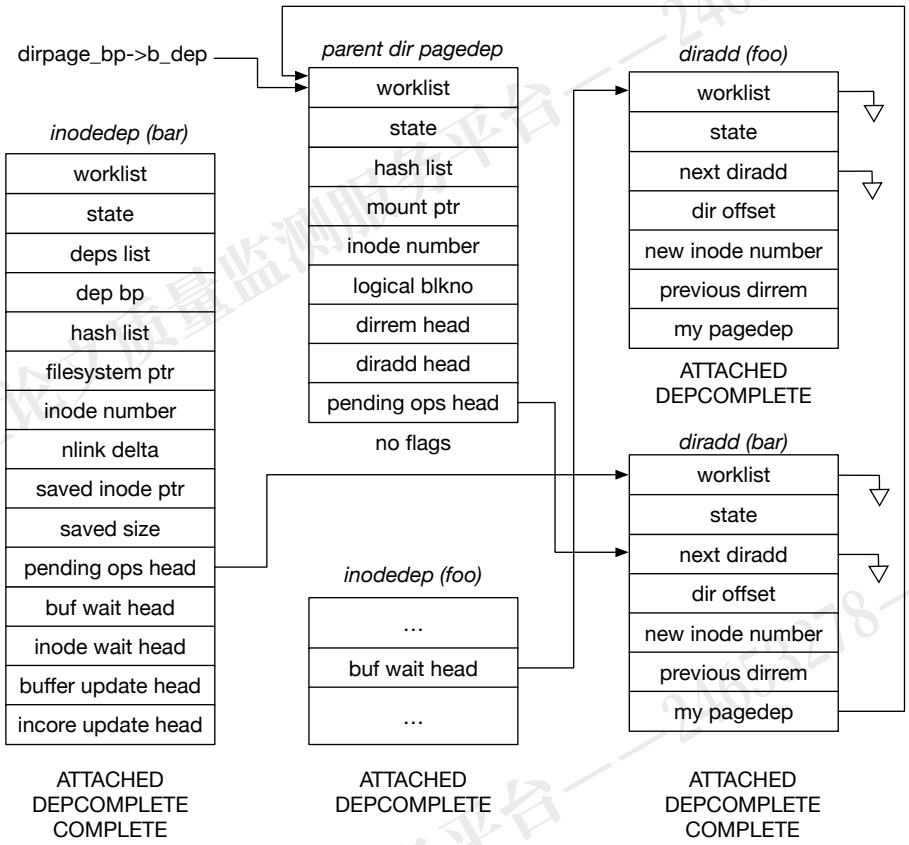


图 2–5 软更新技术论文中所展示的依赖追踪结构示例

3. 在新的指针被设置成指向一个有用的（活的）资源之前，永远不要将指向该资源的旧指针清空。

这三条规则在文件系统中均有所应用。对于第一条规则来说，在创建文件时，需要分配一个新的 inode 结构，初始化该 inode 结构的内容，并将该 inode 的 inode 号写入到父目录的一个目录项中。在这个过程中，需要保证在将 inode 号写入父目录目录项之前，inode 结构就已经被初始化完毕。如果违反了这个顺序，若在 inode 号写入父目录的目录项之后，发生断电，则由于此时 inode 结构还未被初始化完毕，机械硬盘中的父目录目录项会指向一个包含垃圾数据的 inode 结构。当机器重新启动之后，访问到此目录项时，文件系统极有可能无法发现此 inode 结构中数据是未初始化的，导致错误地访问了数据，进而导致程序出错、系统崩溃等情况。

第二条规则在文件系统中的应用可以由一个例子解释：当向一个文件中写入数据，但该文件的现有数据块不足时，文件系统会为此文件分配新的数据块，并将新分配的数据块加入到该文件的索引结构中。第二条规则保证了在该新分配出的

数据块被写入文件的所有结构之前，文件系统中不存在指针指向该数据块。如果此规则被违反，则该新分配处理的数据块可能会被两个不同的结构所指向（一个结构是前面提到的被写入的文件的索引结构，另一个结构是在分配数据块时，已经指向该数据块的结构），而这两个结构彼此不知道对方的存在，将同一个数据块中保存不同文件的数据，甚至将一个数据块所占用的存储空间用于不同功能。这将导致文件系统中的数据错乱，如文件 A 的数据被文件 B 错误的覆盖；数据泄露，如从用户 A 的机密文件中的密码被用户 B 以普通文件方式读取；应用程序、文件系统或操作系统的崩溃，如文件数据覆盖了元数据，导致错误的指针访问等。

第三条规则在文件系统中的应用主要表现在重命名（`rename`）这个文件系统操作上。重命名操作可以将文件 A 从目录 B 中移动到目录 C 中。在过程中，文件系统需要将目录 B 中指向文件 A 所对应的 inode 的目录项删除，并在目录 C 中创建指向该 inode 的目录项。第三条规则要求先在目录 C 中创建指向该 inode 的目录项，再在目录 B 中删除对应目录项。如果此顺序被违反，则在目录 B 中的目录项被删除，但目录 C 中的目录项还未被创建时，发生断电，则在存储设备上将不会有任何目录项指向文件 A 对应的 inode，文件 A 以及其数据在文件系统中消失。这违反了文件系统对用户的保证。如果按照第三条规则的顺序要求进行操作，则唯一可能发生的问题是断电后目录 B 和目录 C 中的目录项均存在，且同时指向文件 A 的 inode。这导致在目录 B 和目录 C 中均可访问该文件，虽然也违背了重命名操作的语义，但是至少文件数据是可访问的，不会影响文件系统的使用。此外，这种问题是可以通过文件系统的一致性检查工具（如 fsck）检查出来，并进行修复。

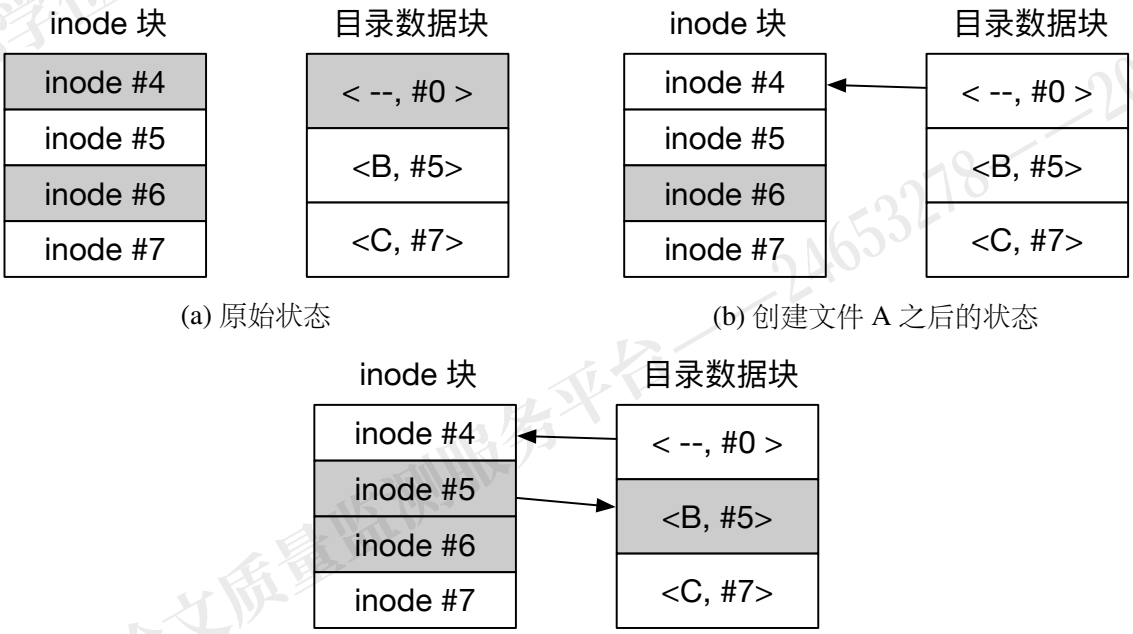
软更新中所提出的个三个顺序要求，是保证文件系统一致性的关键，也是软更新技术中判断依赖的重要标准。

2.1.3.3 依赖保证

依赖保证是软更新技术保证文件系统数据持久性的重要一环。依赖保证按照依赖追踪中所记录的依赖关系将修改持久化到存储设备之中，使得修改后的数据在断电之后依然能够得以保存。在语义上，依赖保证是将最新视图中的数据向一致视图进行同步的过程。

按照依赖追踪中记录的依赖，将更新写入存储设备似乎并不复杂。然而实际上，由于存储设备的块粒度使用方法，软更新在此过程中会遭遇到一个严重的问题：循环依赖。

循环依赖 在理想情况下，文件系统中的数据结构可以根据被记录下来的依赖顺序依次被写入到存储设备中。然而实际中，为了满足存储设备以块为粒度的访问模式，文件系统会将多个数据结构保存在同一个存储块中，以避免存储空间的浪费。如在图 2–6(a)中，四个 inode 结构被保存在同一个存储块（称为 inode 块）中，三个目录项结构被保存在同一个存储块（称为目录数据块）中。这种将多个同种数据结构保存在同一个存储块中的做法，除了可以充分利用存储空间之外，还可以增加文件系统访问的空间局部性——访问相邻的结构时只需要读取一个存储块。因此这种设计在文件系统中是比较常见的。



(a) 原始状态

(b) 创建文件 A 之后的状态

(c) 继续删除文件 B 之后的状态

图 2–6 软更新论文中的循环依赖示例

然而正是这种共享存储块的设计，造成了软更新中的循环依赖问题，影响软更新的依赖保证。图 2–6 是软更新技术论文[54] 中介绍循环依赖产生的一个例子。图中的灰色结构为空闲结构，其可以被再分配使用。图 2–6(a) 展示了最初的状态，inode 块中的 4 号和 6 号 inode 结构是空闲的，5 号和 7 号 inode 被文件系统用于保存文件的 inode。根据目录数据块中所记录的目录项可以看出，文件 B 对应了 5 号 inode，文件 C 对应了 7 号 inode。而第一个目录项是无效目录项，未被使用。图 2–6(b) 在图 2–6(a) 的基础上，新创建了文件 A。4 号 inode 结构被分配给文件 A 使用，目录项信息记录在目录数据块中。根据此前依赖追踪中的第一条规则，4 号 inode 结构需要先被初始化完毕，才能被目录项指向。因此创建文件 A 的操作

会记录一条依赖关系：inode 块应在目录数据块之前被写入存储设备。此关系由图中的箭头表示。图 2–6(c) 在图 2–6(b) 的基础上继续进行删除文件 B 的操作。文件 B 对应的 inode 结构为 5 号 inode，因此图 2–6(c) 中的 5 号 inode 结构和原文件 B 的目录项被标记为无效。根据此前依赖追踪中的第二条规则，5 号 inode 被标记为无效之前，应先将指向其的指针全部删除，在此处为先将目录项标记为无效。因此删除文件 B 的操作产生了另一条依赖关系：目录数据块应该在 inode 块之前被写入存储设备。这个关系由图中 inode 块指向目录数据块的箭头表示。此时，我们可以发现，图中的两个箭头形成了一个环，即产生了环形依赖。此处环形依赖的语义是：inode 块应在目录数据块之前被写入存储设备，同时，目录数据块应该在 inode 块之前被写入存储设备。这是自相矛盾的。其实如果能够将 inode 块和目录数据块原子地同时持久化，也能够保证文件系统的一致性不被破坏。但一般的存储设备[①]通常无法保证两个存储块同时被持久化写入。且多个不同的存储块之间可能会形成更复杂的循环依赖关系，只原子持久化两个存储块无法满足所有情况。

回滚和前滚 为了能够顺利地按照依赖将更新持久化到存储设备中，软更新技术提出使用回滚（Roll-back）和前滚（Roll-forward）技术解决循环依赖问题。在进行依赖追踪时，软更新技术需要记录足够多的信息，使得操作可以被临时撤销（即回滚）。当软更新遇到循环依赖时，其首先根据记录的信息，将内存中造成循环依赖的操作回滚。回滚后，循环依赖被临时打破，软更新技术可以按照其余的依赖关于将操作持久化到存储设备中。在持久化完毕之后，软更新技术再将刚刚回滚了的操作在内存中重做一遍（即前滚），并继续按照依赖关系将该操作持久化。在回滚和前滚之间，软更新技术会阻塞对该数据块的所有读写请求，以防止应用程序看到该数据块中错误的中间状态。在解决了循环依赖之后，软更新技术可以顺利地完成文件系统各个操作的持久化，且能保证在任何时刻发生断电，存储设备上保存的信息均是一致的。

2.1.3.4 软更新与非易失性内存

软更新技术通过双视图、依赖追踪和依赖保证提供文件系统的一致性，相较于其他保证文件系统一致性的方法，软更新技术具有诸多优势：(1) 软更新允许文件系统异步持久化元数据更新，因此文件系统处理用户操作的性能可以达到内存性能。(2) 软更新对文件系统的原有设计无侵入性修改。(3) 软更新能够保证存储设备中的状态始终为一致的。因此发生断电并在重启之后，文件系统可以直接挂

① 目前有一些设备具有事务支持，可以原子更新多个块设备。

载使用，无需等待冗长的检查和恢复过程。

然而，软更新技术也有一些不足：(1) 软更新技术中的依赖追踪需要在内存中记录大量辅助结构，且依赖之间关系非常复杂，这使得系统更容易出错，代码难以演进和维护。(2) 依赖保证过程中需要进行回滚和前滚操作。这进一步增加了系统的复杂性，且会导致存储块在进行回滚和前滚操作时不能用于处理文件系统请求，影响文件系统的并发性能。我们发现，造成这些问题的主要原因在于软更新技术以细粒度结构进行更新依赖的追踪，然而传统存储设备却需要使用存储块这种粗粒度的接口进行访问。软更新的需求与存储设备特性的错配，导致了软更新在应用中大量使用复杂的辅助结构、环形依赖问题和复杂的回滚与前滚操作，让软更新的设计和实现变得十分复杂。

非易失性内存的出现，为软更新技术的应用带来了转机。非易失性内存具有字节级的细粒度存储接口，因此在非易失性内存文件系统之中使用软更新技术，可以避免复杂的依赖追踪和环形依赖问题，从而极大地简化软更新技术的设计和实现。另一方面，软更新的优势同样会为非易失性内存文件系统带来新的益处：使用软更新技术的非易失性内存文件系统可以异步进行文件操作的持久化，从而将耗时的缓存行刷除等操作从关键路径中消除，降低文件系统请求的处理时延，提高文件系统性能。此外，软更新技术对文件系统结构没有特殊要求，因此非易失性内存文件系统可以根据非易失性内存的特性设计和使用最高效的文件组织结构。同时，软更新技术在存储设备中状态的始终一致性，在非易失性内存文件系统中可得以保留，从而避免崩溃后的文件系统检查和恢复过程。将软更新技术与非易失性内存技术相结合，可以取长补短，提升非易失性内存文件系统的性能。

2.2 系统设计

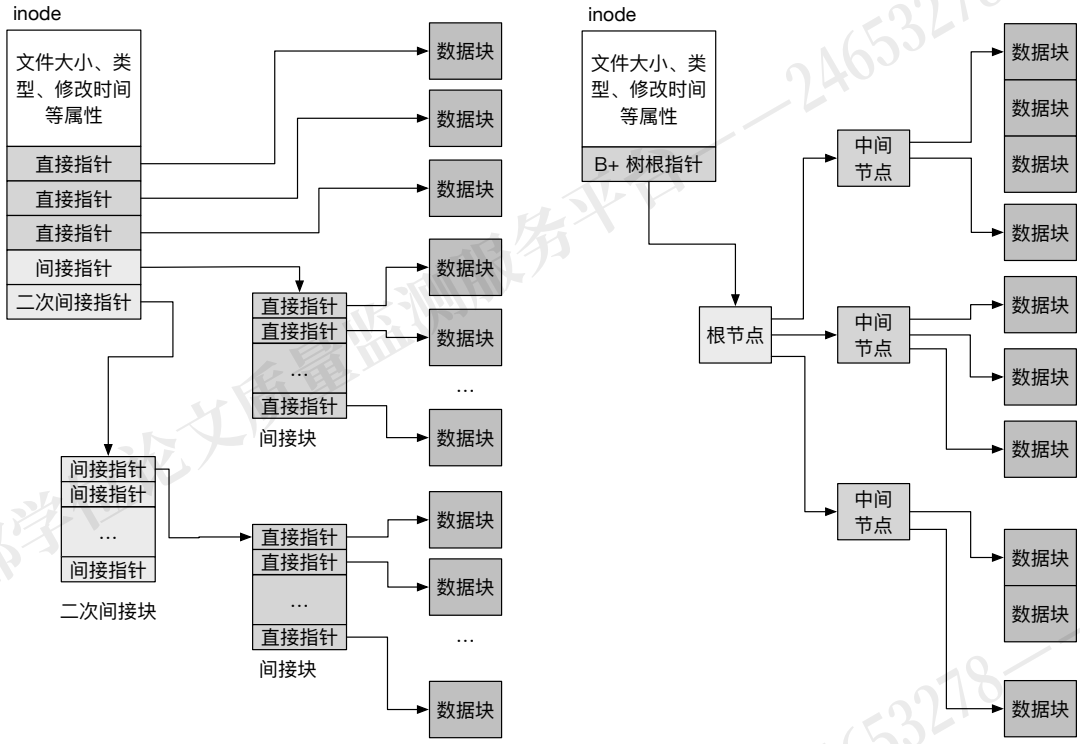
为了充分利用非易失性内存的高性能和可字节寻址的特性，我们基于软更新的思想设计了一个新型内核态非易失性内存文件系统 SoupFS。SoupFS 结合了非易失性内存与软更新技术各自特点，同时发挥两者的优势，实现了文件系统更新的异步持久化以及更简单直接的依赖追踪和保证方法。在本节中，我们将详细介绍 SoupFS 文件系统的设计方法和实现。

2.2.1 文件结构设计

文件是文件系统中最核心的概念。在文件系统中，文件可以进一步划分为不同类型，包括常规文件、目录文件（简称为目录）、符号链接文件（也称为软链接文件，或简称为符号链接、软链接）、设备文件、FIFO 文件、套接字（SOCK）文

件等。在这些不同种类的文件之中，常规文件和目录文件是最常用的，且其保存格式也比较复杂。为了充分发挥非易失性内存的性能优势，SoupFS 对常规文件和目录文件的结构进行了特殊的设计，使这些结构更加符合非易失性内存的访问模式，同时也更利于软更新技术的应用。

常规文件的结构 常规文件用于保存用户定义的任意数据。文件系统的使用者可以将常规文件看为一个可变长的字节数组——通过“文件内偏移地址”作为索引，使用者可以访问到文件中的每个字节。常规文件的结构维护了“文件内偏移地址”到对应字节的位置的映射，用于对文件的内容的定位、读取和修改。传统的文件系统考虑到块设备的块粒度访问模式，通常以存储块保存相邻的文件数据，这些存储块称为数据块（Data Block）。因而常规文件的组织结构一般维护了“文件内偏移地址”到数据块块号的映射，在找到数据块后再从中找到需要访问的字节。



(a) 使用间接块索引文件数据　　(b) 使用 B+ 树索引文件数据

图 2–7 文件系统中常用的文件数据索引方法

图 2–7 展示了两种常见的常规文件的结构：间接块（Indirect Block）和 B+ 树（B+ Tree）。间接块（图 2–7(a)）是一种比较简单的文件数据索引结构。其在 inode 结构中维护多个指针。这些指针可以分为几种：直接指针直接指向文件数据块；间

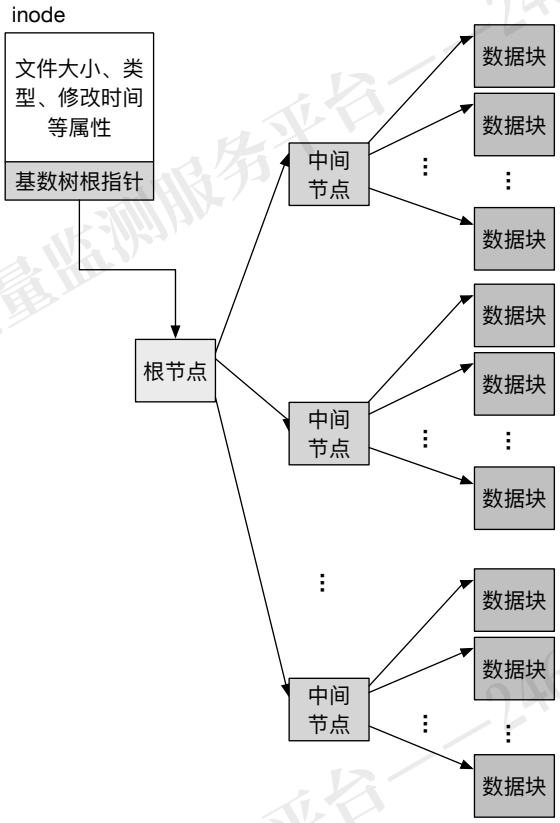


图 2–8 使用基数树索引文件数据

接指针指向一个间接块，间接块中保存了一组直接指针，进而指向数据块；二次间接指针指向一个二次间接块，其中保存了一组间接指针。使用间接块的方法组织管理文件数据有至少两个优势：一是编码简单；二是能够充分利用 inode 结构中的空间，根据 inode 结构的大小，设计（调整）保存在 inode 结构中不同类别的指针个数。

B+ 树（图 2–7(b)）是另一种常用的文件数据组织方式。顾名思义，这种方式使用 B+ 树保存“文件内偏移地址”到数据块的映射。在 inode 结构中，保存有一个 B+ 树的根指针，指向一个 B+ 树的根节点。通过 B+ 树的根节点，文件系统可以访问到 B+ 树中所索引的数据块，进而找到文件数据。使用 B+ 树的方式有以下几种优势：一是 B+ 树结构刚好符合了块设备的访问特点，因此更加高效；二是使用 B+ 树结构可以动态调整树的高度，不会在结构上限制文件的最大大小；三是可以将多个连续的数据块组成区段（Extent），然后在 B+ 树中保存“文件内偏移地址”到“区段开头块号和区段长度”的映射。在访问连续的数据时，可以在一个区段内访问多个数据块，减少索引的查询次数，提高数据访问性能。然而相对于此

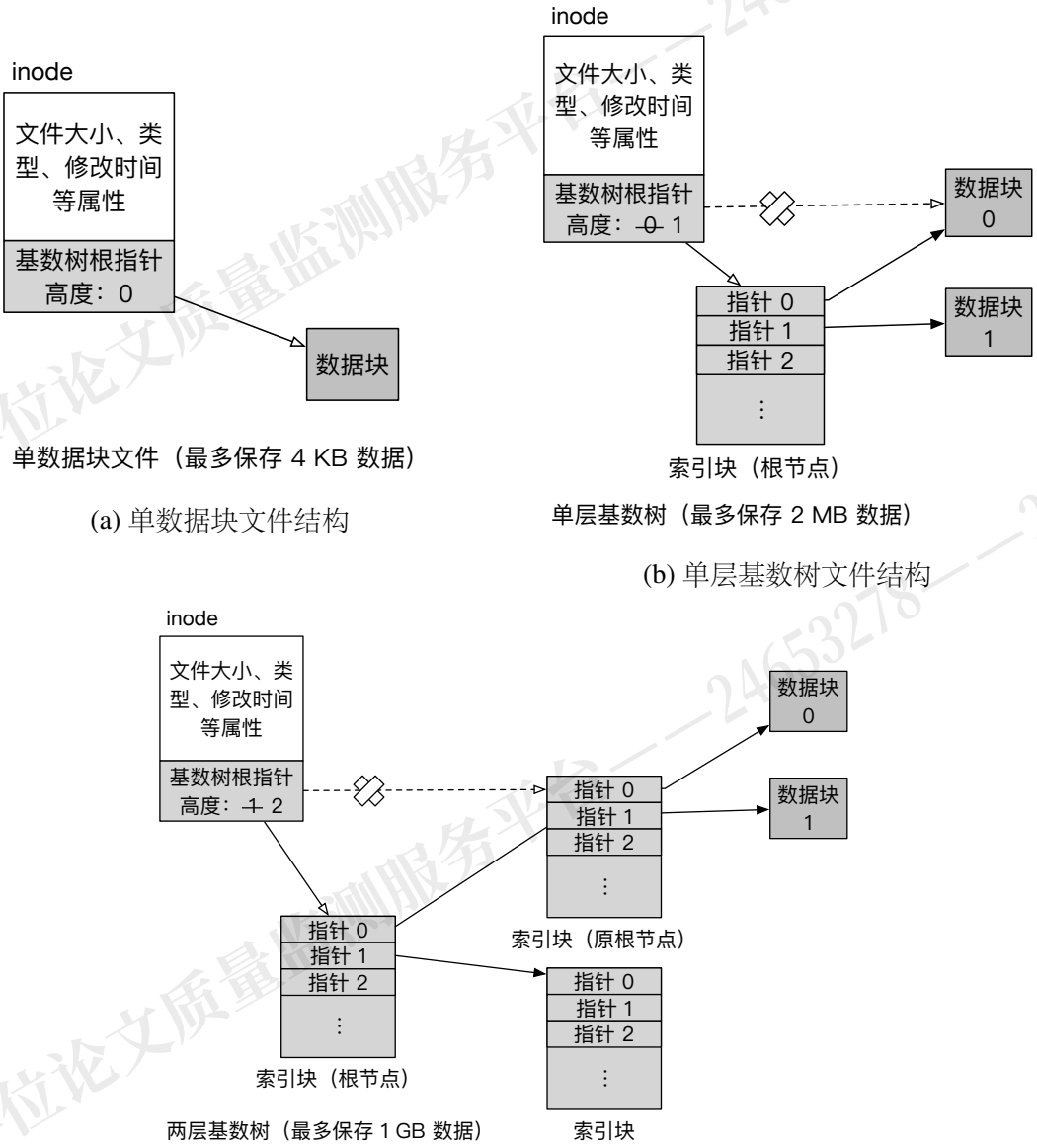
前的间接块方法，B+ 树的实现涉及分裂、合并等操作，实现较为复杂。

SoupFS 使用了另外一种结构组织常规文件数据——可变高的基数树（Adaptable Radix）。图 2–8 中展示了一个使用基数树的常规文件索引结构。其结构与间接块结构和 B+ 树结构均有些相似，却又有所不同。其只有一个根节点，直接保存在 inode 结构中。基数树中的每个节点中保存了一组指针，根据该节点所在的层数，指向下一级节点或数据块。在此结构上，SoupFS 设计实现了可变高的基数树，与间接块的方式相比，可变高基数树可以随着文件数据量的增长变高，不会限制文件大小；与 B+ 树的方式相比，可变高基数树不需要分裂和合并等操作，实现更加简单。

我们通过一个例子进一步介绍 SoupFS 中的可变高基数树。图 2–9(a) 展示了一个零层基数树文件结构（即单数据块文件）。在这种情况下，文件中最多只有一个数据块，由 inode 结构中的指针直接指向。假设数据块的大小为 4 KB，则该文件最多能保存 4 KB 数据。图 2–9(b) 展示了一个单层基数树文件结构。单层基数树索引由一个索引块和多个数据块组成。每个索引块中可以保存最多 512 个指针 ①，每个指针保存了一个数据块的位置。文件 inode 结构中的指针指向了该结构中唯一一个索引块。当进行文件数据查找时，文件系统通过 inode 结构中的指针找到该索引块，并根据所要访问的数据在文件内的偏移量，计算出该数据所在的数据块的位置，在索引块中找到目标数据块。图 2–9(b) 同时也展示了一个不足 4 KB 的文件增长到超过 4 KB 但不超过 2 MB 时的操作过程。在一个不足 4 KB 的文件索引中，文件元数据中保存了数据块 `0` 的位置。在进行文件增长时候，文件系统首先通过分配器分配出一个索引块，初始化其中的指针，并进行持久化。其中的第一个指针指向原有的数据块 `0`。随后文件系统根据文件新的大小分配数据块，并使用索引块中的后续指针指向这些数据块（如图中的数据块 `1`）。当索引块已经初始化并持久化完毕之后，文件系统通过原子写入将文件元数据中的基数树根指针（原本指向数据块 `0`）更新为指向索引块，并增加树高②。在基数树根指针的修改持久化之后，文件便成功增长了一层，且文件中的原有数据自然地出现在新的基数树中，无需进行数据迁移。通过同样的方式，文件可以进一步进行增长为多层基数树结构，如图 2–9(c) 展示了一个双层基数树结构。其中新增出来的“层”同样由索引块组成，整个基数树结构所能保存的文件数据量为单层基数树结构的 512 倍。

① 在本章中，数据块大小为 4 KB，指针大小为 8 字节。

② 为了进行原子写入，树高与根指针编码在相同的 8 个字节中。



(a) 单数据块文件结构

(b) 单层基数树文件结构

(c) 双层基数树文件结构

图 2–9 SoupFS 中的可变高基数树

目录文件结构 目录文件是文件系统中另一个非常重要的文件类型，其中记录了“文件名”到“inode 号”的映射。每个“文件名”到“inode 号”的映射被称为一个目录项。目录文件是文件系统组织文件并呈现出树状结构的关键，在文件系统处理文件路径时将被频繁使用。

一种常见的目录组织方式，是复用常规文件的结构，线性地保存目录项。图 2–10 展示了这种线性目录结构。在该目录文件中，常规文件的索引结构将多个数据

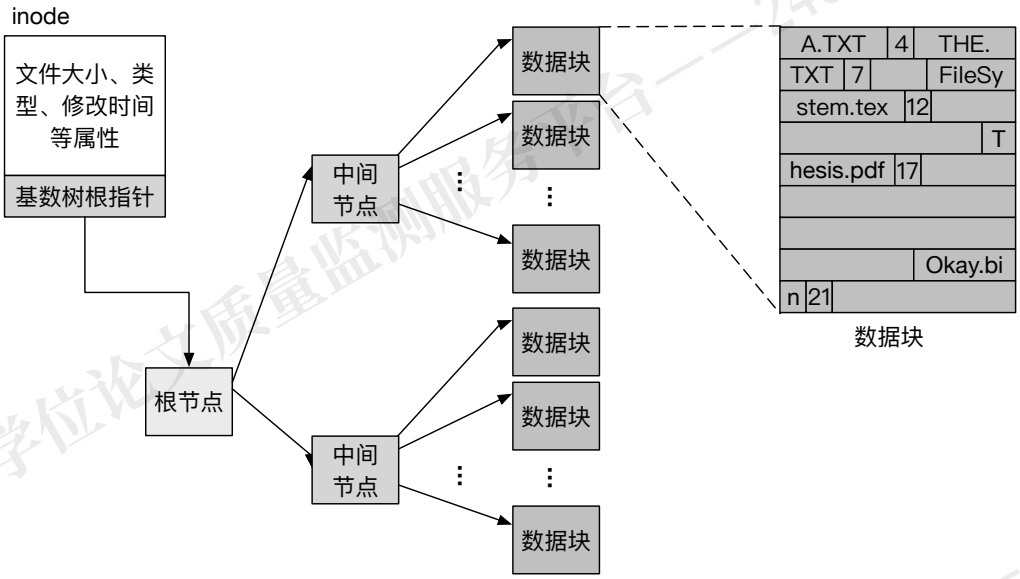


图 2–10 线性目录结构

块组织在一起。每个数据块中，文件系统线性地保存了一个个目录项。每个目录项包括一个变长的文件名[①]和一个 inode 号。当在目录中查找文件时，文件系统需要线性地扫描目录中的所有目录项，依次对文件名进行匹配。若文件名匹配成功，则返回对应的 inode 号，表示查找到该文件。在目录中删除文件时，文件系统首先找到对应的目录项，之后将该目录项所在位置标记为无效，成为“空洞”，以便后续在创建文件时使用。在目录中创建新文件时，文件系统需要在文件的数据块中找到不包含有效目录项、且足够大的“空洞”，将新文件的文件名和其 inode 号记录下来，成为一个新的有效目录项。若无法找到符合条件的“空洞”，则分配新的数据块，加入到索引结构中，再写入新的目录项。

这种线性目录结构复用了常规文件的索引结构，线性保存目录项的实现比较简单，且与块设备的块粒度访问相匹配。但其也有一些不足。其一是性能问题。每次在目录中查找文件时，都需要进行一次线性查找。如文件不存在，则需要扫描整个目录中的数据块，才能确认文件不存在。同样地，在添加新的目录项时，也需要线性扫描以寻找合适的“空洞”。当目录中目录项数量较多时，线性查找的速度将影响文件系统的访问性能。第二是伪共享（False Sharing）问题。多个目录项会保存在同一个数据块中。这种保存方式会造成软更新技术中的循环依赖问题，让依赖追踪和依赖保证变得更加复杂，系统实现难以维护和演进。在应用在非易失性内存中时，虽然不需要按照块粒度访问，但由于目录项的长度不固定，目录项

① 文件名由用户在创建文件是指定，文件系统无法提前获知其长度。

的保存也不会按照缓存行对齐，因此，不同的目录项可能会保存在相同的缓存行中。由于非易失性内存的访问粒度是缓存行①，在软更新技术应用到非易失性内存时，这种线性索引结构同样会产生循环依赖问题。

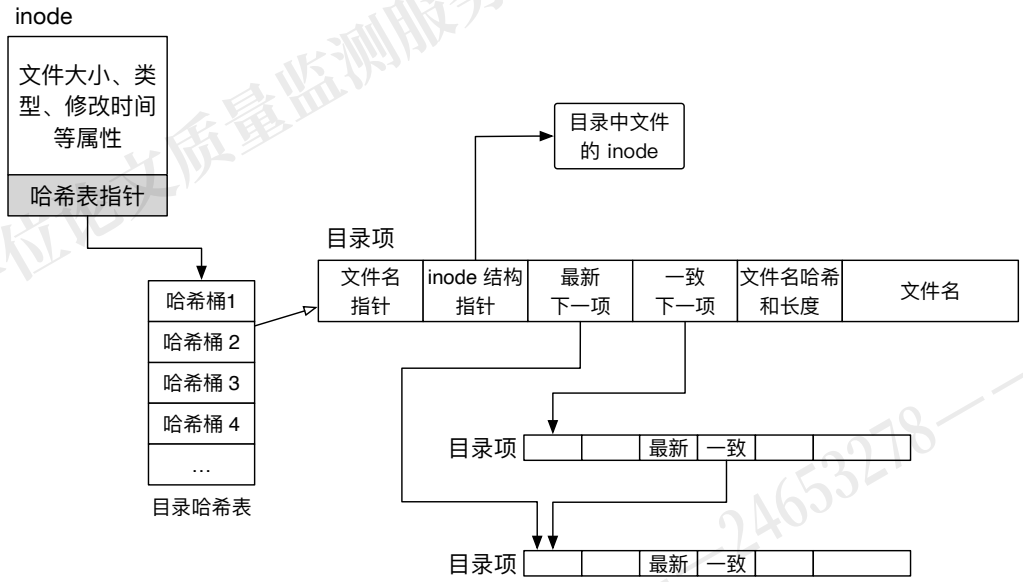


图 2–11 SoupFS 中基于哈希表的目录结构

为了避免线性扫描带来的性能问题和线性存放的伪共享问题，SoupFS 借助非易失性内存的特性，使用哈希表重新组织目录内容。图 2–11 中是 SoupFS 中所使用的哈希表目录结构。目录 inode 结构中保存了一个指针（“哈希表指针”），指向该目录的哈希表结构。哈希表结构占一个内存页大小，内容为一个哈希桶指针的数组。每个哈希桶指针指向一个目录项结构。每个目录项是一个固定大小的结构，大小为 64 字节，即一个缓存行的大小。目录项中保存的数据包括：文件名指针、指向 inode 结构的指针、“最新”下一项、“一致”下一项。由于文件名的长度不固定，目录项中的文件名以一种特殊的方式进行保存，具体方法将在后文章节 2.2.2 中具体介绍。值得一提的是，SoupFS 在目录项中保存了文件名的哈希值和长度，以便快速进行文件名的比较。指向 inode 的指针指向该目录项文件所对应的 inode 结构。两个下一项指针，表示哈希表中保存的下一个目录项。由于 SoupFS 中使用了软更新技术，每个目录项可以同时出现在两个哈希表中（即目录的双视图），因此有“最新”下一项指针和“一致”下一项指针。这两个下一项指针的用法在后文章节 2.2.4 中会有进一步解释。

基于哈希表的目录结构设计，除了符合非易失性内存的访问特点之外，还能

① 虽然非易失性内存可以以字节粒度寻址，但是由于 CPU 缓存的存在，其访问粒度实际上是缓存行。

够简化软更新技术中的依赖关系。哈希表中的结构均按照缓存行对齐，不同结构之间相互不影响。由于非易失性内存可字节寻址的特性，这种缓存行对齐的目录设计避免了伪共享问题，从而消除循环依赖。另外，由于 SoupFS 在处理目录插入操作时，会分配一个新的目录项结构，SoupFS 只需要使用一个指针记录该新目录项结构的位置，不需要使用复杂的结构保存依赖关系。具体来说，在向目录哈希表中插入新的目录项时，SoupFS 只需要记录该操作类型（即“目录项插入”）和与该操作相关的指针（即指向所插入的目录项的指针）就足够了。有了这些信息，SoupFS 在进行依赖保证时就能够按照正确的顺序完成元数据的持久化。具体的依赖追踪方法，将在后续章节 2.2.5 中进行更详细的介绍。

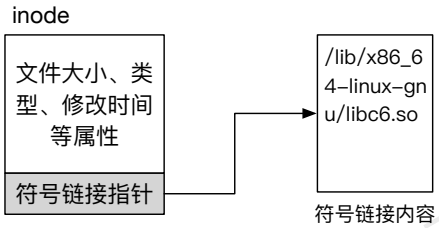


图 2–12 SoupFS 中的符号链接文件结构

符号链接文件 符号链接是一种特殊的文件类型，其中保存了一个路径。在 Linux 内核中，文件系统只需要将这个路径保存下来即可，符号链接上的大多数操作均由 Linux 内核中的 VFS 来完成。为了保存符号链接的内容，SoupFS 使用了图 2–12 中的结构——为符号链接分配一个单独的非易失性内存页，并将路径保存在这个内存页中。符号链接文件的 inode 结构中保存了一个指针，指向了该内存页。

其他文件类型 对于其他常见的文件类型，如设备文件、FIFO 文件、套接字文件等，其信息一般直接在 inode 结构中保存，因此不需要使用额外的结构。

2.2.2 内容无关的分配器

一些文件系统会预留专用的空间用于分配特定的结构。如 Ext4 会在存储布局中预留 inode 表，用于 inode 结构的存放，所有的 inode 结构均保存在此 inode 表中。这种方法有利于 inode 结构的管理，在基于机械硬盘的文件系统中可以利用 inode 访问的局部性，减少磁盘的随机访问次数，提高性能。这种性能提升在进行的文件系统修复（fsck）等操作时尤为明显。然而这种设计也有其所不足：它固定了一个文件系统中可用的 inode 的数量上限。当 inode 表中的 inode 全部被使用之

表 2–1 SoupFS 中的所有数据结构和其占用空间大小。SoupFS 中的所有数据结构均可通过 64 字节和 4 KB 大小的空间分配来保存

数据结构	大小	分配大小
inode	64 字节	64 字节
文件名	变长	64 字节
目录项（不包括文件名部分）	32 字节	64 字节
哈希表（桶数组）	4 KB	4 KB
基数树节点	4 KB	4 KB
数据块	4 KB	4 KB

后，即使存储系统中的其他位置有空闲空间，文件系统也无法再创建新的文件。当被固定的结构越多，文件系统中不同结构数量限制也就越多，因而这种设计会阻碍文件系统对存储设备空间的充分利用。

在非易失性内存上，随机访问的性能与顺序访问的性能差距不大。因此没有必要为 inode 等结构的随机访问进行特殊优化。上述的专用区域的优势并不明显。因此，在 SoupFS 中，我们设计使用了一个内容无关的分配器。内容无关的分配器将整个非易失性内存空间视为一个大的内存池。在无需知道内存用途的情况下分配内存。内容无关的分配器打破了文件系统中各种数据结构之间的界限，在保证性能和正确性的同时，使得内存管理（即存储空间管理）更加灵活和简单。

在使用内容无关的分配器的同时，SoupFS 针对文件系统的特点，进一步对分配器进行简化。表 2–1 给出了在 SoupFS 设计中的所有数据结构，以及每个结构所占用的空间大小。可以发现，除了文件名长度不固定，目录项的大小为 32 字节之外，其他结构的大小可以分为 64 字节和 4 KB（即 4,096 字节）两类，分别对应着缓存行大小（64 字节）和内存页大小（4 KB）。因此，SoupFS 中所使用的分配器只需要支持这两种大小的空间分配即可，这极大地简化了分配器的实现。目录项和文件名是仅有的两个不符合缓存行和内存页大小的结构。每个目录项与且只与一个文件名相对应，因此可以将两者共同考虑：为目录项分配 64 字节的空间，若文件名长度不超过 24 字节，则直接将文件名保存在目录项自身数据之外的 32 字节中（预留的 8 个字节用于保存文件名哈希值与长度）；若文件名长度超过 24 字节，则将超出部分以 56 字节为一段，分割成多段，每段保存在一个 64 字节的空间中（其余 8 个字节用于保存“下一段文件名”指针），多段直接通过指针连接在一起（类似于单链表结构）。图 2–13 中展示了在 SoupFS 中保存的一个具有超长文件名的目录项。

图 2–13 SoupFS 中目录项与变长文件名的存储示例

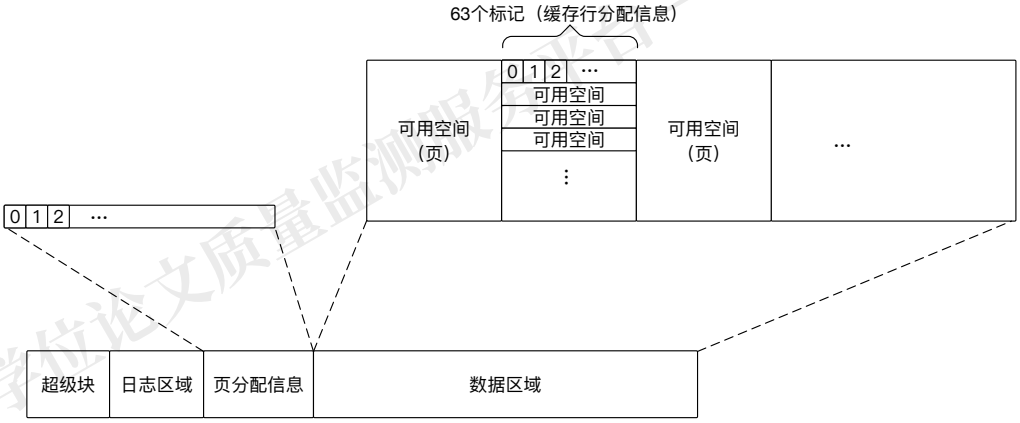


图 2–14 SoupFS 中的分配信息以及其在非易失性内存中的存储结构

由于只需要考虑 64 字节和 4 KB 大小的空间分配，SoupFS 中分配器的设计非常简单。图 2–14 展示了 SoupFS 在非易失性内存上的分配信息的存储布局。SoupFS 的分配信息可以分为两部分：页分配信息与缓存行分配信息。其中页分配信息集中存放在非易失性内存的一个区域。类似于传统文件系统中的块分配信息，页分配信息中的每一个字节，对应数据区域中的每一个页面的分配情况。分配状态包括：已分配为页、已分配为缓存行、未分配。当一个页的分配状态为“已分配为缓存行”时，其对应的页被进一步拆分使用。该页面的第一个缓存行（64 个字节）中

保存了 63 个标记位（即缓存行分配信息），依次对应该页中剩余 63 个缓存行的分配情况。

为了进一步提升分配器的性能，SoupFS 除了在非易失性内存上保存了分配信息之外，还在内存中缓存了一份分配信息。SoupFS 为每个 CPU 维护了两个 CPU 本地空闲链表（CPU-local Free List），分别记录该 CPU 可直接分配使用的空闲页和空闲缓存行。同时, SoupFS 维护了两个全局空闲链表，用于保存共享的空闲页和缓存行。以分配内存页为例，当 SoupFS 需要分配新的内存页使用时，其首先获取其所在的 CPU，然后从该 CPU 对应的空闲页链表中获取一个内存页使用即可。如果该链表为空，则首先从全局空闲页链表中批量获取一些空闲页到该 CPU 的空闲页链表，再进行上述分配。当进行内存页释放时，直接将空闲页加入到当前 CPU 的空闲页链表即可。对于缓存行，当空闲缓存行链表为空，会分配一个空闲内存页，将其拆成 1 个缓存行分配信息和 63 个空闲缓存行使用。当空闲缓存行过多时，也会将相同页面上的 63 个空闲缓存行合并成一个空闲页面。SoupFS 分配器的设计和实现来源于 ssmalloc[62]，并根据非易失性内存的特点和 SoupFS 结构的需求进行了简化。

2.2.3 操作顺序与依赖关系

完成一个文件系统相关的系统调用请求，通常需要修改多个不同的数据结构。为了保证崩溃一致性，这些修改要么被原子持久化，要么按照特定的顺序被持久化。SoupFS 使用了软更新技术，因此需要按照特定顺序持久化这些修改。此前我们已经提到，软更新技术将这些修改被持久化的顺序总结为三个规则：

R1. 不要在一个结构被初始化之前就指向它。例如，一个 inode 必须在目录项引用（即指向）它之前被初始化。

R2. 在所有指向一个资源的指针被置为空之前，不要重新使用该资源。例如，一个 inode 指向一个数据块的指针必须被置为空，然后这个数据块才能被重新分配给一个新的 inode （或其他结构）使用。

R3. 在新的指针未被设置之前，不要改变指向有效资源的旧指针。例如，在重命名文件时，在设置完新的目录项之前，不要删除一个旧的目录项。

这三个规则是软更新技术中依赖跟踪和依赖保证的准则。SoupFS 沿用软更新技术中的 R2 和 R3，但对 R1 进行了修改。根据观察，在大多数文件系统操作中，一系列结构的持久可见性[①]往往取决于单个指针修改的持久化。例如，当创建一

① 此处的持久可见性指：立刻断电后重启，是否能从文件系统中观察到该结构，与后文中的根部可到达性相同。

个常规文件时，只要其对应的目录项没有被持久地插入到非易失性内存上的哈希表中，新的目录项即使在新 inode 被初始化之前也可以指向该新的 inode 结构。为了方便叙述，我们首先定义一个结构的根部可到达性：若立刻发生断电，那么在重启之后，从文件系统的根结构开始遍历，所有可以遍历到的结构，为根部可到达结构。以根部可到达性的角度，我们前面的观察可以重新表述为：在目录项变得可以从文件系统根部可到达之前，目录项中的指针可以暂时违反 R1，且不会引起一致性问题。基于这样的观察，我们将 R1 概括为“在结构被初始化之前，永远不要让结构从根部可到达”，新的 R1 放宽了对部分修改持久化顺序的限制，从而可以用来进一步简化 SoupFS 中的依赖关系。

接下来，我们举例说明 SoupFS 不同操作中的更新依赖关系。

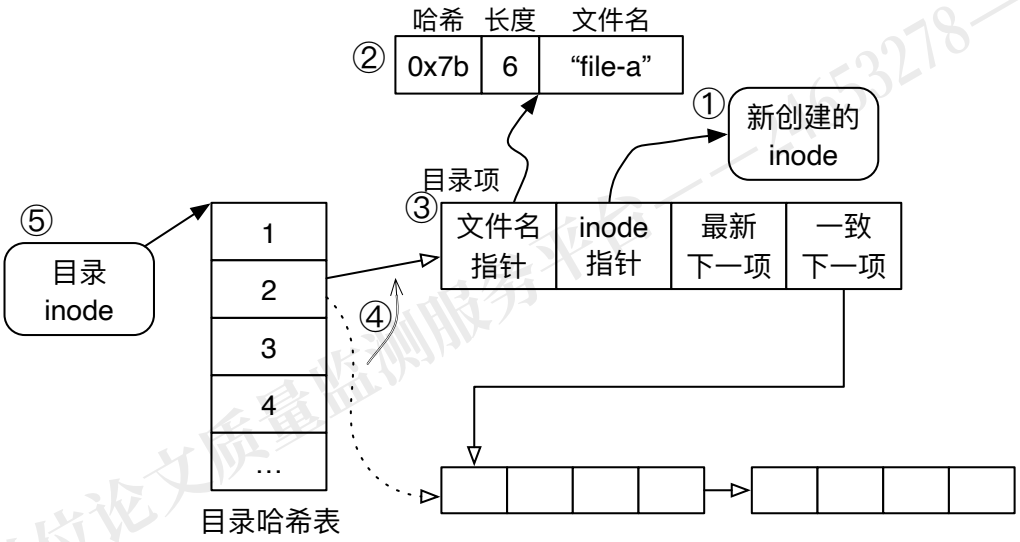


图 2–15 SoupFS 文件创建过程中的操作和依赖关系

文件创建中的依赖关系 首先以文件创建为例，如图 2–15所示，文件创建需要完成一系列的操作。

① 分配一个 inode 并初始化 inode 中的各项信息；
② 为文件名分配空间，并将文件名写入到分配的空间中（即使用文件名对这段空间进行初始化）；
③ 分配一个目录项，并将 inode 指针和文件名的指针填写到目录项中（即对目录项进行初始化）；
④ 将目录项插入到目录的哈希表中；
⑤ 更新父目录的 inode 信息；

这其中的一些步骤还可以进一步细分。比如第一步中的 inode 分配和初始化，可以继续拆开。考虑到这两步关系较为紧密，且他们之间的持久化顺序并不会影响一致性，因此合为一步进行叙述。

这些操作包括了大量的指针操作，按软更新中的更新规则会产生大量的依赖；而根据 SoupFS 中简化的 R1 规则，只需要保证两个顺序：

1. 在目录项被持久地插入到目录的哈希表（④）之前，前面的操作（① ② ③）均已完成持久化；
2. 父目录 inode 中的信息持久化（⑤）在目录项被持久地插入到目录哈希表之后（④）；

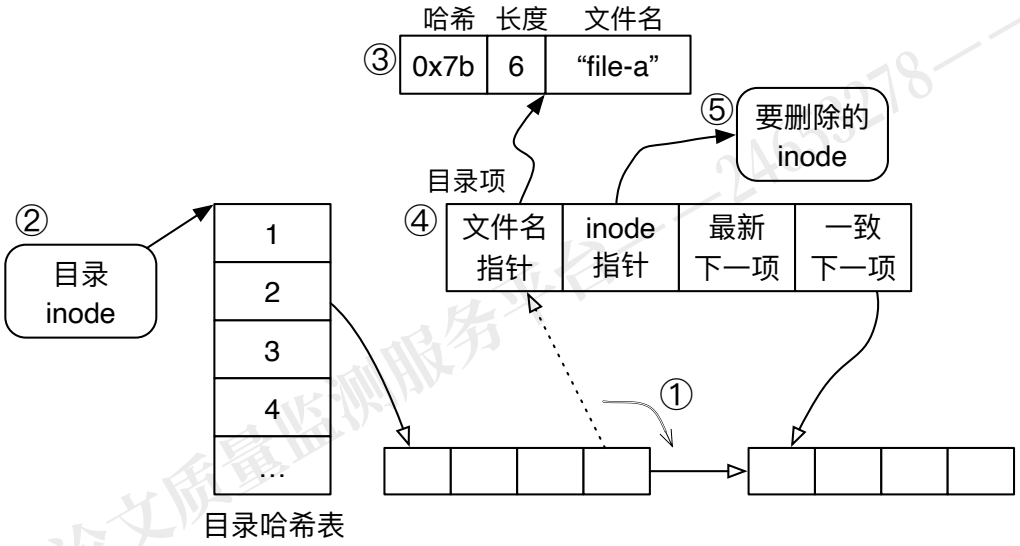


图 2–16 SoupFS 文件删除过程中的操作和依赖关系

文件删除中的依赖关系 图 2–16 中显示了文件删除过程中的操作和依赖关系，包括：

① 从目录的哈希表中删除对应目录项；
② 修改父目录 inode 中的信息；
③ 释放文件名占用的空间；
④ 释放目录项占用的空间；
⑤ 如果 inode 中的链接数为零，则释放 inode 所占用的空间；

在这种情况下，唯一需要保证的顺序，是其他操作（② ③ ④ ⑤）应该在哈希表中删除目录项（①）之后进行持久化。

目录操作中的依赖关系 目录的创建和删除与普通文件的创建和删除操作类似，唯一不同的在于 SoupFS 需要进行哈希表结构的分配、构造、销毁和释放操作。

每个目录在被创建时，需要为哈希表分配空间，并进行哈希表的构造：将所有的哈希桶进行清零操作。在目录创建时，不需要进行“.”和“..”两个特殊目录项的创建。这是由于“.”目录项总是指向当前的目录 inode，而“..”总是指向父目录，因此不需要进行特殊的保存。SoupFS 并不在目录中保存这两个目录项，因而可以避免不必要的依赖关系。

在目录被删除时，应确保其哈希表为空，即其中已经不存在有效的目录项，此后方可将哈希表所对应的内存释放。

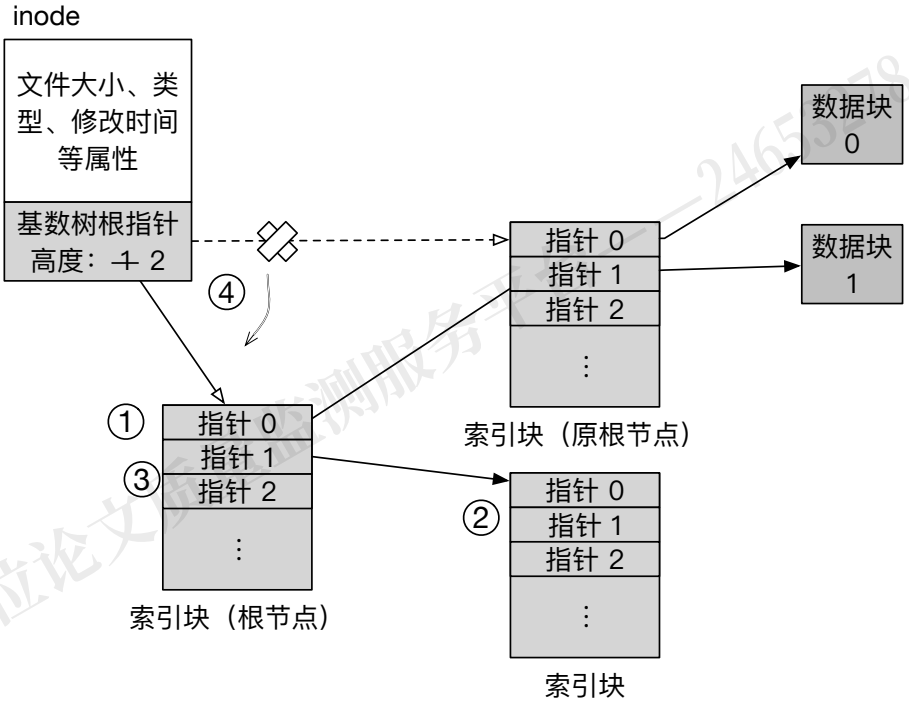


图 2–17 SoupFS 文件扩大（基数树增长）过程中的操作和依赖关系

文件结构操作和依赖关系 基数树的节点是文件结构中的元数据，用于组织存放文件数据的数据块。图 2–17 展示了一个文件中基数树从高度为 1 增长为高度为 2 过程中的操作和依赖关系。其中涉及的操作包括：

① 分配新的根节点，并进行清零和初始化（将第一个指针指向原根节点）；

② 分配新的索引块和数据块，并进行清零和初始化（此步骤取决于文件增大后的大小）；

③ 将新分配的索引块和数据块用指针连接到新的根节点所表示的树中（如将新索引块地址填入根节点）；

④ 更新 inode 结构中的基数树根指针和基数树高度；

为了保证文件结构增高过程中的崩溃一致性，在对上述修改进行持久化的过程中，只需要保证各个子结构的分配和修改的持久化操作（①②③）在基数树指针和高度变化（④）之前即可。由于基数树的根指针与高度需要配合使用，这两个变量需始终保持一致。在 SoupFS 的实现中，基数树的高度被嵌入式地保存在基数树根指针中，两者共同占用了 8 字节空间，因此可以通过指令级原子写入保证两者持久化的原子性。

若在一个文件被扩大过程中，基数树的高度不需发生变化，则 SoupFS 需要进行的操作包括：

① 索引块、数据块的分配和初始化；

② 将新分配的索引块、数据块连接到基数树中；

③ 新数据的写入；

④ 更新 inode 结构中的文件大小信息；

这其中的 ① 和 ② 取决于文件扩大的程度，如果现有文件结构中的数据块个数足以保存增大后的文件，则不需进行这两步操作。为了保证所有这些操作不会影响文件系统的崩溃一致性，SoupFS 需要保证：在新的文件大小信息被持久写入到 inode 结构（④）之前，新写入的数据（③）和元数据结构修改（①②）已经持久化完毕。这是由于，新的文件大小信息被修改之前，新写入的数据和结构修改都无法从根部可到达，因而对用户不可见，故无需保证其持久化顺序。同理，在追加写入数据的过程中，无需使用写时复制等技术保证写入的原子性。

若文件在扩大过程中，涉及到了基数树的高度增加，则 SoupFS 先进行基数树的高度增加，再进行文件的扩大和数据的写入。这两步骤的操作之间只需保证持久化顺序，同样无需进行原子性的保证。这是由于增加基数树高度的操作，不会改变文件结构的合法性，也不会影响对文件的正常访问过程。因此即使只增加了基数树高度，未进行文件扩大和数据写入，文件的一致性也不会被破坏。

在进行文件大小缩小时，操作与文件扩大是相反的。当一个文件被截断时，首先要先持久化新的文件大小，然后再释放因文件减小而不再使用的文件数据块和索引块。为了避免在文件频繁增大和缩小时存储空间被频繁地分配和释放，当空闲的存储内存空间并不紧张时，SoupFS 并不立即释放由于文件减小而空余出来的空间，而是将其继续保留在文件结构中。这样当文件再次增大时，就不需再次分配空间和修改文件结构。当空闲的存储空间低于一个阈值时，SoupFS 会在后台主

动扫描文件并进行多余空间的释放。为了进行这种优化，SoupFS 在 inode 中除了记录文件的大小之外，还记录了文件结构中有效数据块的数量，并在文件结构增大和减小过程中对其进行特殊的维护，以保证其与文件结构的一致性。具体的维护步骤此处不再展开。

2.2.4 基于指针的双视图

双视图是软更新技术中的一个关键环节：保存在于内存中的数据始终反映文件系统的最新状态（即最新视图），而持久保存在存储设备中的数据始终是一致的（即一致视图）。得益于双视图，软更新技术中可以实现延迟持久化，并且可以消除崩溃后的文件系统检查。在此前的软更新技术工作中[52-56]，软更新技术基于内存中的缓存实现了双视图，如用于缓存数据的页缓存（Page Cache）和用于元数据的 inode 缓存与目录项缓存。

然而，当使用非易失性内存作为存储设备时，直接应用原有的双视图设计并不能完全发挥非易失性内存的性能优势。举例来说，原有的双视图设计依赖于内存中维护的页缓存。当使用页缓存技术时，为了处理一个文件的写请求，文件系统先将数据写入到页缓存中，再异步地将数据从页缓存中写入到存储设备。由于非易失性内存的访问时延与普通内存的访问时延相近，且非易失性内存可以以字节为粒度寻址访问，当使用非易失性内存作为存储设备时，将数据先写入页缓存再写入非易失性内存，不如将数据直接写入到非易失性内存之中。前者将数据以内存速度写入两次，而后者只需将数据写入一次，避免的不必要的数据拷贝。同样地，在处理文件的读请求时，可以直接将数据从非易失性内存中读取给用户，也不需要使用页缓存。因此，大多数非易失性内存文件系统[34, 36-37] 出于性能考虑，会避免使用页缓存，直接在非易失性内存上进行数据操作。

在非易失性内存上使用缓存会影响文件系统的性能，但同时，软更新技术依赖于缓存来实现双视图。为了解决这两者之间的矛盾，高效地实现双视图，SoupFS 提出了基于指针的双视图设计。在基于指针的双视图设计中，并非每个结构在两个视图中均有副本。两个视图中大部分结构是共享的，通过结构上的不同指针来区分其在不同视图中的位置。接下来，我们将结合表 2–2 中列出的 SoupFS 中的数据结构，分别介绍它们在基于指针的双视图中的使用情况。

inode 结构中的指针双视图 由于 Linux 内核的 VFS 需要使用其自己的数据结构（VFS inode）来表示一个 inode 结构，在不改变 Linux 的 VFS 的架构的情况下，inode 结构是不可避免会重复的。既然如此，SoupFS 直接复用 Linux 的 VFS 结构用于

表 2–2 SoupFS 中各个数据结构在指针双视图中的表示

数据结构	最新视图	一致视图
inode	VFS 中的 inode 结构	SoupFS 定义的 inode 结构
文件名	共用：SoupFS 定义的文件名结构	
目录项	共用：SoupFS 定义的目录项（通过不同指针区分）	
哈希表（桶数组）	内存中的哈希桶数组	非易失性内存上的哈希桶数组
基数树节点	共用：SoupFS 定义的基数树节点	
数据块	共用：SoupFS 的数据块	
分配器	内存中的分配器空闲链表	非易失性内存上的页/缓存行分配结构

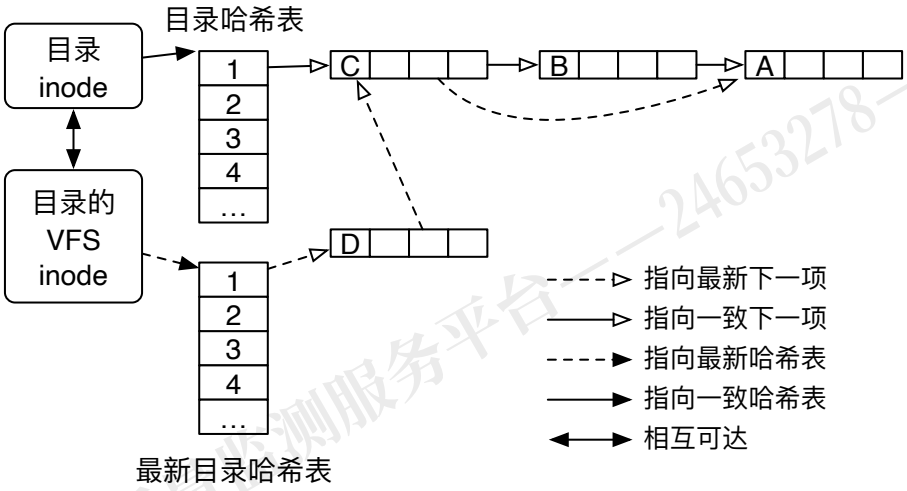


图 2–18 SoupFS 中的指针双视图结构示例

表示最新的元数据，同时，SoupFS 使用其自定义的 inode 结构保存一致的元数据。如图 2–18 中的目录 inode（即 SoupFS 定义的 inode 结构）与其对应的 VFS inode，两种 inode 结构一一对应，可以相互引用。当需要访问最新视图时，可以直接访问 VFS inode 中的元数据；当需要访问一致视图时，可以通过 VFS inode 找到对应的 SoupFS 的 inode，再访问其中的一致元数据。

文件名结构中的指针双视图 在 SoupFS 的设计中，文件名一旦初始化完毕，是不可改变的。同时，一个文件名总是和它的目录项共同存在。在对应的目录项被删除之前，这种绑定关系不会被改变。因此两个视图中的文件名的使用方式，取决于其所对应的目录项的使用。

目录项结构中的指针双视图 SoupFS 中的目录项也很少被修改。在插入过程中，目录项被插入到相应哈希桶链表的头部。此过程不会修改现有的目录项，因此不需为此设计两种结构。当删除一个目录项时，它的前序节点会被修改，指向目录项的后一个节点。这种修改应该反映在 SoupFS 中的双视图中：在最新视图中遍历目录时，应无法看到被删除的目录项；而只要这个删除操作未被持久化，则在一致视图中遍历目录时，应依然能够访问到刚被删除的目录项。否则，当有多个尚未持久化的插入和删除操作时，发生断电可能会使导致目录哈希表中的数据错乱，导致文件系统不一致。

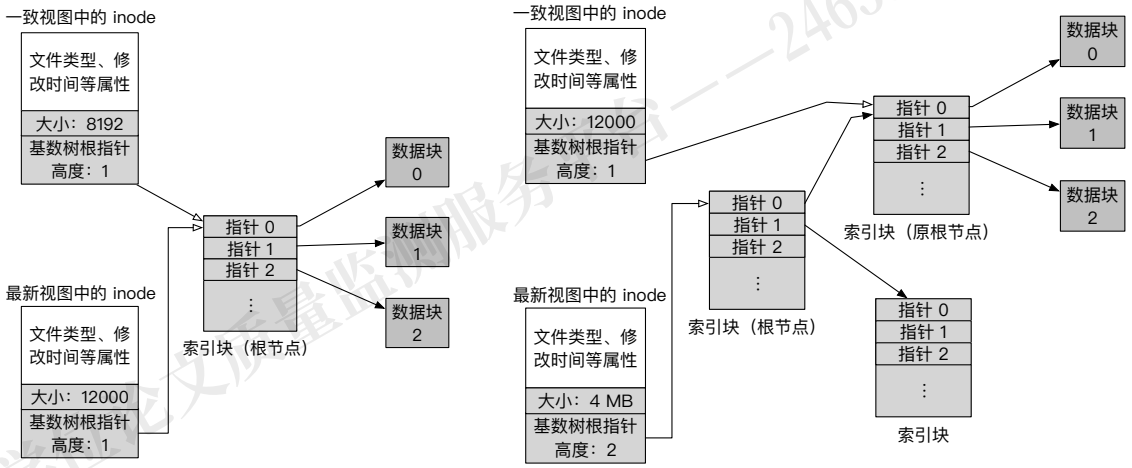
为了在两个视图中共享目录项，SoupFS 在一个目录项中存储了一对下一项指针，即“最新下一项”和“一致下一项”。SoupFS 借助这两个下一项指针，在两个视图中遍历可以获得不同的结果。在最新视图中遍历时，会首先检查最新下一项指针，如果该指针非空，则使用该指针指向的目录项继续遍历。否则，则使用一致下一项指针指向的目录项继续遍历。通过这种方法，可以对目录中最新视图中的目录项进行遍历。而在一致视图中遍历时，只需按照一致下一项指针进行遍历即可。

在图 2–18 中，从目录的 VFS inode 开始，按照虚线箭头可以遍历到最新的视图中的所有目录项，进而找到所有文件；而从目录的 VFS inode 可以找到 SoupFS 中的 inode 结构，并按照实现箭头，可以在一致视图中进行遍历。图 2–18 中，两个视图中看到的内容是不同的。在最新视图中，可以看到目录中保存有文件 D、文件 C、文件 A 三个文件。而一致视图中，可以看到的文件有文件 C、文件 B 和文件 A。这两个视图的区别，是由于有些文件系统请求已经被文件系统处理，但其更新还未被持久化。如在图 2–18 中，文件 D 被创建，而文件 B 被删除，这两个操作只保留在最新视图中，还未被同步到一致视图中（即还未被持久化）。

由于整个目录项被保存在非易失性内存中，那么有可能在发生断电或崩溃时，目录项中还残留了上一次的“最新下一项”指针。为了区分真正的“最新下一项”指针和意外断电后遗留下的“最新下一项”指针，SoupFS 在超级块中维护了一个当前版本号（Epoch Number），并在每个新写入的最新下一项指针中嵌入此版本号。每次挂载时，若 SoupFS 检测到上次使用后未进行正常的卸载操作（即发生了断电和崩溃的情况），SoupFS 会将此超级块中的当前版本号增加一。在访问非空最新下一项指针时，如果其中保存的版本号为旧的版本号，则其是崩溃时遗留下来的，SoupFS 会将其置为空，并当做空指针继续进行后续的遍历。这种访问时的检查可以防止崩溃后的全系统阻塞式（Stop-the-World）检查，允许良性不一致的在线修复。

哈希表桶结构中的指针双视图 哈希表桶存放了目录哈希表中的一个链表头指针。当对目录哈希表进行插入和删除操作时，哈希表的链表头指针可能会被修改。在这种情况下，SoupFS 需要将最新的链表头指针保存在普通内存的一个临时位置中，而非直接更改一致性视图中的链表头，以防止影响一致性视图中的链表使用。为了提供两个视图，SoupFS 为每个哈希桶维护一个“最新”的哈希桶。如果“最新”的哈希桶不为空，则它指向最新视图中目录项链表中的第一个目录项。一个最新的哈希桶和它对应的哈希桶一起使用时，其作用类似于目录项中的两个下一项指针。为了方便内存管理，一个目录哈希表中的所有最新哈希桶被分配在同一个普通内存页中，并按需分配，以降低额外的内存开销。

因此，哈希表在图 2–18 中有两个不同的结构——非易失性内存中保存的 SoupFS 定义的哈希桶数组（图中的“目录哈希表”），和在内存中按需分配的最新哈希桶数组（即图中的“最新目录哈希表”）。



(a) 相同树高下的双视图表示　　(b) 不同树高下的双视图表示

图 2–19 SoupFS 中基数树结构的双视图表示

基数树节点与数据块结构中的指针双视图 SoupFS 并没有专门为基数树节点和数据块设计在两个视图中不同的结构。但是 SoupFS 使用了可变高的基数树结构，该结构本身就可以表现出两个不同的情况。具体来说，由于文件内数据的偏移量范围为 0 到文件当前大小（不包括文件当前大小），一个足够大的基数树结构可以同时表示两个不同大小的文件。图 2–19(a) 展示了一个文件在增大过程中，其 inode 结构和基数树结构的双视图。在此例子中，该文件原本的大小为 8,192 字节，在基数树中需要使用两个数据块（图中的数据块 `0` 和数据块 `1`）进行保存，这些

数据和元数据均持久地保存在非易失性内存中。当用户对该文件进行写入操作后，文件的大小变为 12,000 字节。但是由于 SoupFS 异步进行持久化，此时的文件写入和大小修改只在最新视图中进行，并未同步到一致视图中。于是，在图 2–19(a) 中可以看到，同一个文件在两个视图中的 inode 结构中的信息有所不同，但两个视图中的 inode 结构均指向相同的一棵基数树。在一致视图中的 inode 中记录了该文件的大小为 8,196 字节，而在最新视图中的 inode 结构记录了该文件的大小为 12,000 字节。被共享的基数树共索引了三个数据块，可以最多保存 12 KB 的数据，可以同时满足该 inode 在两个视图中的访问需求：在一致视图的 inode 结构中可以访问索引中的前两个数据块（即文件偏移量从 0 到 8,191）；在最新视图的 inode 结构中可以访问索引中的前三个数据块中文件偏移量从 0 到 11,999 的数据。

图 2–19(b) 中显示了另外一个例子。在此例子中，文件的写入除了改变了文件大小，还导致了基数树高度的增加。在这种情况下，两个视图中的 inode 结构依然共用了同一棵基数树。其中一致视图中的 inode 结构指向的树根节点是这个基数树的一个子树，且恰好保存了整个基数树的前一部分数据块。

通过共享基数树的方式，SoupFS 可以利用双视图中 inode 结构，满足两个视图中的数据访问需求。另外，与软更新技术相同，SoupFS 侧重于对元数据一致性的保证，并不保证写入数据的原子性。因此，在 SoupFS 中，一次写入操作所写入的数据并不保证被原子地持久化。如果需要保证写入数据的原子性，可以在 SoupFS 的基础上采用页粒度的写时复制技术对文件数据进行修改。

分配器中的指针双视图 在章节 2.2.2 中，我们提到 SoupFS 在普通内存中使用 CPU 本地的空闲列表来提升文件系统的空间管理性能。这些普通内存中的空闲列表用来表示非易失性内存中最新的空间使用情况，而在非易失性内存上保存的分配信息（页分配信息和缓存行分配信息）则表示在一致性视图中非易失性内存的空间使用情况。这两种结构共同组成的 SoupFS 的分配器中的双视图。

2.2.5 依赖追踪

依赖跟踪是软更新技术的关键步骤之一。SoupFS 结合非易失性内存的特点，使得非易失性内存上软更新技术的依赖追踪过程被极大简化。具体来说，由于非易失性内存的可字节寻址的特性，文件系统中的各个结构可以避免伪共享的发生，从而从根本上避免循环依赖关系的发生。因此，我们可以使用有向无环图（Directed Acyclic Graph, DAG）来记录文件系统中不同结构之间的依赖关系。同时，由于在 SoupFS 中不使用页缓存和块设备层的抽象，文件系统可以直接操纵向非易失性

表 2–3 SoupFS 中所追踪的操作和每个操作所对应的信息

操作类型	记录的相关信息和结构指针
diradd	增加的目录项、源目录、被覆盖的 inode
dirrem	删除的目录项、目标目录
sizechg	文件的新大小和旧大小
attrchg	无

内存上的写入，消除了在传统软更新技术中写回线程与文件系统之间的语义鸿沟（Semantic Gap）。换句话说，SoupFS 中的写回线程（Persister）可以知道在哪个结构上的哪个操作需要被持久化。通过让写回线程感知文件系统语义，SoupFS 可以进一步简化依赖跟踪和保证。

依赖跟踪结构 传统的软更新技术以字节为粒度的记录更新的依赖关系，但由于块设备的接口，其必须使用额外的结构来记录不同块之间的依赖关系。与传统的软更新技术不同，SoupFS 采取了一种以 inode 为核心的带语义的依赖关系追踪方法。

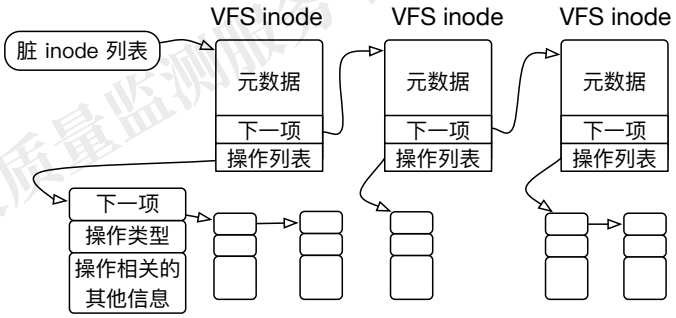


图 2–20 SoupFS 中的依赖追踪结构

图 2–20 展示了 SoupFS 中的依赖跟踪结构。每个 CPU 本地维护了一个脏 inode 列表，其中记录了一组内容被修改过，但还未被持久化的 VFS inode 结构（即脏 inode）。每个 VFS inode 结构中，记录了一个操作列表，其中记录了在此 inode 结构上所有修改但还未持久化的操作。每个操作记录由一个操作类型和该操作相关的其他信息组成，如该操作中所涉及到的结构的指针。

表 2–3 列出了 SoupFS 在依赖追踪过程中所记录的具体操作和信息。其中 diradd 和 dirrem 用于追踪目录内的目录项操作，sizechg 用于表示常规文件的大小变化，attrchg 用于记录对 inode 结构中的各种属性的更新。在系统调用过

程中，SoupFS 会根据进行的操作，创建一个新的操作节点，加入到对应的 VFS inode 的操作列表中，具体我们将在后文介绍系统调用的实现时进行介绍。而若一个 VFS inode 的操作列表不为空，则该 VFS inode 会被添加到脏 inode 列表之中。写回线程（Persister）会在后台根据脏 inode 列表的内容进行持久化处理。

SoupFS 所记录的这些依赖比传统软更新技术更加简单，但这些信息已经足够 SoupFS 获知操作的依赖顺序，并在依赖保证时进行按需持久化。假设一个目录的 VFS inode 中包含一个 `diradd` 操作，通过检查一并记录的目录项（即增加的目录项），SoupFS 可以得知该插入操作的所有细节，例如插入文件的文件名（可以从记录的目录项中获知）和新的 inode（同样可以从记录的目录项中获知）。写回线程可以通过记录的 `diradd` 操作，得知这些信息，并按照正确的顺序进行这些操作的持久化。

系统调用的实现 SoupFS 将 POSIX 中文件系统相关的主要系统调用分为以下几类进行相应的处理。

- **只修改属性的系统调用**。一些系统调用只会修改一个 inode 的部分属性，例如 `chmod` 和 `chown` 两个系统调用。SoupFS 对这些系统调用的处理方式是直接更新相应 VFS inode 中的属性，并在其操作列表中插入一个新 `attrchg` 类型的操作记录。
- **涉及单个目录内修改的系统调用**。`create`、`mkdir`、`unlink`、`rmdir` 等系统调用会修改父目录中的内容。对于这些系统调用，SoupFS 将按照此前章节 2.2.3 中所描述的步骤进行处理。此后，SoupFS 会在父目录 inode 的操作列表中插入一个 `diradd` 或 `dirrem` 类型的操作记录。新创建或删除的目录项同样会被记录在操作记录中。
- **修改文件数据的系统调用**。`write`、`truncate` 等系统调用会修改常规文件的基数树结构。如此前章节 2.2.3 中所述，对于扩大文件的系统调用，SoupFS 直接分配基数树中间节点和数据块，并将其加入到基数树之中。对于缩小文件的系统调用，SoupFS 延迟树节点和数据块的释放。SoupFS 在 VFS inode 中记录新的文件大小和新的基数树根指针和高度，并在 inode 对应的操作列表中记录一个 `sizechg` 类型的操作记录。
- **重命名**。重命名（`rename`）可能会涉及到多个目录，因此需要特殊处理。与传统软更新技术相同，SoupFS 将重命名视为“在目标目录中创建”和“在源目录中删除”两个操作的组合。因此，SoupFS 在目标目录 inode 和源目录 inode 的操作列表中分别插入 `diradd` 和 `dirrem` 两个操作记录。SoupFS 采用了传统软更新技术中对重命名操作的顺序要求：首先持久化创建操作，

再持久化删除操作。为了跟踪这种依赖关系，源目录和目标目录分别记录在 diradd 和 dirrem 两个记录中。

如果在重命名的过程中，目标目录中现有目录项会被覆盖，则可直接修改该目录项中的 inode 指针。同时，为了释放目录项中被覆盖的 inode 结构，SoupFS 将其作为被覆盖的 inode 一并记录在 diradd 操作记录中。

2.2.6 依赖保证

SoupFS 的依赖保证和修改的持久化操作由写回线程（Persister）完成。SoupFS 在挂载时，会创建多个写回线程，并定期将其唤醒。每当被唤醒时，写回线程会从每个 CPU 的脏 inode 列表中遍历所有的脏 inode，并按照顺序要求将操作列表中的每个操作进行持久化。写回线程的数量和写回线程的唤醒频率可以在挂载时指定①。

在 SoupFS 中持久化一个操作非常简单。对于 diradd 操作，写回线程首先要保证新结构的分配信息以及新结构中内容的持久化。之后，其通过访问最新的指针，将一致指针进行相应的更新，并使用内存刷除指令保障持久化的完成。此操作之后，一致视图中的信息与最新视图中的信息回归一致。

对于 dirrem 操作，写回线程首先将目标目录项从一致视图中持久化删除。在保证删除操作持久化完毕之后，写回线程释放被删除的数据结构所占用的空间。

对于 sizechg 操作，写回线程通过检查新、旧文件大小，得知是否有新分配的基数树节点和数据块。如果有，写回线程会对分配信息和新分配的结构中的内容进行持久化。在基数树结构上的所有修改都被持久化后，写回线程会更新文件大小、一致视图中的基数树树根和高度。如果是截断（truncate）等让文件缩小的操作，在新的文件大小、基数树树根和高度被持久化之后，被截断的基数树节点和数据块可以进行释放。

对于 attrchg 操作，写回线程会将 VFS inode 中的属性等信息原子地、持久地写入 SoupFS 在非易失性内存上对应的 inode 结构。操作的原子性将在章节 2.2.7 中进一步讨论。

每当写回线程完成一个操作的持久化，写回线程会将对应的操作记录从操作列表中移除。当一个 VFS inode 的操作列表为空时，写回线程会将其从脏 inode 列表中移除。

① 为了保证 NUMA 架构下文件系统的访问效率，SoupFS 默认会保证每个 NUMA 节点上都有写回线程。

2.2.7 原子性保证

SoupFS 在进行数据写入和持久化时，假设非易失性内存的原子写入单位为一个缓存行，即 64 个字节。基于这个假设，SoupFS 中使用了两种原子写方法。

- 原子指令更新是 SoupFS 中最常用的一种原子更新方法，通过使用 CPU 提供的原子更新指令来完成。SoupFS 在非易失性内存上的指针修改，均通过这种方法进行。
- 缓存行原子更新是另一种原子写入方法，主要在持久化 inode 结构时使用。在 SoupFS 中，一个 inode 结构被设计成一个缓存行大小（即 64 字节）。为了保证整个 inode 的更新是原子进行的，SoupFS 需要在整个 inode 结构完全被更新前防止其所在缓存行被逐出 CPU 缓存。有多种方法可以实现这种效果：
  1. 最简单和高效的方法是使用 Intel 处理器中的受限事务性内存（Restricted Transactional Memory，RTM）技术。该技术可以保证在开始事务之后，所有向内存中写入的数据，都会暂时被保留在 CPU 缓存中，直到事务结束。这种方法利用了硬件提供的机制，方法比较高效，但需要处理器支持受限事务性内存硬件特性。
  2. 第二种方法是使用日志保证缓存行的原子更新。这种方法需要在文件系统的存储空间中预留部分区域，用于记录细粒度的日志。由于只需要考虑一个缓存行大小的内存原子写入，日志区域只需要一个缓存行即可。同时，为了防止共同运行的多个线程争抢使用同一个日志区域，可以使用 CPU 本地的日志区域——在存储空间中为每个 CPU 都预留一个日志区域。当一个线程需要进行缓存行粒度的原子更新时，其获取当前的 CPU 所对应的日志区域，并使用该日志区域进行日志记录。为了防止在此期间线程被迁移到其他的 CPU，在此操作前应关闭抢占和中断等机制，并在原子更新完成之后恢复抢占和中断的原有状态。若由于断电等情况造成文件系统未正常卸载，则再此后挂载时，可通过读取每个 CPU 对应的日志区域进行缓存行原子操作的恢复（前滚或回滚），以保存原子性。由于机器中 CPU 的数量一般是一个较小的常量，日志区域的恢复操作所需的时间是确定性的，且在非易失性内存这种快速的存储设备上，恢复操作进行的非常快。使用细粒度日志的方式保证原子更新，需要承受日志造成的两次写入问题（一次在日志区域中，另一此在数据真正应写入的位置），但不需要使用特殊的硬件机制，较为通用。但是由于日志区域固定，所有的缓存行原子更新（比如 inode 结构写回非易失

性内存）都需要首先写入到这段区域之内。由于非易失性内存依然有磨损问题，这种设计会导致此区域被过于频繁写入，影响非易失性内存的使用寿命。

3. 第三种方法，是使用双生结构（Twin Structure）保证缓存行粒度的原子更新。以 inode 结构为例：原有的 SoupFS inode 结构占用一个缓存行；在使用双生结构之后，每个 SoupFS inode 结构占用相邻的两个缓存行。为了便于叙述，我们称第一个缓存行中的 inode 结构为 inode0，第二个缓存行中的 inode 结构为 inode1 。在第一个缓存行中，使用一个比特作为标记位（Flag）。当标记位为 0 时，则使用 inode0 中的信息；当标记位为 1 时，则使用 inode1 中的信息。当进行更新时，会首先根据当前的标记位，将 inode 内容更新在未被使用的结构中。在更新完毕之后，通过原子修改标记位，切换所使用的结构。这样新的 inode 内容就被原子地应用在文件系统之中。这两个 inode 结构格式相同，交替使用，互为日志空间，因此本文称之为双生结构。这种双生结构在使用时，不会造成非常严重的磨损不均匀问题，但由于双生结构需要使用两倍的存储空间，因此对空间的利用率有一定的影响。

综合考虑三种保证缓存行粒度原子更新的方法，当硬件支持时，使用受限事务性内存是最优的方案；当其不可用时，可优先使用双生结构；当空间利用率非常重要时，可使用细粒度日志。我们在 SoupFS 中优先使用了受限事务性内存的方案。

此处需要注意的是，上述的原子写入方法只能保证更新的原子性。想要保证原子持久化，依然需要配合 `CLFLUSH` 等缓存刷除指令共同使用。

2.2.8 文件系统检查

SoupFS 在文件系统检查和恢复方面与传统的软更新相同。由于双视图中一致视图中的结构永远都是一致的，SoupFS 可以在崩溃重启之后立即使用，无需等待文件系统检查和恢复的完成。

不过，为了收集因为系统意外崩溃所导致的内存泄露，用户或管理员可以定期手动调用一个专门的文件系统检查程序（fsck）。该程序将从 SoupFS 文件系统的根部开始，依次遍历整个文件系统，将所遍历到的结构与文件系统中所记录的分配信息进行对比，以确保没有被 SoupFS 所使用的内存空间在分配器中处于空闲状态，可以被未来的操作所使用。

2.2.9 写耐久度讨论

虽然 SoupFS 并不以优化非易失性内存的写耐久度为设计目标，但 SoupFS 在原理上比现有的文件系统在写耐久度上对非易失性内存更加友好。这主要是由于 SoupFS 使用软更新技术保证崩溃一致性，避免了需要在非易失性内存上频繁写入临时数据的日志等机制。在 SoupFS 中，几乎所有对非易失性内存的写入都是为了修改文件系统的持久化状态。唯一的例外是对目录项中最新下一项指针的更新。虽然 SoupFS 可以进一步将此指针保存在普通内存中，以减少对非易失性内存的写入，提高续航能力。但这需要使用额外的数据结构来跟踪每个目录项与其最新下一项指针之间的对应关系，会让 SoupFS 更加复杂，造成额外的性能开销。同时，更新最新下一项指针的操作对非易失性内存写耐久度的负面影响是有限的。这是因为在实际实现中，对此指针的修改只发生在目录项删除操作中，且在操作被写回线程持久化之前，这个指针修改很有可能一直被缓存在 CPU 缓存中，不会产生对非易失性内存的真实写入。

2.3 评测效果与分析

在本章节中，我们将使用一系列文件系统性能测试展示 SoupFS 和其他文件系统在非易失性内存上的性能。测试将包括对文件创建等操作的微基准测试，和使用 Filebench 和 Postmark 等测试集进行的综合测试。

2.3.1 测试平台与环境

我们使用一台英特尔服务器作为测试平台。该测试平台具有两个英特尔 ® 至强 ® E5 处理器，每个 CPU 有 8 个核心、4 个 DDR4 内存通道。为了保证性能的稳定，CPU 的频率被固定为 2.3 GHz。平台上配有共 48 GB 的 DRAM 普通内存和 64 GB NVDIMM 非易失性内存，平均地分在两个 NUMA 节点之上。在测试中，我们使用单个 NUMA 节点上的 32 GB NVDIMM 作为非易失性内存设备，用于文件系统的测试。

为了表现 SoupFS 与现有文件系统的性能区别，我们同时测试了 Ext4、Ext4-DAX （Ext4 的 DAX 模式）、PMFS 和 NOVA 文件系统作为对比。其中 Ext4 和其 DAX 模式为 Linux 4.9.6 版本内核自带的版本，可直接在 Linux 4.9.6 内核中使用；PMFS 和 NOVA 两个文件系统则在少量修改之后，可以成功地运行在 Linux 4.9.6 内核之上。PMFS、NOVA 和 SoupFS 三个文件系统直接从内核中非易失性内存的驱动程序中获得一块连续的非易失性内存区域，并对此非易失性内存直接进行管理。Ext4-DAX 则使用非易失性内存的驱动程序提供的 DAX 接口访问非易失

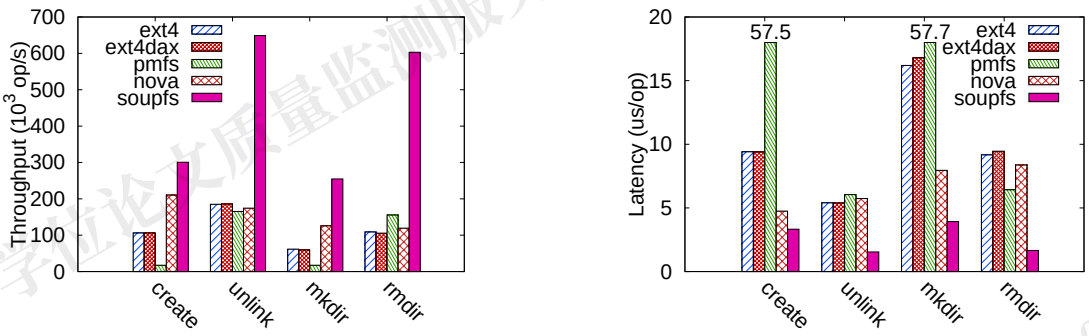
表 2–4 两个微基准测试程序

微基准测试名称	负载内容
`filetest`	(I) create ($10^4 \times 100$) (II) unlink ($10^4 \times 100$)
`dirtest`	(I) mkdir ($10^4 \times 100$) (II) rmdir ($10^4 \times 100$)

性内存，在文件系统中文件的读、写操作中可以绕过页缓存，以提升性能。由于在 Linux 内核中没有文件系统支持传统软更新技术，且在 FreeBSD 等系统中的 UFS 等文件系统无法直接支持非易失性内存，我们在测试中并未与传统的软更新技术进行对比。

2.3.2 微基准测试

为了直接表现 SoupFS 在非易失性内存上的性能优势，我们使用两个单线程的微基准测试程序评估 SoupFS 的吞吐量和时延。表 2–4 中给出了两个微基准测试程序的负载内容。微基准测试程序 `filetest` 会在一个目录中创建 10^4 个常规文件，之后将它们全部删除。这个过程会循环进行 100 次。微基准测试程序 `dirtest` 与 `filetest` 类似，但其创建的是目录，而非常规文件。



(a) 吞吐量　　(b) 时延

图 2–21 在微基准测试中 SoupFS 与其他文件系统的测试结果

图 2–21 给出了使用两个微基准测试程序测试时，不同文件系统的 `create`、`unlink`、`mkdir` 和 `rmdir` 四个操作的吞吐量和时延。SoupFS 在所有的这些测试中均取得了最好的性能表现。在图 2–21(a) 中，SoupFS 的吞吐量比其他文件系统高 43% 到 405%；在图 2–21(b) 所表现的时延中，SoupFS 的时延表现比其他文件系统低 30% 到 80%。这些性能提升主要来源于 SoupFS 使用的软更新技术，使得

SoupFS 可以异步地进行持久化，从而减少了文件系统操作的关键路径中的缓存刷除指令。

NOVA 在测试中也表现出了比较好的性能，这主要是由于其在普通内存中使用基数树来管理每个目录的结构。然而，在修改过程中，NOVA 依然需要使用日志和缓存刷除指令来保证崩溃一致性，因而性能表现与 SoupFS 相比有所差距。

PMFS 的测试结果呈现出了高时延和低吞吐的情况。除了有日志和缓存刷除指令的问题之外，PMFS 的性能结果还与其使用线性的目录组织方式有关。由于线性的目录组织，随着目录中文件数量增长，PMFS 在目录中查找文件的时间也随之增长。在 `create` 和 `mkdir` 两个操作中，需要进行目录项插入操作。PMFS 需要线性扫描现有的目录项，以找到一个可用的目录项空洞，并将新的目录项写入空洞之中。在我们的微基准测试程序中，连续的文件创建操作导致 PMFS 在寻找空洞时需要扫描完所有已有目录项才能在目录的末尾找到空洞。因而线性扫描导致 PMFS 在此两个操作中的性能表现很差。而在 `unlink` 和 `rmdir` 两个操作中，PMFS 的性能表现比较好。这是由于我们的微基准测试创建文件和删除文件的顺序是相同的，而 PMFS 在删除目录项时，会尝试将相邻的目录项空洞进行合并，因此在我们的微基准测试程序中，PMFS 一般只需要扫描两个目录项即可找到目标目录项，从而继续进行删除操作。这种情况使得 PMFS 在进行删除操作时时延较低（图 2–21(b)）。

Ext4 和 Ext4-DAX 两个文件系统使用哈希树（H-Tree）组织目录中的目录项，因此这两种文件系统表现出的性能比 PMFS 要好。

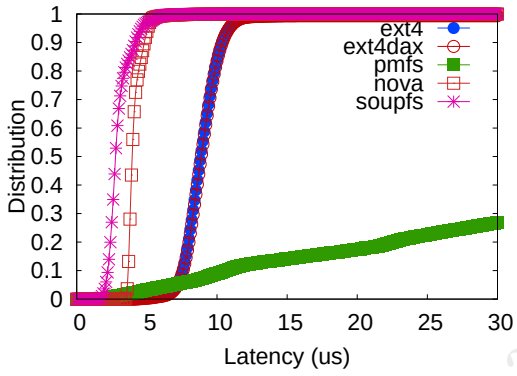


图 2–22 SoupFS 在微基准测试中时延的分布

图 2–22 中给出了 `filetest` 中文件创建的时延分布情况。图中只给出了在 0 到 30 微秒之间的时延分布情况，在此区间中已经可以看出各个文件系统的性能差距。图中的数据再次印证了图 2–21(b) 中给出的平均时延信息。SoupFS 中的大多

数时延落在 3 微秒附近，而 NOVA 的时延大多数在 4 微秒左右。由于线性目录组织在测试中的效率低下，PMFS 的时延数据分布均匀，且稳定升高，符合在加入目录项时，线性扫描操作随文件数增多而变慢的解释。

2.3.3 综合基准测试

在进行了微基准测试之后，我们使用综合基准测试集 Filebench[63-64] 和 Postmark[65]，来评估 SoupFS 对实际应用程序的性能影响。

表 2–5 综合基准测试中所使用的各 Filebench 工作负载的特性

负载名称	平均文件大小	文件数量	I/O 大小	读写比例
FileServer	128 KB	10,000	1 MB	1:2
FileServer-1K	1 KB	10,000	1 MB	1:2
WebServer	16 KB	10,000	16 KB	10:1
WebProxy	16 KB	10,000	16 KB	5:1
Varmail	16 KB	5,000	1 MB	1:1

Filebench Filebench 是一个非常常用的文件系统性能基准测试集，其可以通过指定不同的参数，来模拟各种文件系统上的工作负载。为了使用 Filebench 对非易失性内存文件系统进行测试，我们对 Filebench 程序进行了修改，消除了其中的部分性能瓶颈，并使用更精确的计时单位。表 2–5 中给出了在本测试中使用到的 Filebench 的工作负载特性。对于每个负载，我们分别使用 1 到 20 个线程进行测试，并以折线形式画出平均吞吐量，以同时展现文件系统的可伸缩性（Scalability）。同时，由于我们发现 NUMA 架构对文件系统的性能稳定性造成影响，对于每个负载，我们同时给出“直接运行”与“将线程绑定在 NUMA 节点 0 上运行”两种情况下的测试结果。为了避免覆盖折线，所有测试的文件系统图例统一在图 2–26(a) 给出。

FileServer 是 Filebench 中用于模拟文件服务器的一个工作负载。在 FileServer 负载中，文件的平均大小为 128 KB。对比此后在一些工作负载测试，可以看出 FileServer 负载的吞吐量比其他测试更低。这是由于每个操作需要写入更多的数据，因此每秒钟能执行的操作相对较少。

SoupFS 在 FileServer 工作负载中吞吐量比 NOVA、PMFS 等其他文件系统的吞吐量高一些（图 2–23）。这主要是由于 SoupFS 中使用双视图表示文件，因此文件基数树结构的修改不需要同步地写入到非易失性内存中。这种异步写入的特性

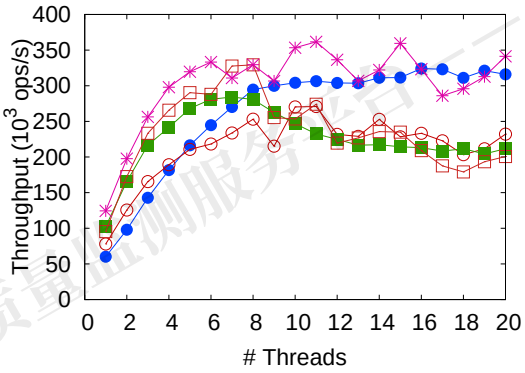
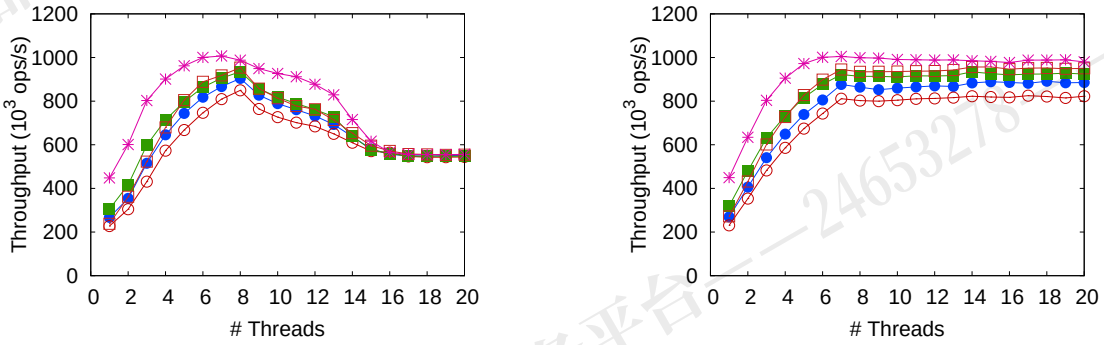


图 2–23 SoupFS 与其他文件系统在 FileServer 负载上的吞吐量对比



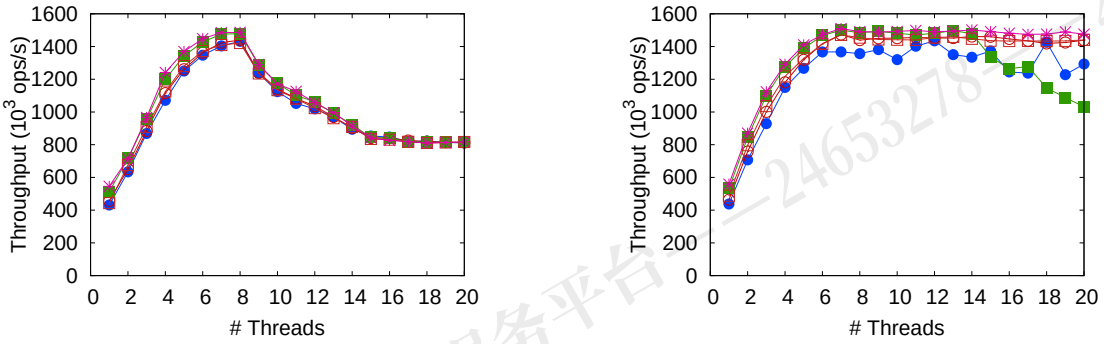
(a) FileServer-1K 负载上的吞吐量　　(b) 绑定 NUMA 节点后的吞吐量

图 2–24 SoupFS 与其他文件系统在 FileServer-1K 负载上的吞吐量对比

提升了文件系统的性能。需要注意的是，这种异步写入机制同时也导致文件写入操作（和文件数据）在系统调用返回之后不一定会持久化。应用程序需要额外使用 `fsync` 系统调用进一步保证写入操作和写入数据的持久化。SoupFS 所提供的这种保证与传统的文件系统是一样的，应用程序可以自行决定其写入的文件数据是否需要立即持久化。因此，通过延迟文件数据的持久化，应用程序有机会获得更高的文件操作性能。而 PMFS、NOVA 等非易失性内存文件系统在 `write` 文件写入系统调用完成后强制了文件数据的持久性，虽然其提供了更强的保证，但同时也剥夺了应用程序对数据持久性的控制，在某些情况下会影响应用程序性能。

为了进一步突显元数据操作在负载中的性能差距，我们还测试了另外一个负载：FileServer-1K，即将原 FileServer 中的文件平均大小修改为 1 KB，其余参数不变。图 2–24(a) 中给出了测试结果，可以看出，所有文件系统的吞吐量都显著增加，同时 SoupFS 的性能优势更加明显。相比 PMFS 和 NOVA，SoupFS 的吞吐量分别提高了 89% 和 47%。

此外，我们还可以看出，大多数文件系统的吞吐量在多于 8 个线程之后发生下降的情况。这是由于 NUMA 架构导致的。由于测试平台上每个 CPU 有 8 个物理线程，当测试线程超过 8 个线程时，操作系统会将线程在不同 NUMA 节点上进行调度。由于测试所使用的非易失性内存均在同一个 NUMA 节点之上，这种调度会导致许多跨 NUMA 节点的内存访问。由于跨 NUMA 节点的内存访问比同 NUMA 节点上的访问慢许多，调度会造成性能下降。由于 SoupFS 使用了后台线程进行持久化，因此其性能会在 8 个线程之前就产生了波动和下降。在将所有测试线程绑定在非易失性内存所在的 NUMA 节点之后，跨 NUMA 的访问的情况消失。因此在图 2–24(b) 中，所有文件系统在 8 个线程时达到最高吞吐量；在更多线程时性能也不会下降。

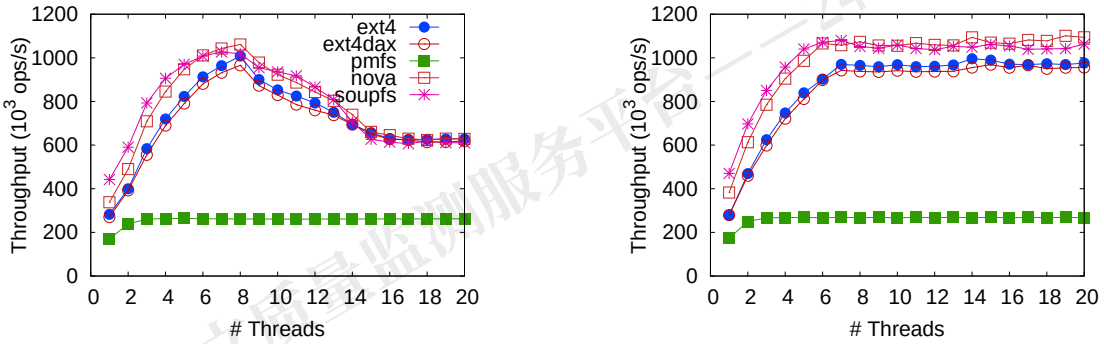


(a) WebServer 负载上的吞吐量　　(b) 绑定 NUMA 节点后的吞吐量

图 2–25 SoupFS 与其他文件系统在 WebServer 负载上的吞吐量对比

Filebench 中的 WebServer 工作负载模拟了一个网页服务器。该工作负载中，线程会模拟网页服务器处理用户的请求：首先访问（读取）网页服务器上保存的多个文件，最后追加信息到日志文件中。由于在 WebServer 工作负载中读取操作占大多数，且测试中并无文件创建和删除操作，只有少量的文件元数据修改，因而，图 2–25(a) 中不同的文件系统吞吐量差距较小。不过 SoupFS 依然以微小的优势表现出最高的吞吐量，NOVA 则紧随其后，优于其他文件系统。Ext4 的两个文件系统的性能也非常不错，主要是由于其使用了页缓存，数据缓存在普通内存中。同时，对比图 2–25(a) 和图 2–25(b)，可以看出在绑定 NUMA 节点之后，文件系统的性能下降消失，说明跨 NUMA 的远程内存访问对 WebServer 负载的影响也较大。

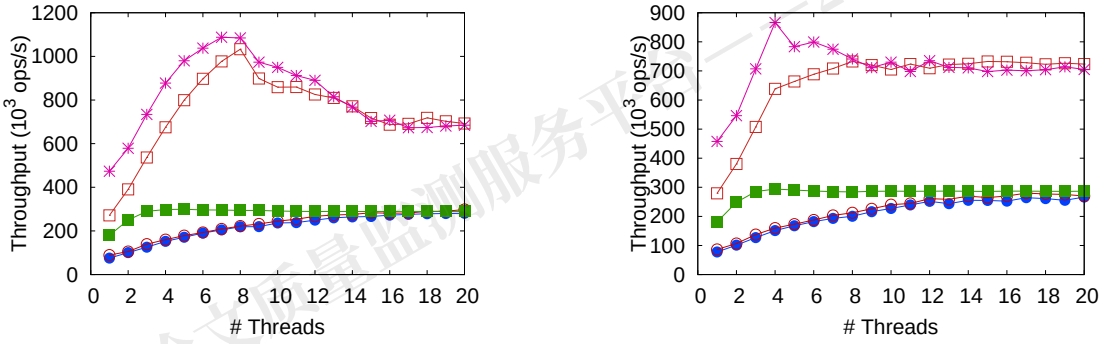
WebProxy 工作负载模拟的是网页代理服务器的行为。每个测试线程在具有大量文件的目录中重复创建和读取多个文件。由于的 PMFS 对目录的线性管理，其在 WebProxy 中的吞吐量最差（图 2–26）；而 SoupFS 使用哈希表、NOVA 使用基数树、Ext4 和 Ext4-DAX 两个文件系统使用哈希树，吞吐量明显更高。由于 WebProxy



(a) WebProxy 负载上的吞吐量　　(b) 绑定 NUMA 节点后的吞吐量

图 2–26 SoupFS 与其他文件系统在 WebProxy 负载上的吞吐量对比

中有频繁的文件删除和创建等元数据操作，SoupFS 性能优势得以发挥，尤其在线程数较少时，优势更加明显。在 WebProxy 负载上，绑定 NUMA 对不同文件系统的性能同样有较大的影响。



(a) Varmail 负载上的吞吐量　　(b) 绑定 NUMA 节点后的吞吐量

图 2–27 SoupFS 与其他文件系统在 Varmail 负载上的吞吐量对比

Varmail 模拟了邮件服务器对文件系统的访问，包括用户读取邮件、删除邮件、回复和发送邮件。值得提出的是，在 Varmail 负责中有大量的 `fsync` 操作。这会强制文件数据写回到非易失性内存之中，因此会消除 Ext4 文件系统使用页缓存所带来的优势。PMFS 的性能依然被其线性目录设计所限制，因此在图 2–27 中可以看出 Ext4 和 Ext4-DAX 性能最差，其次是 PMFS。NOVA 和 SoupFS 直接写入数据到非易失性内存之中，同时目录管理的性能较高，因此表现出相对较高的吞吐量。由于元数据操作的优势，SoupFS 的吞吐量比 NOVA 更高，最多高出 75%。

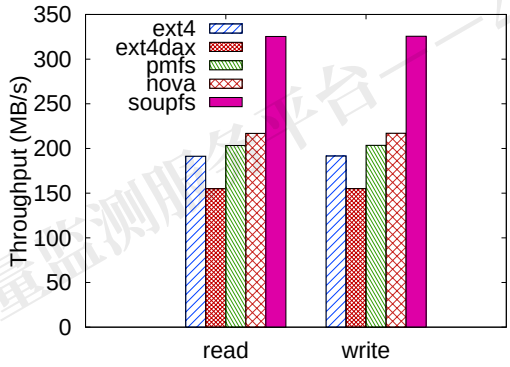
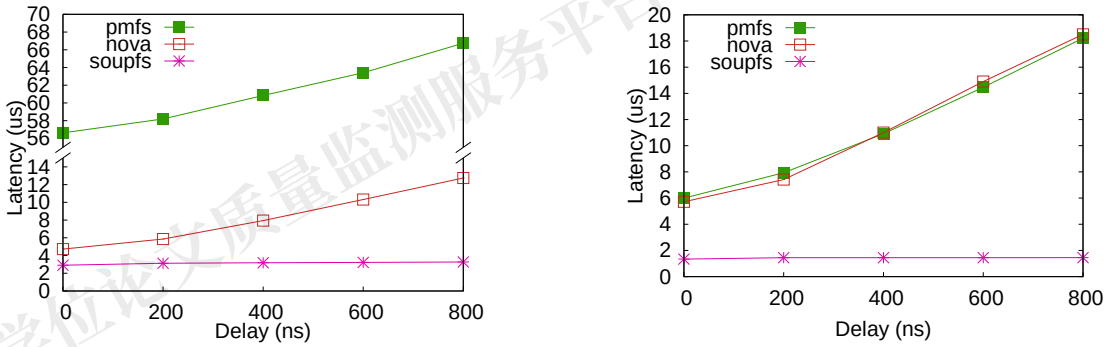


图 2–28 在 Postmark 测试集中各个文件系统的性能

Postmark Postmark 同样是模拟邮件服务器的一个测试集。我们将测试中的操作数量扩大到 10^6，以更好地表现不同文件系统的性能。图 2–28 中给出了单线程 Postmark 测试的性能，可以看出 SoupFS 的读写性能最多超出其他文件系统 50%。

2.3.4 敏感性测试



(a) `create` 的操作时延　　(b) `unlink` 的操作时延

图 2–29 在 `filetest` 测试中不同缓存刷除延迟下的操作时延

不同的非易失性内存技术具有不同的写入时延。我们测试平台上使用的 NVDIMM 非易失性内存具有与普通内存相同的性能；然而，使用 Phase-Change Memory 和 3D-XPoint 等介质所构建的非易失性内存将具有比普通内存更高的访问时延，尤其是写入时延。因此，我们通过在每个 `CLFLUSH` 指令之后插入不同的延迟来评估对不同文件系统对非易失性内存写入时延的敏感性。

图 2–29 中给出了在 `CLFLUSH` 后插入不同延迟之后，`filetest` 微基准测试中的创建（`create`）和删除（`unlink`）的时延。由于 SoupFS 将缓存刷除指令从

关键路径中消除，SoupFS 的操作时延并不随着延迟而增加。而 PMFS 和 NOVA 两个文件系统操作时延，随着增加延迟的增长而变高。这是由于这两个文件系统需要在系统调用返回之前，使用缓存刷除指令保证崩溃一致性和修改的持久化。具体来说，当增加的延迟从 0 增加到 800 纳秒，NOVA 的 `create` 操作时延增加了 8 微秒，提高了 200%（图 2–29(a)）。对于 PMFS 来说，虽然其时延增加了，但 PMFS 的创建时延主要是由于低效的线性目录查找导致的，因此相对影响并不显著。对于图 2–29(b) 中的删除操作，NOVA 和 PMFS 都受到 `CLFLUSH` 延迟的影响，操作时延从 6 微秒增加到 18 微秒。

2.4 其他相关工作

元数据更新方式相关的研究 除了本工作中使用的软更新技术之外，文件系统中还有许多其他保持元数据更新一致性的方法，包括影子页（Shadow Paging）[66-68]、日志结构（Log-structuring）[69-75]、日志机制[76-79] 和原子写入[80]。此外，研究者还使用了多种方法表示数据写入的顺序，如使用反向指针（Backpointers）[81]、事务性校验（Transactional Checksums）[82]、修改补丁（Patch）[83]。举例来说，NoFS[81] 提出使用反向指针的方法，降低持久化数据时的写入顺序要求。然而这种方法需要在元数据中额外存储反向指针，增大了存储的开销；同时，这种方法假设反向指针与其元数据是可以原子持久化的，而这在非易失性内存中往往并不成立，因此无法直接在非易失性内存系统中使用。

非易失性内存存储系统 一些存储工作提出利用非易失性内存加快元数据更新的性能。例如，WAFL[66] 利用非易失性内存保存日志信息以提升同步日志写入的性能。使用不间断电源（Uninterruptible Power Supply，UPS）供电的 Rio 缓存[84] 同样可以用于提供高性能的元数据日志功能。然而，即使将非易失性内存作为缓存使用，这些系统依然需要考虑 CPU 缓存打乱元数据更新的持久化顺序的问题。

Wu 和 Zwaenepoel 提出了 eNVy[85] 存储系统，使用页地址翻译（Page Translation）技术将闪存直接映射到内存总线地址中进行使用。为了克服闪存写入性能慢的问题，eVNy 使用了一小块电池供电的 SRAM 作为一个缓冲区，用于存放被更新的页面，制造出一种原位更新（In-place Update）的效果。由于现在的非易失性内存技术已经与普通内存的性能相近，SCMFS[86] 和 SIMFS[87] 等工作直接将文件数据映射到用户态进程的地址空间中进行使用。这些技术与 SoupFS 的设计是正交的。

非易失性内存上的数据结构 非易失性内存与普通内存相同的访问方式，引发了大量非易失性内存上的数据结构研究[88-95]。Venkataraman 等人[88] 基于非易失性内存设计了一个持久化 B+ 树，但在每次更新时都使用同步的缓存行刷除指令保证持久化顺序。NV-Tree[89] 则使用普通内存作为索引，减少同步缓存行刷除的开销，但在崩溃之后，其需要通过扫描整个非易失性内存区域将普通内存中的索引重建出来。Mnemosyne[90] 为数据结构提供了一套可以保证一致性更新的事务接口，方便应用程序在非易失性内存上设计和使用数据结构。SoupFS 在文件系统操作的关键路径中消除了缓存刷除，且不需使用日志机制保证崩溃一致性。

崩溃一致性和内存持久化模型 Chidambaram 等人[78] 提出将修改的持久性与持久顺序分离，并通过一个假想的“异步持久化通知”（Asynchronous Durability Notification，ADN）硬件机制提出了乐观崩溃一致性。如果异步持久化通知机制可以应用在非易失性内存之中，允许硬件报告写入的持久化操作完毕信息，则 SoupFS 的实现可以因此进一步简化，并变得更加高效。

Foedus[96] 利用了易失内存页与非易失性内存上的分层的快照页组成双页（Dual Pages），在非易失性内存数据库中提供快照功能和崩溃一致性的保证。为了配合双页的使用，Foedus 中的大部分指针均是存储在一起的“双指针”（Dual Pointers），一个指向易失内存页，另一个指向非易失性内存上的持久页。在目录项等结构中，SoupFS 使用了与“双指针”类似的机制来表示文件系统中元数据的双视图。然而在 SoupFS 中，最新视图中的最新指针是按需分配的，且可能与一致指针分开存储。

Pelley 等人[97] 提出非易失性内存的持久化一致性与普通内存的并发一致性是相似的。基于此，他们总结了一组非易失性内存的持久化模型，如严格持久性（Strict Persistency）、Epoch 持久性（Epoch Persistency），并额外提出了 Strand 持久性（Strand Persistency）。Kolli 等人[98] 在不同持久化模型之上进一步提出了多种技术，用于放松已知读写集合的事务中写的顺序要求，从而提高事务的执行性能。与这些工作不同，SoupFS 使用了软更新技术，而非日志方法来保证持久化一致性模型。

2.5 本章小结

大多数单机文件系统均作为操作系统内核的一部分或其可装卸模块运行在内核态空间之中。大量对非易失性内存的文件系统研究也从内核态文件系统开始。由于非易失性内存的高性能，现有的内核态非易失性内存文件系统使用同步更新

的机制，虽然能够更早地保证文件系统修改的持久性，但却无法避免在文件系统操作中的缓存刷除等耗时指令出现在关键路径中，导致文件系统操作时延的增长。

在本章中，我们介绍了新型内核态异步非易失性内存文件系统 SoupFS 的设计与实现。通过对软更新技术的回顾和分析，我们发现软更新技术与非易失性内存的特性的十分契合：软更新技术允许文件系统对更新进行异步持久化，从而允许非易失性内存文件系统将缓存刷除等耗时的指令从关键路径中移除，缩短文件系统操作的时延。在另一方面，因非易失性内存的高性能和可字节寻址的特性，符合了软更新技术对更新依赖关系的细粒度追踪，可以极大地降低传统软更新技术中复杂的依赖关系和复杂的循环依赖处理逻辑，让软更新技术的实现更加简单高效。通过在非易失性内存上实现软更新技术，我们提出的 SoupFS 显著降低了非易失性内存文件系统操作的时延，提供了更高的吞吐量。同时，SoupFS 保留了软更新技术中的特性，在异常卸载之后不需要等待耗时的文件系统检查即可直接挂载使用。

第三章　新型用户态非易失性内存文件系统框架设计

虽然内核态非易失性内存文件系统可以针对非易失性内存的特性进行特殊设计，以尽量避免文件系统的实现成为性能瓶颈，但内核态文件系统依然无法完全发挥非易失性内存的性能优势。首先，跨模式的访问会影响应用程序对非易失性内存文件系统的访问性能。应用程序和文件系统分别处于用户态和内核态。在使用文件系统功能时，应用程序需要通过系统调用将请求传递给内核态的文件系统进行处理，这导致系统调用和上下文切换出现在文件操作的关键路径之上，影响非易失性内存文件系统的访问性能。其次，虚拟文件系统中的复杂逻辑容易成为非易失性内存文件系统的性能瓶颈。为了使用标准的系统调用接口对应用程序提供文件服务，内核态文件系统的实现应遵循虚拟文件系统的规范。虚拟文件系统是针对传统文件系统而设计实现的，其中的诸多设计和机制在使用非易失性内存时已经不再必要或不再最优。内核态非易失性内存文件系统可以通过在实现中不使用某些机制（比如页缓存）来避免其成为性能瓶颈，而另外一些机制（如虚拟文件系统中的 inode 抽象和路径解析等）则与虚拟文件系统的功能强耦合，难以由具体的文件系统进行实现或避免。因而导致内核态非易失性内存文件系统无法充分发挥非易失性内存的性能。

与此同时，由于非易失性内存以字节粒度访问的特性，其在被映射到用户态内存空间中之后，可以在用户态被直接访问。因此，系统研究人员开始考虑在用户态设计和实现高性能非易失性内存文件系统。

在本章中，我们提出一种新的用户态非易失性内存文件系统设计框架 Treasury。在 Treasury 中，用户态非易失性内存文件系统对一定范围内的文件系统数据和元数据具有完全控制权，是真正意义上的用户态文件系统。图 3–1 展示出了 Treasury 在应用和内核架构中的位置和基本架构。Treasury 主要分为用户态部分和内核态部分。在内核态的 kernFS 用于保障文件系统的安全性和完整性，并将非易失性内存映射到用户态；其余逻辑则交给用户态的 FSLibs 中的具体用户态文件系统进行处理。通过劫持传统文件系统的系统调用请求，并在 FSLibs 中进行处理，Treasury 可以支持传统的应用程序。新型的应用程序也可以通过显式调用 FSLibs 中的函数对其中的用户态非易失性内存文件系统进行使用。

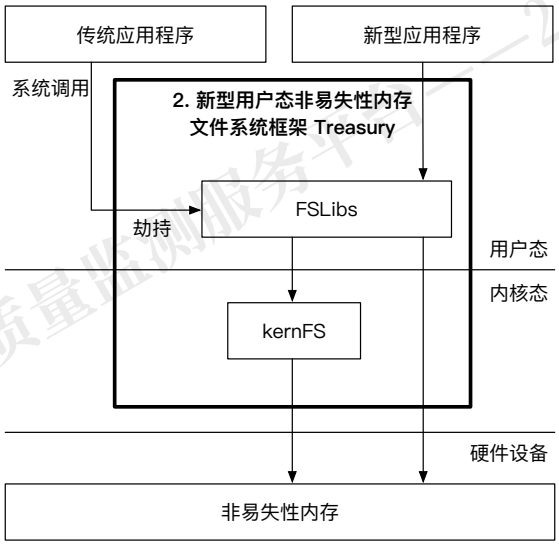


图 3–1 新型用户态非易失性内存文件系统框架 Treasury 在应用和内核架构中的位置

3.1 研究背景与动机

3.1.1 用户态非易失性内存文件系统的不足

由于非易失性内存可以以字节粒度访问，近来的研究逐渐开始关注在用户态对非易失性内存进行管理和访问，实现用户态非易失性内存文件系统。

在章节 1.2.2 中，我们已经对用户态非易失性内存进行了简单的介绍。此处我们进行简略回顾，并对现有用户态非易失性内存文件系统的不足进行分析。

Aerie[38] 是最早的用户态非易失性内存文件系统框架之一。其最主要的特点是灵活性，其将非易失性内存暴露给用户态空间的应用程序，允许用户态的应用程序在不与内核交互的情况下直接访问文件数据。具体来说，Aerie 中的文件系统库可以读取存储在非易失性内存中的元数据来查找文件，然后可以直接读取和修改文件数据。然而为了保证文件系统的安全性，当文件系统库想要更新文件系统的元数据时，其需要通过进程间通讯（Inter-process Communication，IPC）向中心化可信文件系统服务发送请求。然而进程间通讯的开销较大，这使得 Aerie 中的元数据操作性能较差。

Strata[39] 是一个跨介质的文件系统，为多层、混合存储介质设计。Strata 依赖于非易失性内存的可字节寻址特性，允许应用程序在用户态空间处理文件系统的数据和元数据请求。对于更新请求，Strata 使用一个单独的非易失性内存设备作为日志设备，并将所有更新以操作日志（Operation Log）形式记录在非易失性内存设备之上。这部分操作可以直接在用户态空间完成。Strata 的内核部分的文件系统，会定期读取操作日志，并将日志内容重做，保存到下层设备中。这个操作被称为消

表 3–1 多个进程在共享文件/目录中的操作时延（单位：纳秒）

测试	并行进程数	Strata	NOVA	ZoFS
append	1	1,653	2,172	1,147
	2	34,551	3,882	1,703
create	1	4,195	3,534	2,494
	2	283,972	6,167	3,459

化（Digest）；下层设备可以是非易失性内存设备、固态硬盘或者机械硬盘。Strata 的这种“用户态记录操作日志–内核态消化日志”的方式，会造成两次写入（Double Writes）问题，当多个进程共享同一个文件时，会造成严重的效率问题。

总体来看，现有的用户态非易失性内存文件系统允许应用程序在用户态空间完成大部分的数据、元数据读取操作。这减少了应用程序执行过程中的系统调用和上下文切换，通常来说可以提升应用程序的执行速度。然而，现有的用户态空间文件系统也对用户态文件系统库设置了限制：*禁止用户态空间文件系统直接修改文件系统的元数据*。换句话说，用户态空间文件系统只能*间接更新元数据*：如 Aerie 中必须使用进程间通讯，通过中心化可信文件系统服务进行元数据的修改；而在 Strata 中，只能将元数据修改写入操作日志，由内核文件系统来进行实际的修改。

间接更新元数据保证了非易失性内存文件系统的安全性，却会对文件系统的性能造成影响。为了说明间接更新元数据对性能的影响，我们进行了两个测试：

1. 一个/多个进程不断向共享文件中追加 4 KB 数据（`append`）；
2. 一个/多个进程在共享目录中不断创建文件（`create`）；

我们测量不同测试中操作的平均时延，结果如表 3–1 所示。从表格可以看出，当两个进程共享同一个文件或目录时，Strata 追加和创建操作的时延大大增加。当两个进程共同追加时，追加操作的时延延长了近 20 倍；而创建操作的时间延长了近 70 倍。作为参考，我们还在 NOVA 和后续章节 3.4 中介绍的文件系统 ZoFS 上进行了同样的测试。对于单个进程来说，Strata 的追加性能优于 NOVA，而创建操作则相对较慢。这是因为 Strata 中每个创建操作都有两次写入问题，保证元数据的一致性，但延长了创建操作的时延。

表 3–1 中的测试结果显示间接更新元数据会极大地影响文件系统性能。而在当前的文件系统中，间接更新元数据是保证文件系统安全性的重要条件。那么，是否能够设计一个用户态空间的非易失性内存文件系统，既能让用户态文件系统可以完全控制非易失性内存上的数据和元数据，避免元数据的间接更新，以充分发

表 3–2 数据库和网页服务器中的文件权限分析结果

系统名称	文件类型	权限（八进制）	用户 ID/组 ID	文件数量	总大小
MySQL	目录文件	750	970/970	6	32 KB
	常规文件	640	970/970	358	399 MB
	常规文件	644	0/0	1	0 B
PostgreSQL	目录文件	700	969/969	28	128 KB
	常规文件	600	969/969	1,807	99 MB
DokuWiki	目录文件	755	33/33	1,035	5 MB
	常规文件	644	33/33	19,941	452 MB

挥非易失性内存的性能；同时又能为文件系统提供足够的安全性和隔离性保证呢？

3.1.2 权限和隔离

文件系统使用权限（Permission）来限制应用程序对文件的访问：包括能否读、写、执行等。这种做法可以保护文件系统中存储的数据的安全性。为了保证这些权限起到限制作用，文件系统通常与应用程序处于不同的地址空间中，并通过有限的接口（如系统调用）提供服务。通过赋予用户态空间文件系统对数据和元数据的完全控制，可以完全发挥非易失性内存的性能，但作为代价，我们需要重新考虑如何保证应用程序和文件系统之间的隔离性。为此，我们对应用程序的数据文件的权限进行了调查。

首先，我们调查了 MySQL[99] 和 PostgreSQL[100] 这两个数据库程序，以及一个长期运行的 DokuWiki[101] 网站的数据目录中的文件权限。对于每个数据库系统，我们初始化了一个新的数据目录，并导入了示例数据库（MySQL 的 employee、world 和 sakila[102]；PostgreSQL 的 World、dellstore2 和 Pagila[103]）。我们所调研的 DokuWiki 存储着研究所的基本信息，如项目、成员和发表文章等，因此我们直接使用这些文件进行分析。表 3–2 显示了调研的结果。对于两个数据库系统，常规文件的权限高度集中到 `640` 和 `600`。唯一一个例外是 MySQL 中根用户拥有的一个空的 `debian-5.7.flag` 文件，其权限为 `644`。此文件用来表示数据库二进制格式版本的升级[104]，并不用于处理数据库操作。对于我们的 DokuWiki 网站，所有的常规文件都是 `644` 权限。除了这些权限的静态分布之外，我们还发现，这些数据库文件的权限在数据库处理请求的过程中不会发生改变。而对于 DokuWiki 来说，唯一可能改变文件权限的操作是文件上传时的 `chmod` 系统调用。但是这个 `chmod`

表 3–3 FSL Homes Traces 中的文件统计结果。表格左侧为 Traces 中的文件统计信息，右侧为在分组之后的统计信息

权限	不同类型文件数			文件总数	分组数	组大小（字节）		
	常规文件	符号链接	目录			最小	平均	最大
	648,691	6,486	71,574	726,751	4,449	0	79.7 M	52.0 G
644	538,538	18	65,127	603,683	1,935	0	46.1 M	23.4 G
600	105,226	0	4,021	109,247	1,174	0	222.2 M	52.0 G
666	233	6,468	927	7,628	365	7	474.2 K	106.7 M
444	3,313	0	1,099	4,412	48	660	92.5 M	995.1 M
660	342	0	276	618	15	23.5 K	118.2 K	211.1 K
640	921	0	33	954	853	0	31.9 K	10.5 M
664	110	0	91	201	51	28.7 K	348.2 K	5.4 M
440	8	0	0	8	8	455	26.5 K	98.3 K

操作只有在用户权限与 DokuWiki 的 PHP 权限掩码不匹配时才会触发，在我们的设置中从未发生过这种情况。通过这些调研结果，我们可以得出初步结论：*应用程序所保存的文件大多具有类似的权限，而且这些文件的权限很少被改变*。因此，我们可以考虑将文件分组，将具有相同权限的不同文件分在同一组内，组内的修改允许用户态文件系统直接进行，跨组的访问才进行权限检查。这样既能减少权限检查的上下文切换次数，还能在管理这些文件时给予用户态文件系统更多的权限和灵活性。

为了进一步证实我们的发现，我们还分析了 FSL Homes Traces[105]。FSL Homes Traces 是一个共享网络文件系统的快照集合[106]。其中的内容为学校中不同学生的家目录（Home Directory）中所保存的数据。在调研中，我们选择了于 2015 年 4 月 10 日记录的最新快照，其中包括 15 名学生的家目录。表 3–3 中给出了分析的汇总情况。在这个快照中，共有 726,751 个文件，其中大部分（89%）是常规文件。将不同的文件按照其权限进行区分，可以发现 `644` 是常规文件最常用的权限，其次是 `600`。由于快照中没有给出详细的目录信息，我们假设目录的所有权（文件的拥有用户和拥有组）和权限与我们在快照中被分析的第一个文件相同。同时，我们忽略了文件权限中的执行位，这是因为在将非易失性内存映射到用户空间后，对文件执行需要特殊的手段，此权限位可以在执行时进行限制，而与文件读写权限关系不大。

基于我们此前对文件分组的想法，我们尝试通过以下规则对具有相同权限和所有者的文件进行分组。如果一个文件的权限与它的父目录相同，那么它与它的

父目录被分在同一个组中。否则，创建一个新的组，并将该文件放入新的组中。我们从包含文件系统根目录的单个组开始，自上而下对文件进行分组。最终共形成了 4,449 个组，其中最大的组包含了所有文件的 1/3 左右。同时，我们还统计了各个文件组的大小，并在表 3–3 的右侧给出了每种权限的不同组中最小、平均和最大组的大小。从全局来看，最大的组包含的文件总容量为 52.0 GB，而平均组大小约为 79.7 MB。这说明大多数文件被分在较大的几个组中，因此，如果我们将一个组的控制权完全交给用户态文件系统库，用户态文件系统库可以在内部管理该组中的文件，而不会受到内核上下文切换和权限检查的性能影响。值得注意的是，统计中共有 3,795 个单文件组。但这些组中的文件数之和只占总文件数的 0.6%。

由于 FSL Homes Traces 中只包含了文件系统的静止状态，为了进一步调查应用程序在执行过程中对文件权限的改变，我们又调查了 MobiGen Traces[107-108]。MobiGen Traces 包含了两份在三星 Galaxy 智能手机上收集到的 2 分钟的 I/O 系统调用记录。在对 Facebook 应用程序的系统调用记录中，共有 64,282 次系统调用，其中没有 `chmod` 和 `chown` 等能够修改文件权限和拥有者的系统调用记录。在对 Twitter 应用的记录中，有 25,306 次系统调用。其中有 16 次对 `chmod` 的调用；没有对 `chown` 的调用记录。这 16 次 `chmod` 系统调用以一种固定的模式，有规律地出现：应用程序首先用 `600` 权限创建一个文件，在其中写入新的数据，然后将该文件的权限改成 `660`，最后调用重命名（`rename`）系统调用，覆盖掉同名的目标文件。

根据以上的调研结果和分析，我们可以确认此前的发现：*应用程序所保存的文件大多具有类似的权限，而且这些文件的权限很少被改变*。这样的结果，支持我们将具有相同权限的文件进行分组，并在管理这些文件时赋予用户态文件系统充分的控制权，以减少权限检查等上下文切换次数，提升性能，通过对文件系统的数据和元数据提供足够的保护。

3.1.3 内存保护键

内存保护键（Memory Protection Keys，MPK）[109-111] 是一种新的硬件功能。它允许在页式内存访问的基础上，让用户态空间应用程序对自己的内存访问进行限制。英特尔内存保护键技术是英特尔公司在其处理器芯片上对内存保护键机制的一种实现[110-111]。为了叙述简单，除非特殊注明，后文中的内存保护键均指代英特尔内存保护键机制。内存保护键允许操作系统内核在每个页表项（Page Table Entry）中存储一个 4 比特的区域号，并以此将一个进程的内存虚拟地址空间以页为粒度分为 16 个区域。

用户态应用程序可以访问一个名为 PKRU 的寄存器。这个寄存器上保存了 16 对 2 比特位的权限，对应着 16 个内存区域中每个区域是只读、读写、或不可访问。每个操作系统线程都有自己的 PKRU 寄存器值。每个线程可以通过非特权指令 WRPKRU 对其 PKRU 寄存器进行修改，从而改变本线程对每个内存区域的可访问性。

在设置了内存区域和 PKRU 寄存器之后，每次内存访问时，内存管理模块（Memory Management Unit，MMU）除了查询页表并且检查页表权限之外，还会读取页表项中的区域号，并检查 PKRU 寄存器中相应的权限。如果内存访问违反了权限，将触发页中断，最终可将中断信息传递给用户态应用程序进行处理。内存保护键是对页表项中页权限位的补充，在内存访问过程中，这两种权限都将被检查。

总结来说，内存保护键允许内核将一个进程的内存空间划分为多个区域，并允许用户态的每个线程通过写入非特权限的 PKRU 寄存器来限制其对每个区域的访问。

3.2 系统设计

根据此前的调研，我们可以将文件按照权限分组，相邻且具有相同权限的文件可以分在同一组。如果用户态文件系统库有权限访问这些文件，则可以让用户态文件系统库完全控制和管理这些文件。在本章节中，我们首先介绍一种新的抽象，名为 Coffer，用于包含一组具有相同权限的文件。然后我们使用 Coffer 来设计一种新的非易失性内存文件系统框架 Treasury，并展示用户态文件系统库如何使用 Coffer 接口与内核部分通信。最后，我们将展示如何利用内存保护键等机制进一步加强保护和隔离，以及讨论文件系统中的崩溃恢复问题。

3.2.1 用户态隔离抽象：Coffer

为了表示多个相邻的具有相同权限的文件，我们提出了一种新的抽象，称为 Coffer。Coffer 是一组非易失性内存页的集合，这些内存页具有相同的访问权限，可以用来储存文件。每个 Coffer 结构都拥有一个根页（Root Page），其中保存了该 Coffer 的元数据。根页由内核部分进行管理，通过只读映射允许用户态文件系统库进行访问。

图 3–2 中展示了一个在文件系统层次结构中使用 Coffer 的例子。在这个例子中，除了根页之外，每个 Coffer 还有一个根文件，这是其作为文件系统一部分的“入口”。如果根文件是普通文件、符号链接等非目录文件，那么此 Coffer 只保存该

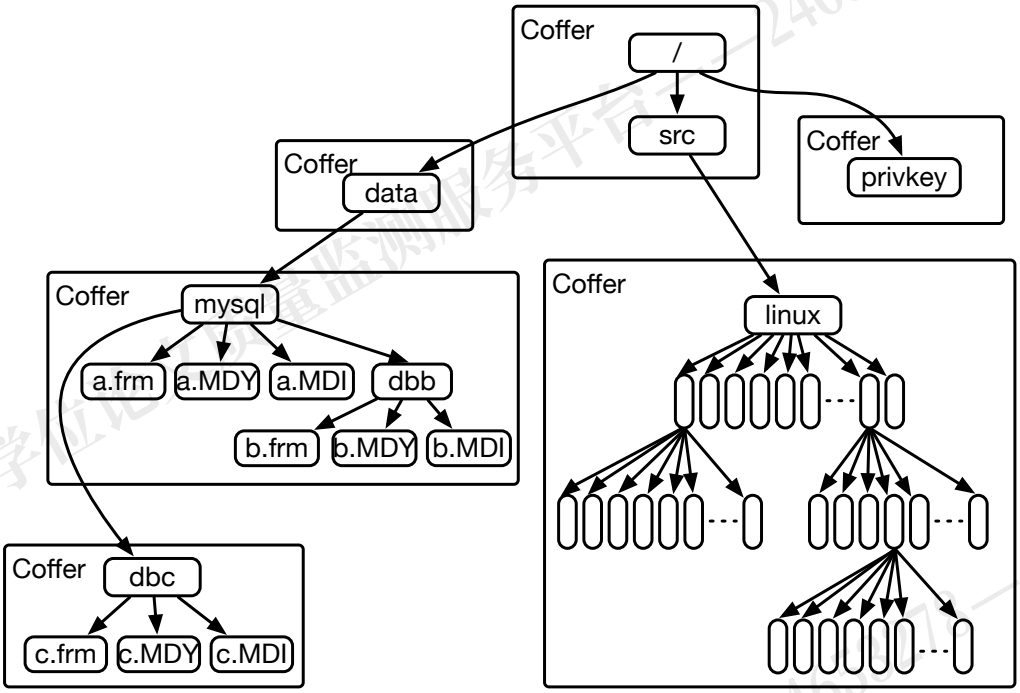


图 3–2 在文件系统中使用 Coffer 抽象的示例。每个 Coffer 包含一个常规文件或者一个目录与其中的部分子孙文件

根文件。如果根文件是一个目录，则与其具有相同权限的子孙文件也可以存储在同一个 Coffer 中，这样它们就可以被用户态文件系统库一起管理，而此过程中不需要内核部分的参与。当子文件与其父目录处于不同的 Coffer 中时，父目录中对应的目录项会记录一条特殊的引用信息，记录在哪个 Coffer 中可以找到目标文件。

Coffer 抽象是允许用户态文件系统库在用户态直接管理非易失性内存文件的关键。由于每个 Coffer 中的文件具有相同的权限，内核部分只需要在用户态文件系统库第一次访问该 Coffer 时进行相应的权限检查即可。一旦权限检查通过，内核会将 Coffer 中所有的非易失性内存页映射到进程中，这样进程中的文件系统库可以直接读取和更新这些页面上的信息。换句话说，通过将非易失性内存空间划分为多个 Coffer，内核可以在 Coffer 粒度上保证文件系统的权限；而当 Coffer 被映射到了用户态，用户态文件系统库可以直接管理 Coffer 中的数据和元数据。

3.2.2 Treasury 架构

基于 Coffer 抽象，我们进一步设计了用户态非易失性内存文件系统框架 Treasury，以充分发挥非易失性内存的性能优势，并提供足够的保护和隔离。

图 3–3 中给出了 Treasury 框架的架构。Treasury 由两部分组成：内核空间的

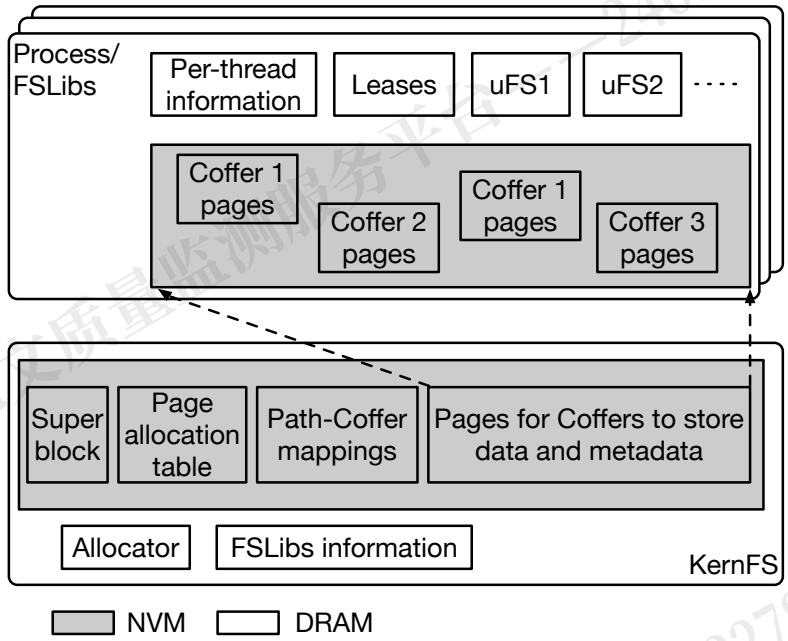


图 3–3 Treasury 的总体架构。Treasury 由一个内核模块（KernFS）和一个用户空间文件系统库集合（FSLibs）组成。内核模块 KernFS 维护着文件系统中的非易失性内存空间的分配信息和路径到 Coffer 的映射。用户空间文件系统库集合 FSLibs 是一个文件系统库的集合外加一些辅助工具。每个具体的文件系统实现称为μFS。一个 Coffer 的内部结构由其对应的μFS 来决定和维护。μFS 和 Coffer 之间的关系类似于一个文件系统管理了一个设备分区

内核模块（KernFS）和用户空间的文件系统库（FSLibs）。

内核模块 KernFS 负责全局空间管理。它使用一个分配信息表来记录非易失性内存上的每个页面是否分配，以及分配给了哪个 Coffer。此外，它还维护了一个路径到 Coffer 的映射表，用来表示每个 Coffer 的根文件在文件系统层次结构中的位置。KernFS 将 Coffer 作为黑盒进行管理，其只关心每个 Coffer 的元数据，如 Coffer 的路径、类型、以及哪些空间属于该 Coffer 等。其不需要知道 Coffer 中具体存储了哪些数据，以及这些数据是如何组织的。前文中提到的“内核部分”即是指 KernFS。

用户态文件系统库集合 FSLibs 包括一组具体用户态文件系统的实现和一些辅助工具。每个用户态文件系统实现称为μFS；辅助工具可以帮助μFS 管理缓冲区内的数据和结构，并与应用程序交互。在 FSLibs 中可以有多个μFS 共同存在，不同的μFS 可以采取不同的方法来组织 Coffer 中的文件数据和元数据。每个μFS 可以通过 Coffer 元数据中的 Coffer 类型来区分不同类型的 Coffer。一个μFS 对一个 Coffer 的管理类似于一个文件系统管理一个设备分区。

表 3–4 KernFS 与 FSLibs 之间的协议。FSLibs 中的μFS 可以通过这些接口发起请求，KernFS 对这些请求进行验证和执行

请求接口	描述
coffer_new	在一个给定的 Coffer 下创建一个新的 Coffer
coffer_delete	删除一个现有 Coffer
coffer_enlarge	为一个 Coffer 分配更多的空间
coffer_shrink	释放 Coffer 中的一些空间
coffer_map	将一个 Coffer 映射到当前进程的内存空间
coffer_unmap	将 Coffer 的映射关系取消
coffer_split	将一个 Coffer 拆成两个 Coffer
coffer_merge	将两个 Coffer 合并成一个
coffer_recover	在一个 Coffer 中进行文件系统检查和数据恢复
fs_mount	注册一个新的文件系统实例
fs_umount	取消一个文件系统实例的注册
file_mmap	内存映射一个文件
file_execve	执行一个（可执行）文件

3.2.3 Coffer 接口

用户态的文件系统库（即μFS）需要与 KernFS 进行配合才能完成某些操作。具体来说，μFS 需要使用 `ioctl` 系统调用向 KernFS 发起一些请求，以对 Coffer 的元数据进行修改。这些请求共分为三类，在表 3–4 中给出。

Coffer 请求 Coffer 请求用于修改 KernFS 中记录的 Coffer 元数据。当一个μFS 需要创建或删除一个 Coffer 时，其通过 `ioctl` 向 KernFS 发起 `coffer_new` 或 `coffer_delete` 请求，KernFS 将检查请求的有效性并进行相应处理。同样的过程也适用于 `coffer_enlarge` 和 `coffer_shrink`。这两个操作分别向 KernFS 批量请求空闲可用的非易失性内存页，以及将空闲的非易失性内存页归还（释放给）KernFS。对于 `coffer_map` 操作，KernFS 会首先检查进程是否有访问该 Coffer 的权限。若检查通过，KernFS 就会将 Coffer 中对应的非易失性内存页映射到进程的内存空间，此后μFS 就可以直接在用户空间访问该 Coffer 的内容。当请求 `coffer_unmap` 或者通过 `setuid` 等系统调用改变进程的用户和组标识符时，映射会被移除。`coffer_split` 操作将一个 Coffer 分割成两个 Coffer，主要在 Coffer 中某些文件的权限被修改时使用。`coffer_merge` 操作则与之相反，它将两个 Coffer

合并成一个 Coffer。`coffer_recover` 操作在检查和恢复一个 Coffer 时使用，将在章节 3.2.5 中进一步讨论。通过发起这些 Coffer 请求，μFS 可以在 KernFS 的监管下对 Coffer 的元数据进行更新。

文件系统请求 FSLibs 使用两个请求向 KernFS 注册或注销自己。在 `fs_mount` 操作中，KernFS 为新的 FSLibs 实例分配相应的结构来跟踪其信息。这些结构在 `fs_umount` 操作中或在进程终止时被释放。

文件请求 在用户态非易失性内存文件系统中，`mmap` 和 `execve` 两个系统调用比较特殊。处理这两个请求既需要了解文件内部结构，又需要较高的权限来修改进程的页表，因此在用户空间中通常无法直接完成这两个系统调用。为此，Treasury 为这两个系统调用特别引入了 `file_mmap` 和 `file_execve` 两个特殊的请求。通过这两个接口，μFS 可以向 KernFS 提供文件数据在非易失性内存上的位置，KernFS 则可以用这些位置信息更新页表，完成 `mmap` 和 `execv` 等系统调用。

3.2.4 隔离和保护

Treasury 将文件系统隔离和权限管理的边界从文件放宽到 Coffer，允许用户态文件系统库（μFS）在映射 Coffer 后直接管理其中的结构和数据。然而，在用户态空间直接管理 Coffer 同样会带来隔离和保护方面的几个问题，我们在这一小节中进行讨论。

3.2.4.1 误写

应用程序很有可能会因软件缺陷（Software Bug）而产生控制流（Control Flow）错乱。在这种情况下，应用程序可能会尝试向随机内存中写入随机数据，从而发生误写（Stray Writes）问题。在非易失性内存出现之前，若应用程序向非法地址（即其无法访问的地址）写入数据，会触发异常，操作系统内核会将出问题的应用程序终止；若应用程序错误地向其能访问的地址写入了错误的数据，则此写入会成功完成。但当应用程序最终被终止之后，其写入内存的错误数据会随内存的回收而变得没有意义，不会造成更严重的问题。

在非易失性内存被引入系统之后，如果依然使用内核态非易失性内存文件系统，则文件系统的关键结构和元数据只在内核的页表中有所映射，因而只有在内核态的文件系统可以修改非易失性内存上的文件系统结构和元数据。应用程序在

用户态产生误写问题无法修改到非易失性内存上的关键文件系统结构和元数据，因此应用程序的缺陷不会造成文件系统的损坏。

然而由于非易失性内存可字节寻址的特性，Treasury 中的用户态非易失性内存文件系统将非易失性内存直接映射到用户态地址空间进行管理，应用程序可以像访问普通内存一样，使用 CPU 的 LOAD、STORE 指令直接访问非易失性内存。在这个情况下，如果因为应用程序中的缺陷发生误写问题，错误写入的错误数据很容易破坏非易失性内存上保存的文件系统数据和结构，从而造成文件系统损坏。更严重的是，若这个文件系统是与其他应用程序共享的，一个出错的应用程序还可能会导致其他正常的应用程序的崩溃。

此前的工作对误写的问题有过讨论，并且提出了一些解决方案。PMFS[34] 主要通过组合使用页表隔离机制和 CR0.WP 位来避免误写问题。在用户空间中，由于页表隔离和特权级提供的隔离，出错的应用程序不会影响到内核内存空间中非易失性内存和使用其他内存空间的其他应用程序①。而在内核空间中，为了访问非易失性内存，非易失性内存文件系统需要将非易失性内存映射到内核的地址中。然而由于内核中不同模块共用页表，内核中的其他模块（如驱动程序）中发出的错误，依然可能因误写问题而破坏在非易失性内存上的关键结构和数据。PMFS 提出并使用了一种基于 CR0.WP 的窗口机制来避免此类情况的发生。CR0.WP 是 CPU 的 CR0 寄存器上的一个特殊比特位。当此比特位为 0 时，当前 CPU 可以在内核态修改在页表中被标记为只读的内存；当此比特位为 1 时，当前 CPU 对于只读内存页无法修改。在使用窗口机制时，PMFS 首先将非易失性内存以只读的权限映射到内核内存空间之中，并将 CR0.WP 比特位设置为 1。当 PMFS 想要对非易失性内存上的结构和数据进行修改时，其临时将 CR0.WP 设置为 0，此后再进行修改，修改完毕后 PMFS 将 CR0.WP 重新设置为 1。在这个过程中，PMFS 打开了一个临时窗口，在此窗口中 CPU 可以修改非易失性内存。而在窗口之外，修改非易失性内存会由于没有写入权限而产生异常中断，从而避免非易失性内存中数据被错误修改。这种机制可以防止内核中驱动等其他模块在触发缺陷后误写非易失性内存上文件系统的关键结构和数据，保护了非易失性内存上文件系统的完整性。

对于 Treasury 来说，避免误写问题要更困难。这主要是因为 Coffer 中的非易失性内存可以在用户态直接修改，且其中包括了文件系统中比较关键的元数据信息。一个比较直接的想法，是使用类似基于 CR0.WP 的窗口机制：将 Coffer 中的非易失性内存映射为只读，只有当μFS 想要修改非易失性内存的内容时，才临时将其变为可写。然而，更改非易失性内存映射的读写权限，涉及到大量的页表权

① 出错的应用程序可能会错误地修改通过 mmap 映射到其内存空间中的非易失性内存文件数据，但由于其本身就有权限修改此文件的内容，故这种出错认为是可以接受的。

限修改，需要在内核中才能实现，这反而需要大量 Treasury 中极力避免的系统调用和上下文切换。同时，这种方法还有一个问题：其无法防止同一进程中其他并发线程中的误写对非易失性内存上的文件系统结构产生破坏。

防止误写问题 在 Treasury 中，我们对 PMFS 中基于 `CR0.WP` 的窗口机制进行了拓展，并引入了内存保护键来保护文件系统免受误写问题的破坏。

当 KernFS 将一个 Coffer 映射到一个进程的内存空间时，KernFS 将该 Coffer 的页面划分入一个不同于进程运行时内存的内存保护键区域。在返回用户空间之前，KernFS 还在线程的 `PKRU` 寄存器中禁用对该 Coffer 内存保护键区域的访问权限。

为了保护非易失性内存不受误写问题的影响，μFS 应与 KernFS 进行配合。具体来说，μFS 应遵循以下规则：

G1. 只有当μFS 访问一个 Coffer 时，该 Coffer 才能被标记为可访问。

当一个μFS 想要访问一个 Coffer 时，它首先通过更新 `PKRU` 寄存器来启用对 Coffer 区域的访问权限。在μFS 完成访问后，访问权限被再次禁用。因此，当应用程序自身代码运行时，没有任何一个 Coffer 是可以访问的；只要当μFS 要访问 Coffer 时，其内存才可以被访问。而由于我们认为μFS 的代码是可信的，所以不会有误写问题对 Coffer 中的内容进行破坏。

Treasury 所使用的这个保护方法非常高效，这主要得益于更新 `PKRU` 寄存器只需要一条 `WRPKRU` 指令，开销很小（在我们的测试平台上大约 16 个 CPU 周期）。同时，由于操作系统会为每个线程维护其寄存器，因此每个线程在μFS 中所打开的访问窗口只能允许其自己访问目标 Coffer，其他并发线程中发生的误写依然不能访问该 Coffer 的内容。这进一步加强了对 Coffer 中文件系统的完整性保护。

3.2.4.2 错误返回和故障隔离

使用基于内存保护键的窗口机制，虽然可以有效地防止应用程序中的软件缺陷破坏 Coffer 中的数据和元数据，但仍可能出现应用程序进行恶意攻击或硬件故障造成的文件系统数据和元数据损坏。文件数据的损坏对运行中的应用程序伤害有限，因为用户空间文件系统库只是返回错误的数据。然而，元数据损坏可能会导致整个应用程序的异常终止，或者导致用户态文件系统库将文件系统的损坏扩散到进程所映射的其他 Coffer 中。对于这种情况，我们再次利用内存保护键机制，将对正常应用程序的干扰降到最低，并将故障限制在一个 Coffer 内，防止其扩散。

优雅的错误返回 当系统调用过程中出现错误时，内核文件系统会返回一个错误代码。应用程序可以根据此错误代码进行不同的处理。然而如果在用户态文件系统访问 Coffer 时，某些 Coffer 结构被破坏，用户态文件系统很可能会因此尝试访问无效的内存区域，进而导致整个进程被段错误异常终止。

一个解决此问题的方法，是在用户态文件系统每次访问内存前，都检查所访问地址的有效性。然而这需要在每次内存访问前都加入检查，在关键路径上引入了过多的地址检查，影响文件系统的性能。

因此，Treasury 采用了另外一种方法：FSLibs 注册段错误的处理函数（Handler），当发生段错误时，FSLibs 在处理函数中检测故障原因，并将其转换为相应的文件系统错误代码，最终返回给应用程序，让应用程序进行处理。具体来说，在 FSLibs 进入文件系统函数之前调用 `sigsetjump`，以保存此时的线程运行状态；在 `SIGSEGV` 信号处理函数（Signal Handler）中，FSLibs 调用 `siglongjump` 跳回到此前保存的状态，这样控制流可以从错误中恢复，并返回相应的文件系统错误。这种方法可以保护应用程序不会因 Coffer 中的无效内存引用而被异常终止，使得 FSLibs 中的错误总是可以转换为文件系统错误，并优雅地返回给应用程序。

故障隔离 为了防止一个 Coffer 中的损坏蔓延到其他 Coffer，KernFS 将不同的 Coffer 映射到不同的内存保护键区域。为此，μFS 的实现还应遵守以下准则：

G2. 在任何时候，每个线程在用户空间中最多只能访问一个 Coffer。

在实现中，μFS 需要追踪记录每个 Coffer 属于哪个内存保护键区域，并在访问前只打开该区域的访问权限。因此，对 Coffer 中数据和元数据的意外访问会被内存保护键机制所阻止，保证故障不会跨 Coffer 传播。需要注意的是，虽然只有一个 Coffer 可以访问，但 FSLibs 可以同时映射多个 Coffer，并通过一条 `WRPKRU` 指令可以快速地切换可访问的 Coffer。

由于在英特尔的内存保护键中，只有最多 15 个内存保护键区域可以使用，一个进程最多可同时映射 15 个 Coffer。在μFS 发出 `coffer_map` 请求之后，如果 KernFS 发现全部 15 个内存保护键区域均已经被使用，则 KernFS 会返回一个特殊的错误代码。μFS 在收到此错误代码后，应首先发出 `coffer_unmap` 请求释放一些内存保护键区域，再映射新的 Coffer。

3.2.4.3 元数据安全保证

在使用内核态文件系统时，恶意应用程序进程只能通过修改与其他应用共享的文件来攻击其他进程。但在 Treasury 中，由于 Coffer 中文件系统的元数据可以

在用户空间直接修改，恶意进程也可以通过操纵共享的 Coffer 中的元数据来攻击其他进程。接下来，我们分情况讨论 Treasury 如何设计机制防止这种攻击。

如果攻击者篡改的元数据不涉及其他 Coffer，那么被攻击者要么会读取到被篡改的错误元数据或数据，要么会由于 Coffer 中结构或元数据的损坏而收到 FSLibs 返回的错误信息。对于前者，由于攻击者原本就有权限修改 Coffer 内的所有文件（攻击者能访问此 Coffer 说明其有权限访问其中的所有文件。），其通过正常的文件系统操作也可以达到同样的效果，所以 Treasury 不需要对此进行特殊考虑。对于后一种情况，被攻击者虽然收到了文件系统的错误信息，可能无法正常工作。但在此过程中，其并没有泄露任何数据，也没有因错误而错误地修改数据。在不考虑拒绝服务（Denial-of-Service，DoS）攻击的情况下，Treasury 也不需要对此进行特殊的保护。

若攻击者篡改的元数据与其他 Coffer 有关，则 Treasury 需要防止恶意元数据的效果扩散到其他 Coffer，以抵御此类元数据攻击。我们以一个例子来介绍 Treasury 如何在这种情况下进行保护。假设两个进程共享同一个 `Coffer A`，恶意进程（攻击者）试图通过操纵 `Coffer A` 中的共享元数据来攻击另一个进程（受害者）。假设受害者遵守此前的规则 *G1*和规则 *G2*，当受害者访问 `Coffer A` 时，通过修改 PKRU 寄存器让 `Coffer A` 可以访问，而其他所有 `Coffer` 无法被访问。因此，如果受害者根据 `Coffer A` 中被操纵的元数据尝试访问 `Coffer A` 之外的 Coffer，就会触发内存保护键的异常，受害者可以得知自己访问的元数据有问题，因而停止对文件的继续访问，避免发送数据泄露等问题。

另一方面，μFS 在访问正常的元数据时，也可能会访问跨 Coffer 引用。在这种情况下，μFS 会主动切换可访问的 Coffer，取消当前 Coffer 的可访问性，并使目标 Coffer 可访问。为了应对攻击者操纵这些合法的跨 Coffer 引用进行攻击，Treasury 对μFS 有最后一条规则，用于强制检查跨 Coffer 引用的完整性：

G3. 对于每一个跨 Coffer 引用，μFS 应该在访问目标 Coffer 之前检查它的有效性。

在此后章节 3.4 中的μFS ZoFS 例子中，目录项是唯一可能包含跨 Coffer 引用的结构——它可能指向另一个 Coffer 的根 inode。在这种情况下，攻击者只有一种情况可能绕过内存保护键机制发起攻击：修改 `Coffer A` 的目录项中的跨 Coffer 引用。然而，当受害者进程中的 ZoFS 访问包含跨 Coffer 引用的目录项（源目录项）时，它将检查源目录项的路径和目标 Coffer 的路径，以确保要访问的 Coffer 确实是预期的 Coffer，而不是被攻击者修改成的其他位置。同时，ZoFS 也会验证其指向的目标确实是目标 Coffer 的根 inode。因此，无论攻击者如何操纵 Coffer 内

的元数据，受害者（被攻击者）都可以在发生错误（泄露数据或改错数据）之前发现问题，并及时防止错误传播到其他 Coffer 中。

Coffer 中的其他元数据也可以被操纵并变得无效，如文件大小可以被修改为负数。因此，如果应用程序以自己的方式直接读取和使用元数据，在使用前，其应该检查元数据的有效性，以防止缓冲区溢出（Buffer Overflow）等攻击。另外需要注意的是，所有的 Coffer 被映射到用户态空间时，都被映射成不可执行状态。因此，在 Coffer 上注入恶意代码对 Treasury 不会产生效果。

3.2.5 检查和恢复

当一个 Coffer 中的结构由于各种原因发生问题是，需要进行 Coffer 检查和恢复操作。检查和恢复操作可以由任何μFS 发起。作为发起者的μFS 调用 `coffer_recover` 来通知 KernFS 开始恢复。KernFS 在需要恢复的 Coffer 根页面中使用租约（Lease）将 Coffer 标记为“正在恢复中”，然后将 Coffer 从发起者以外的所有进程中的映射去除，此后将控制流返回给发起者。发起者在获得控制流后，在 Coffer 内根据其内部的文件结构，开始检查和恢复操作。在检查和恢复之后，发起者将仍在使用中的非易失性内存页地址发送给 KernFS，KernFS 会将这些使用中的内存页与分配给该 Coffer 的内存页进行比较，回收分配了但未被使用的页面。

3.3 具体实现与应用

在本章节中，我们将重点讨论在实现 Treasury 和运行实际应用中的问题和解决方案。

3.3.1 KernFS

Coffer 管理 KernFS 维护着整个文件系统中所有的 Coffer 信息。正如章节 3.2.1 中所定义的那样，每个 Coffer 都有一个根页，存储着 Coffer 的元数据。Treasury 使用根页面的地址（即 `Coffer-ID`）来标识每个 Coffer。Treasury 还引入了一个持久化的哈希表（图 3–3 中的“路径到 Coffer 映射”）来存储所有的 Coffer。哈希表的键是 Coffer 的路径，值是 Coffer 的地址（即 `Coffer-ID`）。使用路径来索引所有的 `Coffer-ID` 可以提升文件的查找速度：当一个μFS 想要访问一个文件时，KernFS 通过比较文件路径和哈希表中存储的路径来找到并映射包含该文件的 Coffer。

空间管理 Treasury 采用两级空间管理。KernFS 将非易失性内存以页为粒度批量分配给 Coffer，每个 Coffer 进一步分配其页面来存储数据和元数据。

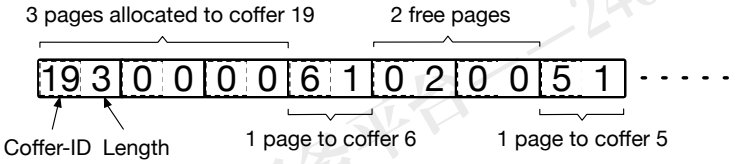


图 3–4 KernFS 中的分配信息表。KernFS 中的分配信息表记录了每个内存页的分配信息。对于每个页，左边的数字代表其是空闲（0）还是分配给某个 Coffer（非 0 的 `Coffer-ID`）。右边的数字表示包括此页在内，有多少连续的页被一同分配

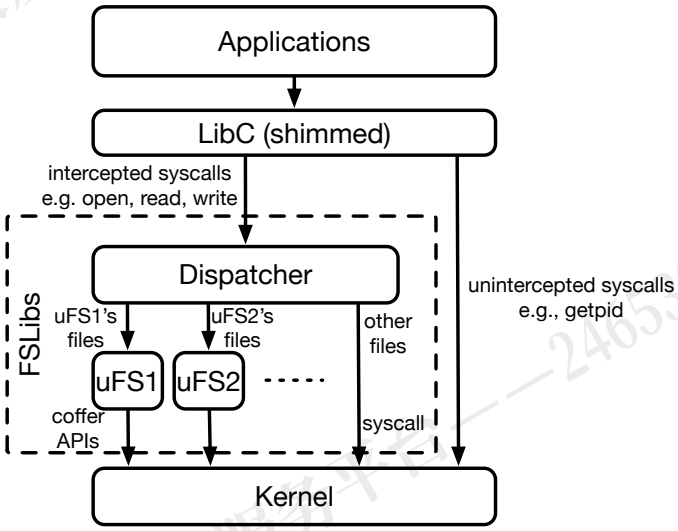


图 3–5 FSLibs 的架构

KernFS 在页粒度上对所有非易失性内存空间进行全局管理。它利用一个持久化的分配表（如图 3–4 所示）来跟踪每个页面的分配状态。在分配表中，每个页的分配状态由一个 32 位的整数（`Coffer-ID`）表示，它记录了内存页被分配到哪个 Coffer。当 `Coffer-ID` 为零时，表示空闲页，即没有被分配给任何 Coffer 的页面。

为了加快大量连续页面的分配速度，具有相同 `Coffer-ID` 的相邻分配信息会被合并。因此，每一页的分配信息还包括另一个 32 位整数，表示从这一个内存页开始，有多少个连续的页面一同被分配到同一个 Coffer 之中。

为了进一步提升分配的性能，我们还使用一些普通内存中的数据结构作为分配器的缓存。例如，我们使用一个在普通内存中的全局红黑树来跟踪分配表中所有的空闲页面；使用另一个红黑树跟踪所有已分配的空间和相应的 `Coffer-ID`。

3.3.2 FSLibs

FSLibs 的作用是允许在用户态非易失性内存文件系统库（即μFS）上透明地运行动态链接的应用程序。通过预加载 FSLibs 链接库，应用程序可以访问由用户态非易失性内存文件系统库管理的文件，且无需进行修改或重新编译。图 3–5 展示了 Treasury 中 FSLibs 的架构。其中包括有一个请求分发器（Dispatcher）和一个或多个μFS。

系统调用拦截 为了能够透明地将动态链接程序直接运行在 Treasury 上的文件系统中，FSLibs 需要拦截所有文件系统相关的系统调用。为此，我们在 glibc 中加入中间层（Shim）函数，当应用程序通过 glibc 调用系统调用时，控制流会首先进入该中间层函数。中间层函数可以决定是将请求通过真正的系统调用交给内核处理，还是将其直接在用户态完成。

在 Treasury 系统中，有两个中间层函数的实现。其中一个是在 glibc 中实现的默认中间层函数。此函数不会进行任何判断，直接调用真正的系统调用。另一个中间层函数由 FSLibs 实现，其通过系统调用的类型、参数等判断是否应该在用户态的μFS 中完成文件系统请求。如果是，则调用对应的μFS 函数；否则，通过真正的系统调用通知内核进行处理。

默认情况下，应用程序使用 glibc 中默认的中间层函数，因此无法通过 Treasury 管理非易失性内存。为了使用 Treasury，我们将 FSLibs 编译成一个名为 `libfs.so` 的动态链接库，并将其路径加入到 `LD_PRELOAD` 环境变量中。在此后启动应用程序时，加载器会首先加载 FSLibs 中的函数（包括中间层函数、请求分发器、一种或多种μFS 的实现）。FSLibs 中的中间层函数会覆盖 glibc 中默认的中间层函数。因此，应用程序调用系统调用时，会进入 FSLibs 的中间层函数，并通过请求分发器，决定是否应该在用户态的某个μFS 中完成请求。

请求分发器 请求分发器根据拦截下来的系统调用参数，判断请求应由哪个μFS 或由内核完成。做出判断的关键是区分由 FSLibs 管理的文件和由内核文件系统存储的文件。在文件系统调用中，有两种参数可以表示要访问的文件。

一种表示方法是使用路径，如在 `open`、`unlink` 等系统调用中，直接以字符串形式的路径表示操作的目标文件。对于绝对路径，我们将其与 FSLibs 的挂载路径进行比较，可以得出此文件是否存储在 FSLibs 管理的文件系统中。具体来说，如果 FSLibs 的挂载路径是目标路径的前缀，则说明此文件需要交给 FSLibs 里面对应的文件系统（μFS）进行管理。对于相对路径，在将其与 FSLibs 的挂载路径

进行比较之前，FSLibs 先将其所维护的当前工作目录的路径拼接在目标路径之前，构成一个绝对路径，再按照绝对路径进行对比。

另外一种表示文件的方法，是使用文件描述符（File Descriptor，FD）。为了对使用文件描述符的文件进行区分，FSLibs 维护了一个用户空间的文件描述符映射表，其在每次使用文件描述符时对该表进行查询和更新。

文件描述符映射表 之前的研究工作（如 Strata[39]）通过设定一个阈值，区分用户空间文件系统库的文件描述符和内核文件系统所使用的文件描述符。例如，在数值上大于 5,000 的文件描述符为用户态文件系统库所使用，其余的由内核文件系统使用。这种方法简单高效，但当应用程序（如 `bash`）依赖于 `dup` 等系统调用时，就会产生问题：`dup` 系统调用的语义是复制一个现有的文件描述符，并以新的文件描述符代表此文件。另外，所返回的新文件描述符应该为当前可用的数值最小的文件描述符，这一点在使用阈值区分不同文件系统的文件时无法满足。

为了解决这个问题，FSLibs 维护了一个用户空间的文件描述符映射表，其中每个应用程序所使用的文件描述符（称为 appFD）被映射到内核所使用的文件描述符（称为 kernFD）或某个μFS 中文件结构。FSLibs 会拦截所有涉及到文件描述符的系统调用，将其中的文件描述符（即 appFD）通过映射表进行翻译，若目标文件是内核的文件描述符 kernFD，则使用此 kernFD 进行系统调用；若目标文件是μFS 中的文件结构，则将请求交给μFS 进行处理。对于 `open` 等返回文件描述符的系统调用，FSLibs 会为其返回的文件描述符（kernFD）分配一个新的 appFD，在映射表中记录此映射关系，并将 appFD 返回给应用程序。通过这种方式，FSLibs 将 kernFD 和μFS 中的文件结构结合起来，对应用程序呈现出统一的 appFD 空间。在进行 `dup` 等系统调用时，可以通过控制分配的 appFD 的数值，保证总是返回应用所能观察到的最小可用的 appFD，从而满足 `dup` 的语义。

使用文件描述符映射表的方法有一个副作用：在执行 `exec`、`clone`、`vfork` 等系统调用时，需要对 FSLibs 所维护的文件描述符映射表进行特殊的操作，并在父子进程之间传递此映射表。FSLibs 采用的一个方法是将文件描述符映射表的内容使用 base64 编码进行序列化，并使用专用的环境变量将其在父子进程之间传递。

符号链接 Treasury 支持符号链接文件。在μFS 中实现符号链接文件的结构比较容易；但在解析路径时考虑符号链接却需要一些技巧。FSLibs 使用了一种简单但可行的方法来处理路径解析过程中的符号链接。路径解析的过程由请求分发器和μFS 共同完成。请求分发器，首先根据当前的绝对路径，将请求交给某个μFS；μFS 会根据路径一层层地访问目录，如果在过程中遇到了一个符号链接文件，μFS

会读取符号链接文件中保存的路径，并用其修正当前正在解析的路径。修正后的新路径会再交还给请求分发器。请求分发器根据新的路径，继续进行分发。这种方法虽然需要在请求分发器和μFS 之间多次交互，且需要进行多次字符串拼接等操作，但可以处理大部分符号链接情况。

3.3.3 限制和讨论

为了让应用程序不需修改和重新编译就能直接使用到 Treasury 中的文件系统，我们试图让 Treasury 的功能和接口与内核文件系统保持一致。然而在实际实现中，存在一些限制导致 Treasury 与内核文件系统的行为有所区别。本章节将对 Treasury 中对这些限制的取舍进行讨论。

权限 Treasury 使用页表中的访问权限位来限制一个进程是否可以读或写一个 Coffer 中的内容。然而，每个页表项中只有一个比特位表示该页是只读还是读写。这与文件系统中传统的 `rwx` 权限有所区别。例如，使用文件系统中的权限可以实现某个文件只能写入，不能去读，这使用页表是无法进行限制的。又如，对于文件的执行权限，Treasury 总是将 Coffer 中的页映射为不可执行；文件的具体执行权限由μFS 在用户态维护，在执行文件时，也是需要μFS 与内核共同完成。Treasury 中的μFS 可以支持更加复杂的权限机制，如 POSIX 访问控制列表（Access Control List，ACL），然而这些权限是在用户态维护的，无法得到内核的强保护。

目录权限 在文件系统的权限中，目录的权限与普通文件的权限有不同的含义。为了系统的简单和高效，Treasury 忽略了这种差异，将目录的权限归为不可访问、只读和读写三种，与普通文件的权限保持一致。

访问时间 由于只读 Coffer 的访问时间（`atime`）元数据不能在用户空间更新，所以 Treasury 的实现中不支持对只读文件的 `atime` 的维护。对于可读可写的文件来说，不受此限制。

拒绝服务攻击 在 Treasury 中，FSLibs 被赋予了对已映射的 Coffer 内的数据和元数据的完全控制权，这使得 FSLibs 可能会受到应用程序的拒绝服务（Denial-of-Service，DoS）攻击，如应用程序可以故意持有租约锁，让其他应用程序无法访问共享资源。

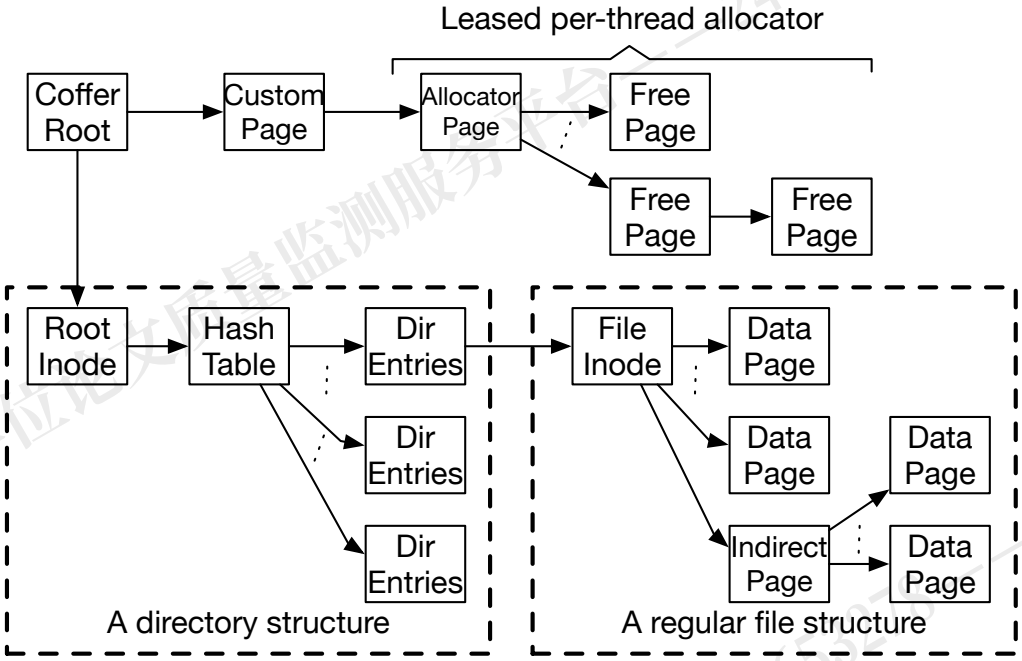


图 3–6 ZoFS 的设计概览。图中每个方框表示一个内存页，Coffer 的根页是在 Treasury 架构中必须的，其余页则可根据 ZoFS 的需要进行组织

其他使用内存保护键机制的应用 Treasury 的安全性依赖内存保护键机制。如果应用程序也需要使用内存保护键，应用程序和 Treasury 将在同一个虚拟内存地址空间中竞争使用有限的内存保护键区域。

3.4 示例文件系统 ZoFS

在 Treasury 框架中，KernFS 只在 Coffer 粒度上对整个文件系统进行管理，而 Coffer 内部如何组织文件数据和元数据，由具体的μFS 实现来决定。在本章节中，我们将介绍一个名为 ZoFS 的μFS 实现，以展示μFS 如何管理 Coffer 并使用 FSLibs 与 KernFS 进行交互。

正如在章节 3.2.1 中所描述的那样，ZoFS 以树状层次化结构来管理文件（图 3–2）。ZoFS 采用的分组规则是：当一个文件与其父目录具有相同权限时，其保存在其父目录所在的 Coffer 之中。这只是 ZoFS 的设计规则，其他μFS 可以选择不同的方法进行分组。例如，一个μFS 可以使用一个扁平化的结构来组织一个目录中的所有子孙文件。又或者，一个μFS 可以只按权限将其管理的文件划分到不同的 Coffer 中，而忽略文件之间的层次关系。

图 3–6 显示了 ZoFS 中的 Coffer 架构。KernFS 在创建 Coffer 时，会默认分配三

个内存页。第一个内存页是 Coffer 的根页，它是整个 Coffer 的入口，包含了 Coffer 的路径、类型等元数据。第二个内存页用于保存根文件的 inode 结构。第三个内存页用于保存μFS 自定义的数据。ZoFS 可以利用此自定义页面，保存其自己定义的 Coffer 中的元数据，如 ZoFS 中的分配信息。后两个内存页的地址存储在 Coffer 的根页面中，ZoFS 通过其中的地址访问相应的数据。

3.4.1 数据和元数据组织

为了简化空间管理，ZoFS 中只支持 4 KB（内存页）大小的空间分配，因此所有 ZoFS 中的结构均是 4 KB 对齐的。这样的空间管理会造成一定的内碎片问题，但可以通过类似将文件数据嵌入到 inode 中等方法进行缓解。我们将其留作未来的工作之一。

目录 ZoFS 使用自适应的两级哈希表来组织目录项。一级哈希表有 512 个指针，每个指针指向一个二级页面。二级页面由两部分组成：页面的前半部分存储了一些目录项，后半部分存储了一个具有 256 个桶的二级哈希表。二级哈希表的每个桶都存储了一个包含目录项的页面链接列表。ZoFS 会尝试先把新的目录项直接保存在二级页面中，只有当二级页面中所有的目录项位置均被占用，新的目录项才会被插入二级哈希表。目录结构中的内存页是按需分配的，可以减少不必要的存储开销。

ZoFS 的每个目录项中包括：一个用于快速对比文件名的哈希值、文件名本身、`Coffer-ID` 和 inode 指针。当 `Coffer-ID` 为 0 时，说明目标 inode 与当前目录处于同一个 Coffer 之中；而当 `Coffer-ID` 不为 0 时，说明该 inode 保存在另外一个 Coffer 之中（即 `Coffer-ID` 所代表的 Coffer）。在访问目标 inode 时，ZoFS 首先将对应的 Coffer 映射到当前进程的内存空间之中，再切换内存保护键，将对应的 Coffer 区域变为可访问，再访问其中的 inode 结构。

常规文件 ZoFS 管理常规文件的方式与 Ext2 等文件系统类似。文件 inode 中保存了指向数据页、索引页和间接索引页的指针。这种设计只是处于实现简单考虑，使用 B+ 树、基数树等更复杂的结构也是可以的。

其他特殊类型的文件。 ZoFS 中一个 inode 结构占据了一整个 4 KB 内存页，因此在 inode 中有足够大的空间来存储特殊文件所需保存的数据。例如，符号链接的目标路径可以直接保存在 inode 所在的内存页之中。

3.4.2 租约锁和空间分配

多个线程（包括进程内和进程间的多个线程）可能会同时修改一个 Coffer 中的相同数据结构。为了对共享结构中的数据进行保护，文件系统通常会使用锁保证互斥访问。然而由于 ZoFS 是用户态文件系统，而用户态的进程很有可能会异常终止，使用锁进行共享资源的访问保护，可能会导致资源锁死的问题：一个进程中的 ZoFS 实例在持有锁的情况下终止，被锁保护的资源无法被释放出来。为了防止锁死问题，ZoFS 在实现中使用了租约锁（Lease Lock）。

租约锁 在 ZoFS 的租约锁设计中，每个锁维护了一个过期时间戳，当当前时间超过租约锁上时间戳之后，锁被自动释放。因此，即使进程在持有租约锁时被终止，在时间戳过期后，锁会被自动释放，其所保护的资源可以被其他人继续使用。

租约锁的实现需要在所有 CPU 上使用全局同步的时间戳。ZoFS 使用 `clock_gettime` 系统调用来获取租约的时间戳。得益于虚拟动态共享对象（Virtual Dynamic Shared Object, vDSO）的特性，虽然 `clock_gettime` 是一个内核提供的系统调用，但其可以在用户空间直接被处理，而不需要真的调用到内核空间，因此获取时间戳的效率很高。

基于租约锁的线程本地分配器 租约锁可以保护对共享对象的并发访问。然而，它对具有高度竞争（High-contention）的结构（如分配器）的可伸缩性提升不大。在 ZoFS 中，我们将租约锁和线程本地存储（即 per-thread 变量）结合起来，以提高分配器的可伸缩性。图 3–7 展示了 ZoFS 分配器中基于租约锁的线程本地空闲列表（Leased Per-thread Free List）。线程本地空闲列表的头部保存了一个线程号（Thread ID，TID）、一个租约过期时间戳（Lease）和空闲列表的表头。

ZoFS 会在每个 Coffer 的共享池中预先分配足够的线程本地空闲列表结构。当一个线程想要分配非易失性内存页时，它首先检查自己是否已经持有一个线程本地空闲列表结构，以及其租约是否仍然有效（即还未到期）。如果两者都为真，该线程会更新租约过期时间，并直接从其持有的线程本地空闲列表中进行分配。否则，它需要从共享池中找到一个未被其他线程使用的、或者已经过期的线程本地空闲列表结构，通过更新其 TID 和租约信息获取该结构的使用权，之后再使用该列表中的非易失性内存页进行分配。如果一个空闲列表中的空闲内存页数量不足，ZoFS 会向内核发起一个 `coffer_enlarge` 请求以获取更多的非易失性内存页。

基于租约锁的线程本地分配器提高了 ZoFS 的分配器的可伸缩性，同时，即使一个线程意外终止，它所持有的线程本地空闲列表（以及其中的空闲页）也可

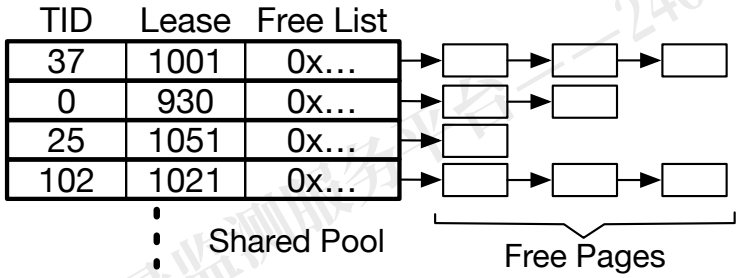


图 3–7 基于租约的线程本地分配器

以在租约时间到期后被其他线程重新使用。

3.4.3 一致性和恢复

ZoFS 使用原子指令，并使用同步的缓存行刷除指令，确保元数据的原子更新，以保证文件系统的崩溃一致性。如此前在介绍 SoupFS 二 时相同，每个文件系统操作都分为多个步骤，ZoFS 严格遵循这些步骤的顺序进行持久化，因此在发生崩溃后，ZoFS 可以对未完全完成的元数据更新进行回滚恢复。同时，为了简化设计，ZoFS 并未保证文件数据的原子更新。

ZoFS 的崩溃一致性依赖于恢复操作。虽然 Treasury 提供了在线恢复的方法，允许μFS 进行在线的恢复，但我们在 ZoFS 中只实现了离线恢复。在进行 Coffer 的检查和恢复时，ZoFS 首先扫描整个文件系统中的所有 Coffer，并按如下方式检查和恢复每个 Coffer 中的元数据：从 Coffer 根页开始遍历整个 Coffer，记录所有被使用的非易失性内存页和跨 Coffer 的元数据。当在遍历过程中发现有损坏的文件或目录项时，ZoFS 首先尝试识别并恢复。如果无法恢复，ZoFS 会跳过损坏的结构。在遍历结束时，ZoFS 会将 Coffer 中所有在使用中的非易失性内存页发送给 KernFS，KernFS 会通过比对其记录的分配信息，回收分配给该 Coffer 但未被使用的页面。在检查完所有在用元数据后，ZoFS 会根据 Coffer 遍历过程中记录的信息，继续检查和恢复每个跨 Coffer 的元数据的有效性。

ZoFS 是 Treasury 中μFS 的一个示例实现。应用程序的开发者或者文件系统的开发者可以在 Treasury 中利用 Coffer 抽象实现其他μFS，针对应用程序的需求设计更加高效的文件存储接口，或对新的高性能非易失性内存文件系统设计进行快速实现和验证。例如，可以在 Treasury 中实现一个提供键值接口的日志机制μFS 或者一个使用特殊日志结构组织的μFS。

3.5 评测效果与分析

在本章节中，我们将对 Treasury 中的 ZoFS 的性能进行评测。为了全面呈现 ZoFS 的特点，我们首先使用一组微基准测试对 ZoFS 的基本性能进行评估和分析，然后使用宏基准测试集展示 ZoFS 在模拟的综合工作负载下的表现，最后我们在 ZoFS 上运行生产环境中的真实应用，揭示 ZoFS 的优势和不足。

实验平台设置 评测中所使用的服务器平台装载有两颗十核心英特尔 ® 至强 ® 金牌 5215M 处理器。为了在评测过程中得到稳定的结果，我们禁用了超线程，并将处理器频率设置为 2.50 GHz。服务器配有 384 GB DDR4 DRAM 内存和 1.5 TB 英特尔 ® 傲腾 ™ 持久内存（Persistent Memory），平均分布在两个 NUMA 节点上。为了避免 NUMA 对性能的影响，我们使用 NUMA 0 上的 750 GB 持久性内存进行所有的实验。

为了展示 ZoFS 的性能，我们还测试了 Ext4-DAX、PMFS、Strata 和 NOVA 文件系统。由于 Strata 只能在某些测试中完成测试，故只在部分测试中展示了其性能情况。我们还尝试构建和运行 ZuFS[112] 和 Aerie 但未能成功，因此未加到测试之中。

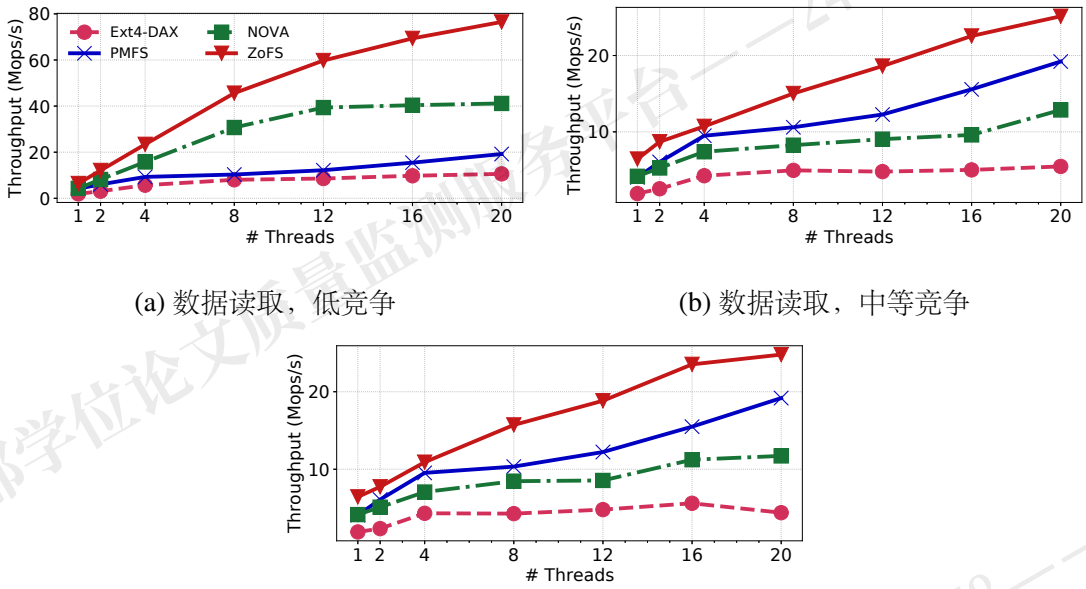
3.5.1 微基准测试

我们使用文件系统测试集 FxMark[113] 来评测 ZoFS 和其他对比文件系统的基本操作的性能和可伸缩性。

在 FxMark 的大多数工作负载中，ZoFS 达到了最高的吞吐量和可伸缩性，包括数据读取（图 3–8）、数据写入（图 3–9）和元数据操作（图 3–10）。总体来说，这主要得益于 ZoFS 在用户空间直接管理数据和元数据，避免了系统调用造成的上下文切换和虚拟文件系统中的复杂逻辑。我们在后文中会对性能提升进行详细分解和分析。

在数据读取的工作负载中，由于使用了读写锁（Readers-Writer Lock），无论竞争程度如何，所有文件系统的性能都能随线程数的增加而增长（图 3–8）。所有被测试的文件系统中，ZoFS 始终保持了最高的吞吐量。

图 3–9 中给出了 FxMark 中数据写入操作的吞吐量和可伸缩性。数据追加（图 3–9(a)）需要向新分配的数据页中写入数据，因此除了数据写入操作之外，空间分配器的实现也会影响文件系统的性能。ZoFS 使用基于租约的线程本地分配器，在不超过 12 个线程时，吞吐量随线程数增长而增长。在超过 12 个线程之后，吞吐量受限于 `coffer_enlarge` 请求在内核中的处理速度，无法继续提升。NOVA 使



(a) 数据读取，低竞争　　(b) 数据读取，中等竞争

(c) 数据读取，高竞争

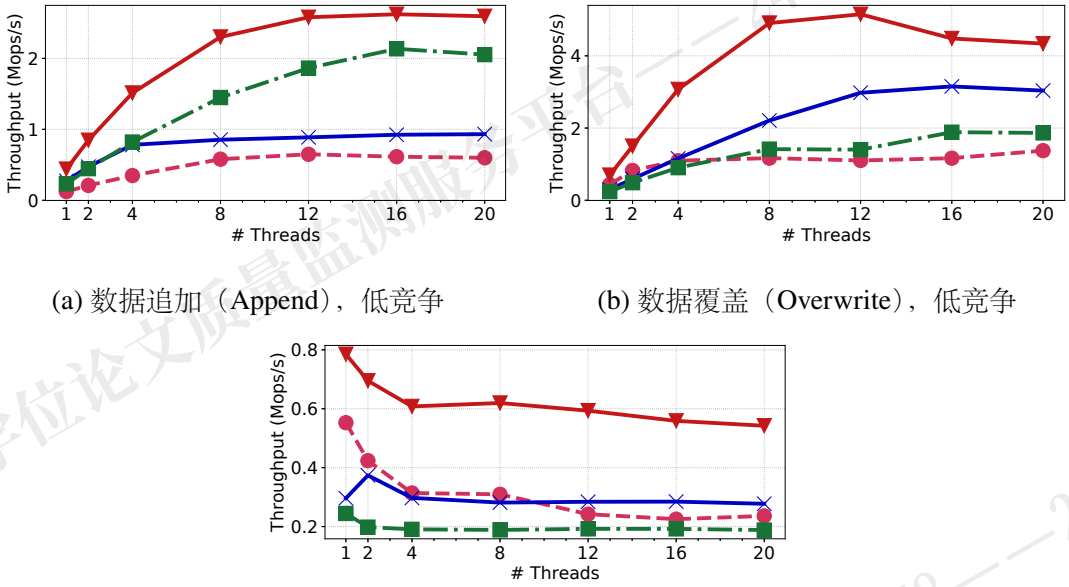
图 3–8 FxMark 测试集中的只读负载测试结果。每个数据读取操作读取 4 KB 的文件数据

用一个 CPU 本地分配器管理非易失性内存空间的使用，因此伸缩性在这种工作负载下也比较好。但绝对性能依然与 ZoFS 有所差距。PMFS 使用的是全局分配器，并未对可伸缩性进行特殊优化，因此在 4 个线程之后，吞吐量就停止增长。

图 3–9(b) 显示了不同线程同时覆盖写入 4 KB 数据的性能。在此工作负载中，每个线程向一个单独的文件的前 4 KB 进行写入，因此此工作负载中资源竞争程度不严重。在线程数为 12 时，ZoFS 达到了最高的吞吐量。经过计算，此时的文件写入速度已经到达了我们所使用的非易失性内存的写入带宽上限。而随着线程数继续增加，不同线程对非易失性内存的写入之间有所干扰，导致写入带宽降低，从而导致了 ZoFS 吞吐量的略微降低。

图 3–9(c) 展示了不同线程覆盖写入同一个共享文件中不同位置时，各个文件系统的性能。由于所有测试的文件系统都使用 inode 级别的锁，将并行的写入串行化执行，所以随着线程数量的增加，所有文件系统的吞吐量都不会提升，并有所下降。不过在这种情况下，ZoFS 仍然呈现出最高性能。

图 3–10 中给出了在 FxMark 测试集中的元数据测试。当不同的线程在不同的目录中创建文件时（图 3–10(a)），ZoFS 的吞吐量在 4 个线程之后就停止了增长，并被 NOVA 所超越。这是由于 `coffer_enlarge` 请求造成的资源竞争造成的。即使每个线程都有自己的本地分配器，当线程用完其本地分配器中的空闲空间时，会向 KernFS 发出 `coffer_enlarge` 请求，以获取更多的非易失性内存空间进行使



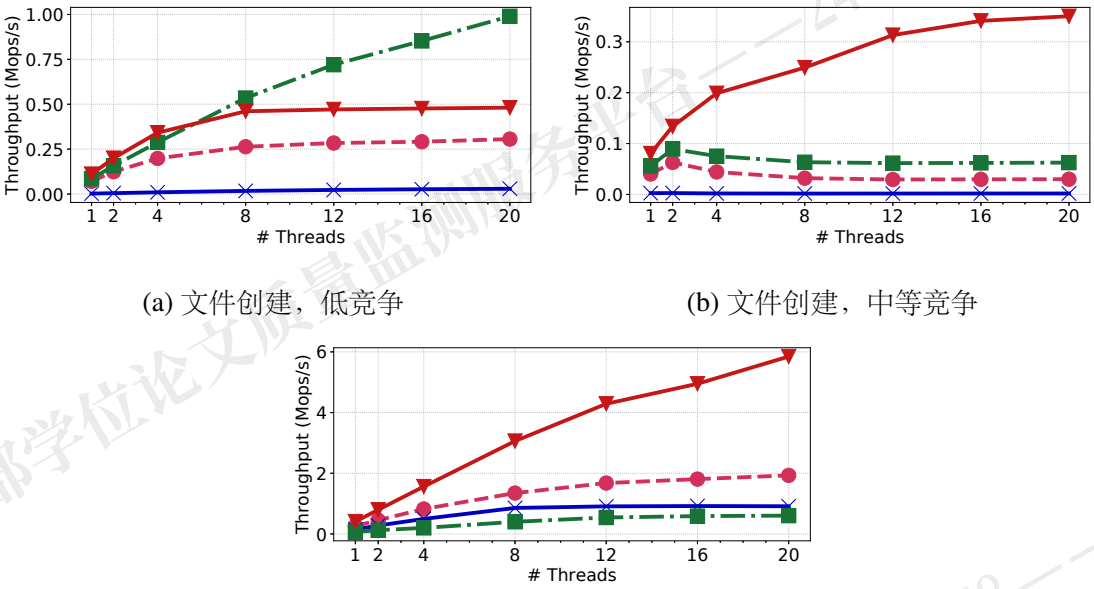
(a) 数据追加（Append），低竞争　　(b) 数据覆盖（Overwrite），低竞争

(c) 数据覆盖（Overwrite），中等竞争

图 3–9 FxMark 测试集中的写入负载测试结果。每个操作会写入 4 KB 的文件数据。图例在图 3–8(a)之中给出

用。当空间的分配操作极为频繁时，不同线程并发调用 `coffer_enlarge` 会在内核中发生资源竞争，造成性能不可伸缩的问题。这也是图 3–9(a) 中 ZoFS 在 12 个线程后吞吐量不再增长的原因。然而，在 NOVA 中，所有非易失性内存空间全部被提前分给所有的 CPU 本地分配器，因此，在 NOVA 中，需要更多的分配请求才会耗尽一个 CPU 本地分配器中的空闲内存，故在图 3–10(a) 中，NOVA 未受到分配器的限制，其吞吐量可以保持持续增长。

为了更好地说明 ZoFS 性能提升的来源，我们修改被测试的文件系统来分解文件数据覆盖写入操作的吞吐量，并在图 3–11 中给出结果。图中 `ZoFS-sysempty` 是 ZoFS 的一个变体，它在每次写文件之前发起一个空系统调用；`ZoFS-kwrite` 是另一个 ZoFS 变体，其文件写入操作是在内核空间中实现的；`NOVAi` 表示 NOVA 的直接写入版本（默认的 NOVA 是使用写时复制技术的）。后缀 `-noindex` 表示在 NOVA 实现中不更新文件写入的索引结构。图中的两个带 `-noindex` 后缀的 NOVA 实现由于去除了文件索引结构的更新，在文件数据覆盖写之外的工作负载上会产生错误，此处展示它们只是为了说明索引结构更新对性能的影响。当 `CLWB` 指令可用时，默认的 PMFS 实现会在非易失性内存写入之后使用 `CLWB` 指令将数据从缓存中刷出，确保数据持久性。在 `PMFS-nocache` 中，PMFS 会使用非临时性（Non-temporal）写入指令，直接绕过 CPU 缓存，直接写入到非易失性内存之



(a) 文件创建，低竞争　　(b) 文件创建，中等竞争

(c) 文件重命名，低竞争

图 3–10 FxMark 测试集中的元数据测试结果。图例在图 3–8(a)之中给出

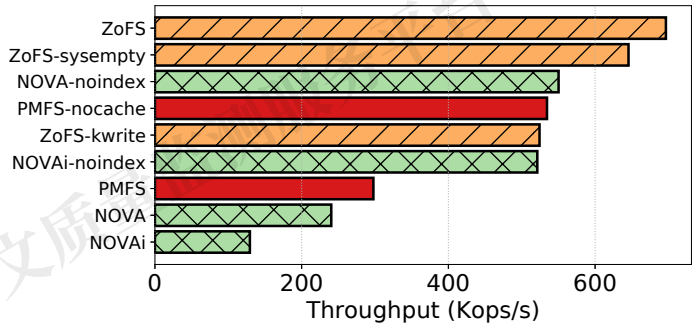


图 3–11 文件数据覆盖写入的性能分解

中。

图 3–11 中给出了分解后的文件系统性能。所有的系统按照性能可以分为三个梯队。ZoFS 的性能表现最好，其次是 `ZoFS-sysempty`。两者性能最高，处于速度最快的第一梯队。`NOVA-noindex`、`PMFS-nocache`、`ZoFS-kwrite` 和 `NOVAi-noindex` 在第二梯队中，表现相似。PMFS、NOVA 和 `NOVAi` 处于最慢的一个梯队。ZoFS 和 `ZoFS-kwrite` 的区别在于写入操作是在用户态实现还是在内核态实现，因此它们之间的吞吐量差异展示出了在用户空间实现文件系统的好处。对比多种 NOVA 实现的性能，可以发现索引结构的更新会显著影响性能。另外，由于

表 3–5 Filebench 中各个测试负载的参数特征

工作负载	文件数量	目录宽度	平均文件大小	读写比例
FileServer	10,000	20	128 KB	1:2
WebServer	1,000	20	16 KB	10:1
WebProxy	10,000	1,000,000	16 KB	5:1
Varmail	1,000	1,000,000	16 KB	1:1

测试中所有的写入的大小都是 4 KB，且和非易失性内存的页边距是对齐的，因此 `NOVAi` 与 NOVA 相比没有任何性能优势。在 PMFS 的测试结果中，我们发现相比于普通写入加 `CLWB` 的组合方式，使用非临时性写入指令的性能更高。为此，我们检查了其他文件系统的实现，确认 NOVA 和 ZoFS 在所有实验中都使用了非临时性写入指令。

简而言之，ZoFS 的性能优于现有的非易失性内存文件系统，在几乎所有被测试的工作负载中都表现出良好的可伸缩性。同时，ZoFS 在用户空间的实现是 ZoFS 能够取得高性能的关键。

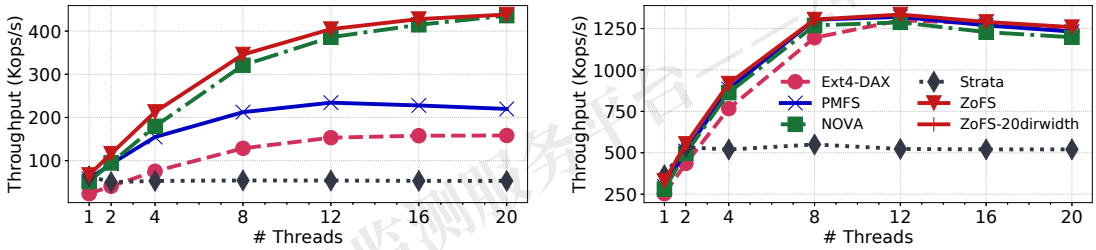
3.5.2 宏基准测试

我们还使用 Filebench 基准测试集对综合工作负载进行评测。工作负载的特性在表 3–5 中给出，测试结果如图 3–12 所示。

总体来说，ZoFS 在这四种工作负载中所表现出的吞吐量均是最高的。在单线程的 FileServer 工作负载中（图 3–13(a) 所示），ZoFS 的性能比 NOVA 高出 30%，比 PMFS 高出 16%，比 Strata 高出 5%。随着线程数的增加，ZoFS 和 NOVA 的吞吐量持续提升，它们之间的性能差距不断缩小。PMFS 性能提升幅度越来越平缓，在 12 线程之后不再增长，且在吞吐量上明显落后于 ZoFS 和 NOVA。Strata 的性能在 2 个线程时略有下降，然后随着线程增长而保持不变。

在图 3–12(b) 的 WebServer 工作负载中，只有一个线程时，ZoFS 的吞吐量比 NOVA 高 17%，比 PMFS 高 11%。除了 Strata 外，其他所有的文件系统的吞吐量在 12 个线程之前都能持续增长。

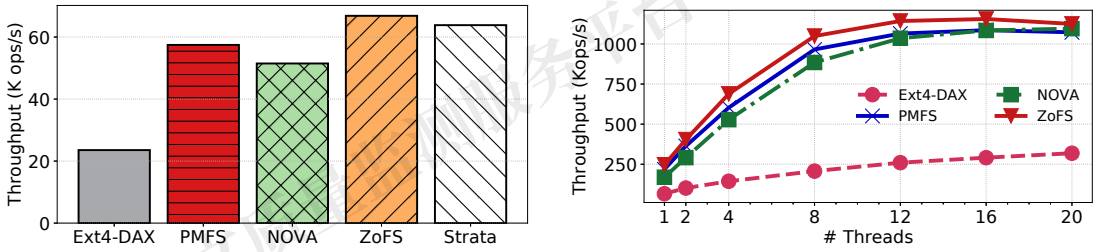
在图 3–12(c) 和图 3–12(d) 的 WebProxy 和 Varmail 工作负载中，ZoFS 的吞吐量始终是最高的。随着线程数的增加，ZoFS 和其他文件系统的性能差距会扩大。这主要是因为 WebProxy 和 Varmail 工作负载中的目录宽度比较大（表 3–5 中的目录宽度），导致所有文件都存储在一个目录中。由于 ZoFS 使用自适应的两级哈希管理目录（章节 3.4.1），ZoFS 的性能增长可以很好地延续到 12 个线程。PMFS 和



(a) FileServer　(b) WebServer

(c) WebProxy　(d) Varmail

图 3–12 ZoFS 和其他文件系统在 Filebench 测试集中的测试结果



(a) 单线程下 FileServer 工作负载的吞吐量　(b) 目录宽度为 20 时的 Varmail 工作负载

图 3–13 特殊配置下的 Filebench 测试集测试结果

NOVA 的吞吐量在 12 个线程之前就停止了增长，且吞吐量略有下降。这是由于它们的目录结构在大目录中查找文件的效率较低。为了证明目录宽度的影响，我们还测试了将目录宽度设置为 20 时，Varmail 工作负载上不同文件系统的性能。结果如图 3–13(b) 所示，所有文件系统的吞吐量都随线程数增加而提高，ZoFS 仍然比 PMFS 和 NOVA 分别高出 13% 和 46%。

值得注意的是，与默认 Varmail 配置下的性能相比，当目录宽度减少到 20 后，ZoFS 的性能有所下降。这是由于在总文件数固定的情况下，目录宽度减小会导致目录结构深度增加。因此，在工作负载中所访问的文件往往有很长的路径。然而，ZoFS 为了找到文件所在的 Coffer，会从后向前解析文件路径。换句话说，从路径

表 3–6 LevelDB 中 db_bench 测试结果

时延/微秒	Ext4-DAX	PMFS	NOVA	ZoFS
Write sync.	58.115	23.490	29.055	21.080
Write seq.	7.630	5.019	10.063	3.705
Write rand.	20.052	11.553	19.949	10.296
Overwrite.	30.536	18.223	30.336	16.835
Read seq.	1.389	1.079	1.220	1.071
Read rand.	4.472	3.553	3.990	3.523
Read hot.	1.192	1.164	1.187	1.146
Delete rand.	3.907	2.810	9.418	1.719

最长的前缀开始，依次检查路径的所有前缀，直到找到一个前缀是 Coffer 的路径。对于路径较长的文件来说，这种操作所消耗的时间较长，因此降低了 ZoFS 的性能。图 3–12(c) 和图 3–12(d) 中用 `ZoFS-20dirwidth` 表示的折线即为将目录宽度设置为 20 时 ZoFS 的性能。与默认配置下的性能相比，ZoFS 的吞吐量下降了 10% 到 30%。

3.5.3 真实应用测试

在本章节中，我们使用两个真实应用来评测 ZoFS 和其他文件系统的性能：LevelDB[114] 和 SQLite[115]。

LevelDB LevelDB 是一个被广泛使用在真实生产环境中的高速键值存储系统。我们在不同的文件系统上运行 LevelDB 自带的 `db_bench` 基准测试程序，并在表 3–6 中给出结果。在所有测试的操作中，ZoFS 的时延在所有被测试的文件系统中是最低的。在顺序写（Write sync.）和随机删除（Delete rand.）等操作中，ZoFS 的时延是 PMFS 时延的一半。PMFS 在实验中的时延排名第二，仅次于 ZoFS。NOVA 由于采用写时复制技术，在测试中的时延表现不如 PMFS。

TPC-C SQLite SQLite 是一个被广泛使用的轻量级 SQL 数据库引擎。TPC-C[116] 是一个在线事务处理标准测试，其模拟了一个订单处理应用。我们用 TPC-C 标准中的负载驱动 SQLite。

TPC-C 涉及五种类型的事务，包括新订单（`NEW`）、支付（`PAY`）、订单状态（`OS`）、交付（`DLY`）和库存查询（`SL`）。我们使用了四种不同工作负载进行测试。第

表 3–7 TPC-C 混合工作负载中各事务所占的比例

事务	NEW	PAY	OS	DLY	SL
占比	44%	44%	4%	4%	4%

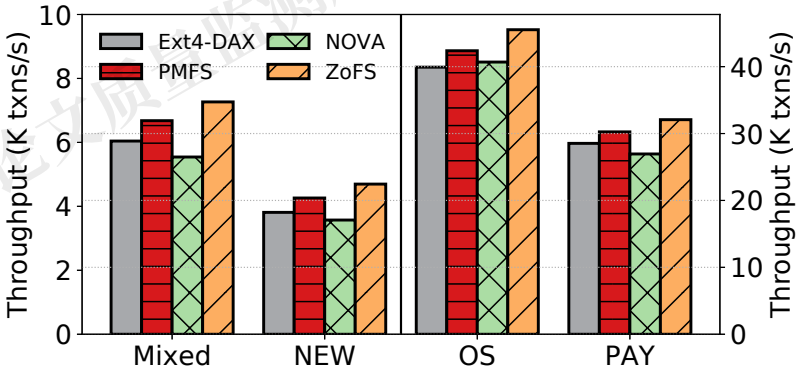


图 3–14 TPC-C SQLite 测试结果

一个工作负载是混合工作负载（Mixed），其中所有类型的交易都以表 3–7 中给出的比例执行。在其他三个工作负载中，我们分别只执行新订单（NEW）、订单状态（OS）和支付（PAY）三种事务。我们在客户表（customer）和订单表（orders）两个数据库表上建立二级索引，并按照 TPC-C 的规范要求设置外键。我们用一个线程运行每个工作负载，每个工作负载的规模为 1 个仓库（Warehouse）和 10 个区（District）。

图 3–14 展示了测试的结果。ZoFS 在混合工作负载中表现出了最高的吞吐量，并分别比 PMFS 和 NOVA 的吞吐量高出 9% 和 31%。同样地，由于 NOVA 中的写时复制机制，PMFS 的性能也稍稍优于 NOVA。其他三个工作负载的性能差异也类似。PAY 工作负载的吞吐量远高于 NEW 工作负载，这是由于 PAY 事务更简单、更快。在混合工作负载中，尽管 NEW 和 PAY 都占总事务数的 40% 以上，NEW 事务限制了总体吞吐量。OS 工作负载是只读事务，因此其吞吐量总体上高于 PAY 工作负载。在 OS 工作负载中，ZoFS 仍然比 PMFS 和 NOVA 分别高出 7% 和 11%。

3.5.4 最差情况测试

在本小节中，我们进行了一些特殊的测试，以研究 Treasury 在一些最不利场景下的最差表现。

由于 Treasury 的架构限制，Treasury 在处理跨 Coffer 的操作时会比较复杂。这

表 3–8 最差情况下的性能测试结果

时延/纳秒	NOVA	ZoFS	ZoFS-1coffer
chmod	1,830	23,342	675
rename	6,261	28,264	1,681

些操作必须调用 KernFS 来修改 Coffer 元数据，涉及到系统调用和上下文切换开销。为此我们实现了两个简单的测试程序，频繁改变文件权限（chmod）和改变文件路径（rename），来展示 Treasury 在这些情况下的性能如何。我们比较了 NOVA、ZoFS 和 ZoFS-1coffer 三个系统的性能。其中 ZoFS-1coffer 是 ZoFS 的一个变体，其无视文件权限的差异，将所有文件存储在同一个 Coffer 中，。

测试结果在表 3–8 中给出。在频繁改变文件权限的测试程序（chmod）中，所有文件最初具有相同的权限和拥有者信息，故被存储在同一个 Coffer 之内，测试程序将随机改变不同文件的权限或拥有者信息，在此过程中，ZoFS 需要频繁进行 Coffer 的拆分（coffer_split）操作，频繁进入内核态，并修改 Coffer 的元数据，修改 Coffer 中页面的所属 Coffer 等。这些操作会降低 ZoFS 的性能。如结果所示，ZoFS-1coffer 的时延最低，因为它无视了文件权限改变导致的 Coffer 变化，因此在用户空间直接处理了所有的权限变化。NOVA 的时延长于 ZoFS-1coffer，这是由于其需要系统调用，在内核中改变文件权限。ZoFS 时延最长，比 NOVA 和 ZoFS-1coffer 分别慢了约 12 倍和 33 倍。

在另一个频繁改变文件路径的测试程序（rename）中，所有文件被平均存储在两个 Coffer 之中。测试程序会随机挑选文件，将其重命名到另一个 Coffer 之中。此测试的结果与 chmod 测试程序类似，原因相同。这两个测试程序都证实了 ZoFS 中跨 Coffer 操作的高成本。

3.5.5 安全性和恢复测试

在本章节中，我们设计了一些测试来验证 Treasury 设计中的安全性和文件系统的恢复，包括对有缺陷代码和恶意应用的防护，和 ZoFS 的恢复时间和效果。

在安全性测试中，我们设置了两个进程 P1 和 P2，和两个 Coffer C1 和 C2。P1 将 C1 映射为可以读写，P2 将 C1 和 C2 均映射为可以读写。

第一个测试是测试 ZoFS 对有缺陷代码的误写问题防护。在测试中，P2 不断访问存储在 C1 中的文件，而 P1 向其内存空间中的随机地址写入随机数据，直到 P1 被内核终止。当 P1 开始在 ZoFS 外开始随机写如内存时，模拟的是应用代码

中的误写，在此情况下，P2 的对文件的正常访问没有受到影响。这是由于 P1 对 C1 内存的误写都被内存保护键机制拦截。当 P1 开始在 ZoFS 打开写窗口后开始随机内存写入，则其对 C1 的误写会破坏 C1 中的元数据，P2 在正常访问文件的过程中，收到了 ZoFS 返回的文件系统错误，但不会意外终止。该测试呈现了内存保护键机制对非易失性内存文件系统的保护，以及优雅的错误返回的有效性。

在第二个测试中，P1 作为攻击者，试图通过修改 C1 来访问 C2 中的数据。然而由于 C2 在 P1 中没有映射，所有这些访问都被内核阻止了。然后，P1 试图精心操纵 C1 的元数据，以诱导 P2 错误地访问 C2 中的文件（此时 P2 在不断访问 C1 中的文件）。测试结果显示，P2 从未访问 C2 中的文件。相反，P2 在运行过程中发现 C1 中元数据有错误，并将其报告出来。这些结果表明，ZoFS 可以成功防御恶意进程通过操纵元数据展开的攻击。

在第三个测试中，我们测量了恢复一个 Coffer 的时间和效果。当 ZoFS 访问一个损坏的文件系统时，该进程成功地检测到了损坏。然后我们启动 ZoFS 的恢复程序并测量时间。恢复一个保护有 1,000 个 2 MB 大小文件的 Coffer 总共需要 20,748 微秒，其中用户空间和内核空间分别占 5,386 微秒和 15,362 微秒。

3.6 其他相关工作

使用容器抽象的文件系统 许多文件系统都曾将不同的文件组合成一个类似于容器的抽象。Chunkfs[117] 将文件系统切分成多个 Chunk 以提升文件系统的可靠性和恢复性能。IceFS[118] 则提出了 Cube 的抽象，用于将文件系统结构与物理存储的结构解耦。SpanFS[119] 提出 Domain 的概念，提升固态硬盘上文件系统的可伸缩性。每个 Domain 都是一个微型文件系统用于管理整体文件系统的一部分。Zone 是在 BetrFS 0.4[120] 中提出并使用的类似的概念，其用于支持全路径索引中的快速重命名机制。本工作中的 Coffer 抽象使用了相似的思想。然而 Coffer 的目的在于将非易失性内存资源的管理和保护分离，以在提供足够的保护和隔离的情况下，让用户态非易失性内存文件系统拥有直接管理非易失性内存资源的权限。

用户态文件系统与存储管理 大量的用户态文件系统[121-123] 都基于 FUSE[124] 机制。然而由于 FUSE 的高开销，这些文件系统通常性能不高。ZuFS[112] 是一个以零拷贝（Zero-copy）为特色的新型用户态文件系统框架。在 ZuFS 中，数据不会在用户态和内核态直接来回拷贝，因此其开销远低于 FUSE。然而，使用 ZuFS 的文件系统依然需要使用系统调用，依然有上下文切换等在 Treasury 中避免了的开销。

Arrakis[125-127] 系统将文件系统的命名与文件系统的实现分离，并且允许应用

程序在虚拟化技术的帮助下直接管理传统存储设备。这与 Treasury 背后的思想是类似的。然而，Treasury 针对的非易失性内存设备，在性能和可字节寻址的特点上与传统设备相差巨大。

外核（Exokernel） 外核架构（Exokernel Architecture）将硬件资源的保护与管理分离，并赋予不可信的应用程序对硬件资源尽可能多的控制权[128-129]。Treasury 在某种意义上更像是一个外文件系统架构（Exo-filesystem Architecture）。其将非易失性内存资源的保护与管理分离，并给予用户态文件系统库对非易失性内存上的完全控制权。

用户态库隔离和保护 Hodor[130] 提出了一种利用英特尔内存保护键机制的方法保护用户态库的隔离性，其方法依赖于硬件的监视点（Watchpoint）机制和可信的加载器（Loader）提供安全性保障。ERIM[131] 同样利用英特尔内存保护键机制提供进程内的隔离，并通过二进制检查防止此保护机制被绕过。Treasury 将文件系统切分成 Coffer，并使用页表机制保护每个 Coffer 的访问权限。Treasury 使用英特尔内存保护键机制，和内核态与用户态内存空间的隔离，来进一步增强文件系统的隔离和保护。

3.7 本章小结

内核态非易失性内存文件系统由于跨模式访问和虚拟文件系统等原因无法充分发挥非易失性内存的性能，因而用户态非易失性内存文件系统逐渐成为了一个热门研究话题。

在本章中，我们介绍了用户态非易失性内存文件系统框架 Treasury 的设计和实现。通过提出 Coffer 抽象，Treasury 将权限和拥有者相同的文件和非易失性内存页组织在一起。随后，内核中的管理模块（KernFS）以粗粒度对 Coffer 进行管理和强隔离性保护，用户态的非易失性内存文件系统，如 ZoFS，则对 Coffer 内的文件数据和元数据（包括文件系统结构）进行全权管理。因此，大多数文件系统操作可以在用户态的非易失性内存文件系统中完成，避免了文件访问过程中的跨模式访问和虚拟文件系统等开销，从而可以最大程度地发挥非易失性内存的性能优势。同时，我们结合使用内存保护键等机制，在用户态对非易失性内存文件系统和应用程序进行保护和隔离，从而进一步提升了安全性。

虽然用户态非易失性内存文件系统可以在文件读写等负载中发挥非易失性内存的最高性能，在没有内核帮助的情况下，部分功能难以在用户态的文件系统中

实现或者难以高效实现。例如文件内存映射（mmap）功能，由于其实现过程中需要修改页表，用户态文件系统难以独立实现此功能。又如文件描述符的问题，虽然在我们的实现中通过序列化的方式对其进行了实现，但在 fork、exec 等系统调用的实现中，我们所使用的方法有些取巧（Hacky），并不能完全符合 POSIX 规范或完全符合现有内核所提供的语义。这使得用户态非易失性内存文件系统比较适合新型应用程序或对文件系统使用并不复杂的传统应用程序。

第四章　传统文件系统在非易失性内存上的优化

非易失性内存的高性能和可字节寻址的特性促进了大量新型非易失性内存文件系统的产生，其中包括新型内核态非易失性内存文件系统和新型用户态非易失性内存文件系统。然而，新型文件系统从出现到被大规模使用，往往需要经过多年时间，以不断丰富和完善功能，修补缺陷和漏洞等。而在此之前，使用现有成熟的传统文件系统管理非易失性内存，虽然无法完全发挥出非易失性内存的极致性能，但能够在对上层应用和整体环境影响最小的情况下利用到非易失性内存的性能优势。正如我们在章节 1.2.3 中所介绍，传统的文件系统，如 Ext2、Ext4 和 XFS 已经提出了直接访问（Direct Access，DAX）模式，去除了文件数据访问过程中的页缓存机制，让文件数据的读写直接在非易失性内存中进行，从而避免了不必要的数据拷贝开销，提升了非易失性内存上的文件访问性能。然而，这些传统的文件系统在非易失性内存上的性能依然与新型非易失性内存文件系统有所差距。

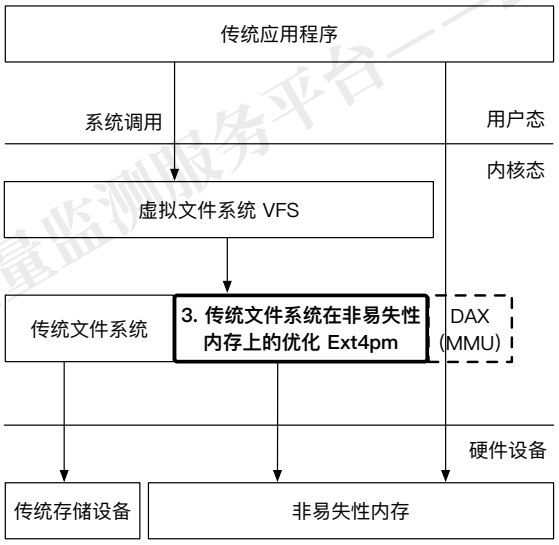


图 4–1 传统文件系统在非易失性内存上的优化 Ext4pm 在应用和内核架构中的位置

在本章中，我们提出，在保证兼容性的前提下，通过少量的代码修改，传统文件系统在非易失性内存上的性能可以被优化到与新型非易失性内存相近。为此，我们详细测试了新型非易失性内存文件系统（NOVA）和传统文件系统（Ext4、Ext2 和 XFS）在非易失性内存上的访问性能，通过分解文件数据和元数据访问操作的时延，我们找出了影响传统文件系统发挥非易失性内存性能的 10 个性能问题。除了一个非易失性内存专有的问题之外，我们将性能问题进一步归为 3 个根本原因，

包括过度抽象、非对称的实现和不可伸缩的设计。我们进而针对这些根本原因提出不同的优化策略，并实现针对性优化和应用能够解决性能问题的第三方补丁。最终在不修改 Ext4 存储结构、总修改量少于 1,000 行的情况下，提升了传统文件系统在非易失性内存上的运行性能。图 4–1 中可以看出，我们的优化 Ext4pm 作为传统文件系统 Ext4 的优化出现，当 Ext4 用于管理传统存储设备时，Ext4pm 部分的优化不会起任何作用，而当 Ext4 用于管理非易失性内存设备时，在使用相同存储格式的情况下，Ext4pm 优化过的文件系统能以更高的性能完成对文件的操作请求。

4.1 研究背景和动机

4.1.1 新型非易失性内存文件系统不够成熟

正如之前章节 1.2 中的介绍，非易失性内存的出现引发了一系列非易失性内存文件系统的研究工作。大量新型内核态文件系统被提出，以充分利用非易失性内存的高性能和字节粒度访问特性。如 PMFS[34] 是一个使用了轻量级日志机制的非易失性内存文件系统。其使用细粒度、轻量级日志机构保证文件系统更新的一致性，并允许使用内存映射机制让用户态应用程序直接访问非易失性内存上的文件数据。NOVA[36-37] 则将日志结构文件系统技术应用到非易失性内存之上，通过可伸缩的设计和内存缓存结构提升文件系统性能。本文第二章中的 SoupFS 则将软更新技术与非易失性内存技术相结合，允许文件系统操作的异步持久化，以将耗时的缓存刷除指令从关键路径中去除，降低文件系统操作的时延。

除了内核中的新型非易失性内存文件系统之外，用户态的新型非易失性内存文件系统也是近些年研究的热点。正如在第三章中所介绍，由于非易失性内存的可字节寻址允许用户态空间以简单的 `LOAD` 和 `STORE` 等 CPU 指令进行访问，用户态非易失性内存文件系统可以避免系统调用、上下文切换和内核中复杂的虚拟文件系统机制，充分发挥非易失性内存的性能优势。

用户态非易失性内存文件系统可以分为两类。第一类用户态非易失性内存文件系统需要与内核中专用的模块进行交互，内核中的模块负责对非易失性内存的管理和保护，而文件和目录的管理和访问、崩溃一致性的保证等文件系统功能在用户态完成。这类文件系统包括 Aerie[38], Strata[39] 和前文章节 3.4 中的 ZoFS。

另外一类文件系统，如 SplitFS[132]，Libnvmmio[133] 和 SubZero[134]，构建在现有的内核态文件系统之上，通过内存映射文件的方式在用户态管理对文件数据的访问。这类用户态文件系统通常只截获对文件数据的访问请求（如 `read` 和 `write`），提升对文件数据部分的访问性能。而对于文件元数据的操作（比如对文件权限的

修改）和对文件系统元数据的操作（如分配空间），这些文件系统选择将其留给底层内核中的文件系统进行管理。

尽管近年来有许多新型非易失性内存文件系统被提出，却没有一个被大规模应用和部署。一个新型文件系统被大规模使用之前一般会经历一个漫长且艰难的过程。这主要有两个原因。第一个原因是，除了高性能之外，保存了重要数据的文件系统还应该具有足够的健壮性、可靠性和安全性，以防止重要数据的泄露或丢失。对于完全重写的内核态和部分用户态非易失性内存文件系统来说，它们需要足够的时间和实践以证明其成熟程度；对于一些依赖于成熟内核态文件系统的用户态非易失性内存文件系统来说，尽管它们将文件系统元数据和崩溃一致性等关键操作交给了成熟的内核态文件系统，它们在用户态的实现需要劫持应用程序的系统调用，才能在不修改或重新编译应用程序的情况下实现文件系统的功能。而对应用程序的劫持和在用户态完成文件系统功能，存在多种边界情况，难以完全模拟内核系统调用所表现出的功能。

第二个原因是文件系统的生态环境。在使用某个文件系统时，系统管理员需要一系列用户态的工具来创建（如 `mkfs`）、查看（如 `debugfs`）、调优（如 `tune2fs`）和检查与恢复（如 `fsck`）文件系统。对于成熟文件系统来说，这些辅助工具同样比较成熟。而对于新型非易失性内存文件系统来说，开发者依然需要时间来实现和完善这些工具，且系统管理员需要时间学习和掌握这些工具的使用。

总结来说，尽管许多新型非易失性内存文件系统被提出，成熟的传统内核态文件系统在一段时间内依然是比较合理的选择，而这些成熟的传统文件系统的性能依然值得被关注。

4.1.2 成熟的传统文件系统性能有待提升

然而，想要完全发挥非易失性内存的性能，传统的文件系统面临着多重挑战。首先，针对非易失性内存的优化不应该影响传统文件系统在传统块存储设备上的原有行为和性能；其次，传统文件系统的现有功能特性不应该被针对非易失性内存的优化所破坏。再次，优化对传统文件系统的修改应尽可能的少，且与现有代码的耦合度要低。同时，为了保证兼容性，改变传统文件系统的核心结构以提升在非易失性内存上的性能并不现实。所以，即使现有的传统文件系统支持“直接访问”（Direct Access，DAX）的特性，传统文件系统在非易失性内存上的性能依然无法与新型文件系统相提并论。

为了展示传统文件系统在非易失性内存上的性能，我们使用著名的键值存储系统 RocksDB[135] 中的测试程序 `db_bench` 进行了一个实验。实验中测试了包括

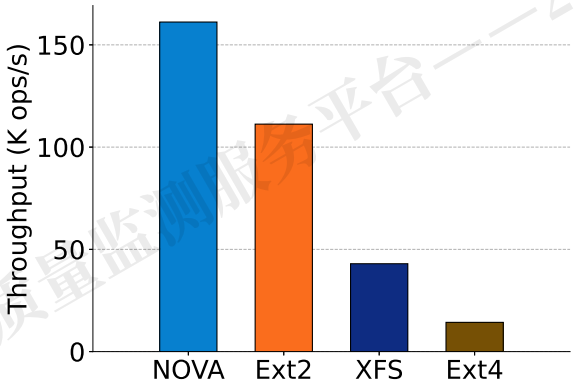


图 4–2 RocksDB 在传统文件系统与新型非易失性内存文件系统上的性能测试结果，传统文件系统的性能远差于新型非易失性内存文件系统的性能

成熟的传统文件系统 Ext2、Ext4 和 XFS 与为非易失性内存设计的新型文件系统 NOVA[36-37]。

在实验中，我们向 RocksDB 中同步插入 2,000,000 个键值对。每个键值对由 16 字节的键和 300 字节的值组成。图 4–2 中给出了实验的结果。可以看出，NOVA 取得了最高的吞吐量，远超出传统文件系统的性能。在传统文件系统之中，Ext2 由于不需要保证强崩溃一致性，比 Ext4 和 XFS 的性能更高。

基于这些结果，我们在本文中将讨论并尝试回答以下两个问题：(1) 为何在非易失性内存之上，传统的文件系统性能不如新型文件系统？(2) 我们是否以及如何能进一步提升传统文件系统在非易失性内存上的性能？

4.2 性能问题和应对策略总览

通过测试和对比传统文件系统和新型文件系统在非易失性内存上的性能，并对性能差距进行分析，我们共发现十个导致 Ext4 在非易失性内存上性能和可伸缩性不如新型文件系统的原因，并将其展示在表 4–1 中。除了一个特殊的性能问题之外，这些性能问题可以被分为三个根本原因：过度抽象、非对称的实现和不可伸缩的设计。在介绍细节之前，我们首先对这三个根本原因进行介绍。

1. **过度抽象**。抽象是软件在进行模块化以及代码重用中非常重要的手段。然而，由于非易失性内存的性能极高，抽象所带来的性能开销也逐渐变得不可忽视。在复杂的访问接口问题中，`.read_iter` 和 `.write_iter` 两个虚拟文件系统要求的接口虽然功能全面，但对于简单的 `read` 和 `write` 系统调用来说过于复杂，因

表 4–1 本工作发现的性能问题的汇总，以及每个问题对应的解决方法（优化或第三方补丁）。除了内存拷贝问题之外，所有的性能问题可以归为 3 个根本原因，在表格最后一行中给出。这 3 个根本原因揭示了为何 Ext4 在非易失性内存上的运行性能与新型非易失性内存文件系统有所差距

	性能问题（根本原因）	优化或应用的补丁
数据读取/Read	机器异常安全的内存拷贝 (4)	乐观的机器异常安全的内存拷贝（+C）
	复杂的访问接口 (1)	使用简单高效的接口（+I）
数据写入/Write	复杂的访问接口 (1)	使用简单高效的访问接口（+I）
	保守的数据擦除 (1)(2)	使用有效数据初始化文件块（+Bi）
	不可伸缩的孤立 inode 链表 (3)	孤立 inode 文件（+Of）†
	低效的日志提交 (1)(2)(3)	快速提交（+Fc）†
文件创建/Create	低效的目录查找 (2)	目录项全缓存（+L）
	低效的目录项分配 (2)	目录项分配提示（+A）
	多余的 inode 修改同步 (1)	延迟的 inode 同步操作（+D）
	缓冲区伪共享 (3)	分离的 inode 分配（+S）
	日志空间耗尽 (1)(2)(3)	扩大日志空间（+J）
文件删除/Unlink	日志空间耗尽 (1)(2)(3)	扩大日志空间（+J）
	不可伸缩的孤立 inode 链表 (3)	孤立 inode 文件（+Of）†
	根本原因: (1) 过度抽象 (2) 非对称的实现 (3) 不可伸缩的设计 (4) 其他	
	† 快速提交和孤立 *inode* 文件为两个第三方补丁	

而带来了许多不必要的性能开销。多余的 *inode* 修改同步问题和保守的数据擦除问题则是由于抽象间的语义沟（Semantic Gap）带来的性能问题。日志相关的问题（例如日志空间耗尽问题和低效的日志提交问题）则与块抽象相关。在非易失性内存文件系统中，传统的块抽象在多数情况下是不必要的，只会带来额外的性能瓶颈。

2. **非对称的实现**。在存储设备性能普遍不高的时代，存储设备几乎一直都是存储系统中的性能瓶颈。因此，传统文件系统的设计一般关注于如何最大程度地利用好存储设备的访问特性。例如，对于机械硬盘，文件系统需要考虑如何合理地安排对机械硬盘的访问请求的顺序，以减少随机访问，避免冗长的寻道时间。为了做到这一点，文件系统中使用一些较复杂的算法也是可以接受的。因为这些复

杂算法所节省下来的存储时间要远长于这些算法本身所消耗的时间，收益远大于代价。然而，随着存储设备性能的提升，尤其是对于非易失性内存这种接近内存速度的存储设备，文件系统原有的这些设计与存储设备的性能已经出现了非对称的变化，这些设计严重影响了文件系统的性能。我们在 Ext4 中发现的许多问题都可以归为此类，包括保守的数据擦除、低效的目录查找、低效的目录项分配和日志相关的问题。这些设计和实现在慢速存储设备上几乎不会造成性能问题，然而在非易失性内存上成为了性能瓶颈。

3. **不可伸缩的设计**。传统文件系统中的一些结构在设计时未考虑或未将多核可伸缩性做为首要设计目标。在高性能且高可并发的非易失性内存上，对这些结构的操作成为妨碍 Ext4 在非易失性内存上性能伸缩的瓶颈。与非对称的设计不同，在单线程情况下，不可伸缩的设计和实现本身在非易失性内存上并不一定会造成性能下降，但当多个线程共同访问时，文件系统的性能无法提升，甚至会由于资源竞争导致性能严重下降。在我们后面的深入分析中，不可伸缩的孤立 *inode* 链表问题和缓冲区伪共享问题是这一类问题的典型代表。日志相关问题则主要由 Ext4 所使用的日志机制 JBD2[136] 的不可伸缩问题所导致。

4. **其他问题**。机器异常安全的内存拷贝是一个特殊的性能问题。其主要由非易失性内存的字节粒度的访问方式导致，不属于前 3 种根本原因，因此我们将其单独列出。

尽管这些性能问题是通过测试和分析 Ext4 而总结出来的，但是我们认为这 3 个根本原因对于其他传统文件系统同样适用。由于传统文件系统是从传统的慢速存储设备时代演化而来，在进行设计时并为针对如今的新型高速存储设备的特性，因此才会出现这 3 个影响性能的根本原因。然而这是无法避免的，当非易失性内存某一天也成为相对较慢的存储设备时，这些问题依然会出现。但这并不意味着我们无法通过较小的更改，将传统的文件系统在非易失性内存上的性能优化到与新型文件系统相当。

通常来说，优化现有的文件系统比从零开始设计和实现一个新的文件系统要更加复杂。这是由于在进行优化时，文件系统的开发人员需要考虑到文件系统的兼容性。旧的文件系统可能已经在众多设备上被部署和使用，新的文件系统优化应兼容现有的文件系统存储格式且不应破坏现有的文件系统语义。在此基础上，针对我们前面总结的影响传统文件系统在非易失性内存上性能的三个根本原因，我们提出下面几个优化策略。

1. **优先使用简单高效的抽象**。在完成一个相同功能时，往往有多种不同的抽象和方法。这些不同的抽象和实现方法有不同的侧重点：一个复杂的抽象

需要处理多种情况;而一个简单的抽象往往只需要也只能处理某一种情况。针对某个特定功能，使用最直接最简单的抽象可以避免过度抽象带来的性能开销，往往是最高效的方法。本工作在 Ext4 的*使用简单高效的访问接口*优化中，将复杂的 `.read_iter` 和 `.write_iter` 接口更换为更加简单的 `.read` 和 `.write` 接口正是遵循了此方法。

2. **打破一些抽象**。如果在不同抽象层次之间共享信息能够获得足够多的性能收益，那么略微打破一下抽象也是一个值得考虑的优化方法。本工作在 Ext4 上的*使用有效数据初始化文件块*优化和*延迟的 inode 同步操作*优化正是通过打破不同抽象层次之间的隔离而完成的，*使用有效数据初始化文件块*优化将用户要写入的有效数据透过多层抽象传递到了文件块分配函数中；*延迟的 inode 同步操作*打破不同函数之间的隔离，维护了共享的 inode 修改信息，从而减少了不必要的 inode 同步操作。
3. **调整参数好过修改代码**。系统中总会存在某部分成为性能瓶颈。如果通过修改现有参数就可以解决性能瓶颈问题，那么调整参数比修改现有的系统设计和实现是一种更加简单和直接的优化方式。*扩大日志空间*优化正是通过修改预先分配的日志空间的大小参数解决日志空间耗尽问题。这种调整参数的方式不需要对系统设计具有非常深入的了解，且由于不需要修改代码，在解决问题的同时不会引入新的软件缺陷。
4. **使用普通内存中的结构作为缓存**。虽然非易失性内存可以以内存方式访问，普通内存的访问性能依然优于非易失性内存。因此，NOVA 等新型非易失性内存文件系统依然会使用普通内存中的数据结构作为缓存保存临时元数据。因此，只要元数据一致性和持久性不被破坏，使用普通内存作为缓存，可以在不大量修改现有实现的情况下，提升传统文件系统在非易失性内存上的性能。*目录项全缓存*优化和*目录项分配提示*优化通过在普通内存中缓存元数据，分别提升传统文件系统在非易失性内存上进行目录查找和目录项分配时的性能。
5. **逐步改变文件系统的设计**。为了适应应用程序的新需求和对越来越快的存储设备，传统文件系统也在逐步进行演化。这种演化通常由一系列小的改变组成。*快速提交*和*孤立 inode 文件*两个第三方补丁均是对传统文件系统（特别是存储格式）的逐步改变，因此处于此类策略。

上述列表列出了本工作对 Ext4 在非易失性内存上的性能优化。在下面一个章节中，我们将对 Ext4 在非易失性内存上的性能和性能问题进行深入分析，并详细介绍我们为此提出的各种优化。

4.3 深入分析与优化

在本节中，我们将使用 FxMark[113] 中的测试负载，对文件系统的一些常用接口进行测试，包括数据操作接口（`read`、`write` 和 `mmap`) 和元数据操作接口（`lookup`、`create`、`unlink` 和 `rename`)。对于每个操作，我们首先展示不同文件系统在非易失性内存上进行该操作的吞吐量和可伸缩性，再通过分解每个操作的时延来准确地定位传统文件系统在非易失性内存上的性能瓶颈。我们测试的传统文件系统包括 Ext4、Ext2 和 XFS，以及新型非易失性内存文件系统 NOVA。由于 Ext4 是 Linux 中支持“直接访问”模式最流行的文件系统之一，而 NOVA 是在学术界比较流行，且向 Linux 社区中提交了补丁的非易失性内存文件系统，因此，在本文的分析中，我们选择着重关注 Ext4 和 NOVA 这两个比较有代表性的文件系统。

测试平台与环境搭建 我们使用一台配载有两颗 28 核心英特尔 ® 至强 ® 金牌 6238R 处理器的服务器作为测试平台。测试平台共有两个 NUMA 节点，每个节点配有 192 GB 的 DDR4 普通内存和 768 GB 英特尔 ® 傲腾 ™ 持久内存。

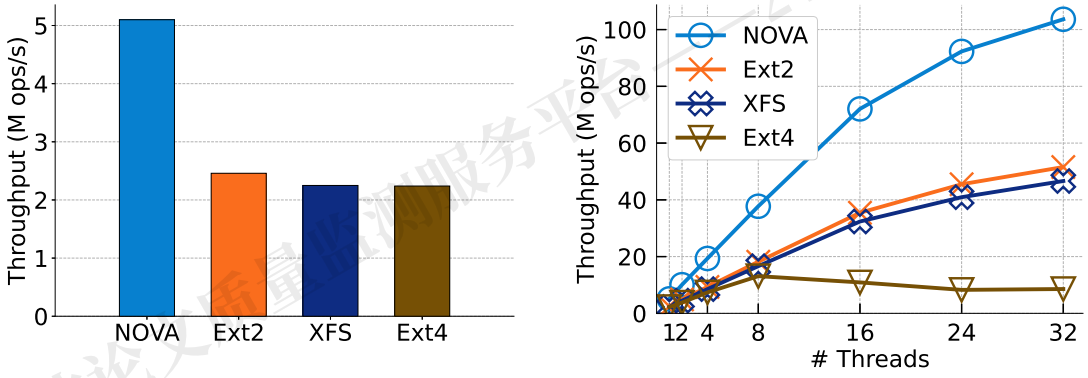
我们将所有测试线程和内存与所测试的非易失性内存设备绑定在同一个 NUMA 节点之上，以避免跨 NUMA 节点的访问对测试结果的影响。同时，我们将处理器的超线程特性打开，以支持更多的测试线程。测试平台的操作系统是 Fedora 32，Linux 内核版本为 5.8.1。

对于传统文件系统，我们使用默认的参数创建文件系统，并使用“直接访问”模式进行挂载。我们测试的 NOVA 使用覆盖更新（In-place Update）模式，而非写时复制模式，以避免写时复制技术带来的写放大问题。这样能够让 NOVA 在测试中展现出更高的性能[37]。

4.3.1 文件数据读取操作分析

我们首先测试的是使用不同的线程数，不断地读取不同文件中的 4 KB 文件数据。图 4–3(a) 中给出了单线程时的性能对比图，可以看出在单线程情况下，新型文件系统 NOVA 取得了最高的吞吐量。所有传统文件系统的性能表现相似，吞吐量不到 NOVA 吞吐量的一半。

图 4–3(b) 给出了吞吐量随线程数增多的总体情况。NOVA 的吞吐量随着线程数增多而提高，始终保持大概两倍于传统文件系统中最高的吞吐量。Ext2 和 XFS 的吞吐量也随着线程数增加而提高，但趋势比 NOVA 更平缓。然而对于 Ext4 来说，在 8 线程的时候，其吞吐量到达了顶点，更多的线程反而导致了其吞吐量的下降。



(a) 读取 4 KB 文件数据的吞吐量（单线程）　　(b) 读取 4 KB 文件数据的吞吐量

图 4–3 使用传统文件系统和新型文件系统在非易失性内存上读取 4 KB 文件数据的吞吐量对比。新型文件系统 NOVA 的性能超过所有的传统文件系统

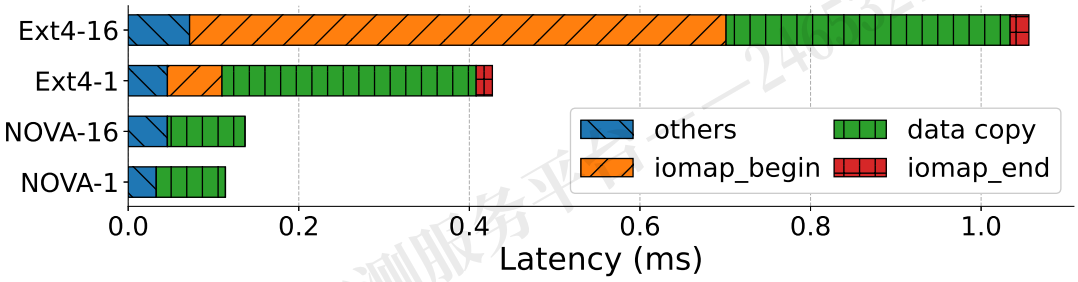


图 4–4 在单线程（图中“-1”后缀）和 16 线程（图中“-16”后缀）情况下，一个 4 KB 文件数据读取操作的时延分解。对于单线程情况来说，Ext4 中的数据拷贝（data copy）占用了最多的时间，`iomap_begin` 操作的时延在 16 线程下明显增加

为了寻找 Ext4 实现中的性能瓶颈，我们对 Ext4 和 NOVA 的读取 4KB 操作的时延进行了分解和对比，图 4–4 中给出了结果。

机器异常安全的内存拷贝　图 4–4 中的 *data copy* 部分占据了 *Ext4-1* 中文件读取操作中的大部分时间。在进一步检查其实现之后，我们发现大多数时间消耗在 `memcpy_mcsafe` 函数中。这个 `memcpy_mcsafe` 函数是内存拷贝函数 `memcpy` 的一个变种。当在拷贝过程中出现了机器异常（Machine-check Exception）时，使用默认的 `memcpy` 函数会导致整个操作系统崩溃重启；而使用 `memcpy_mcsafe` 进行拷贝的话，机器异常会被内核捕获并处理。

为了能够捕获并报告触发机器异常的内存地址，`memcpy_mcsafe` 函数中将每

个从非易失性内存中进行读取的 MOV 指令位置进行了标记。当发生机器异常时，可以根据这些标记得知出错的位置。为了能够标记每条 MOV 指令，memcpy_mcsafe 使用循环的方法进行实现，而无法使用硬件提供的指令级别优化，比如在默认的内存拷贝函数 memcpy 中使用的 rep movsb 指令。我们对上述不同的内存拷贝指令进行了测试，发现 memcpy_mcsafe 从非易失性内存上拷贝 64 KB 数据需要用时 25,088 纳秒，而默认的 memcpy 指令只需要 22,538 纳秒。如果数据已经缓存在 CPU 缓存中，那么使用 memcpy_mcsafe 的数据拷贝时间为 3,974 纳秒，而默认的 memcpy 只需要 1,940 纳秒。因此，基于循环实现的 memcpy_mcsafe 函数比使用硬件指令级优化的 memcpy 性能差，且差距明显。考虑到非易失性内存读取错误发生的概率较低，使用性能差、但能保证机器异常安全的 memcpy_mcsafe 作为默认数据拷贝函数并不是一个合理的选择。

优化：乐观的机器异常安全的内存拷贝（+C）

根据我们此前的分析，内存拷贝操作占了大部分文件读取时间。考虑到非易失性内存的读取异常很少发生，我们提出了一种*乐观的机器异常安全的内存拷贝*机制。在此机制中，我们首先尝试使用硬件指令优化的 memcpy 进行数据的拷贝，只有在读取发生了错误之后，再回落到使用性能差，但机器异常安全的 memcpy_mcsafe 函数。

具体来说，我们首先将 memcpy 的参数保存在 CPU 本地变量中，并修改一个 CPU 本地的标记表示内存拷贝正在进行。之后，我们使用硬件指令优化的 memcpy 进行数据拷贝。如果在拷贝过程中一切正常，则 memcpy 可以顺利完成，我们将此前的 CPU 本地变量清空即可。若发生了读取错误，硬件会调用机器异常的处理函数。在处理函数中，我们首先检查 CPU 本地的标记，确定是否在进行内存拷贝操作，以及造成机器异常的是否为内存拷贝操作，如果是，则将控制流跳转到一个特殊的处理函数中，并在其中调用 memcpy_mcsafe 函数再次进行数据的拷贝，并获取成功拷贝的字节数作为函数返回值。

由于在最新的 CPU 中已经不再需要此机制，且在更新的内核版本中实现了 copy_mc_enhanced_fast_string 函数[137]，虽然我们依然保留了对此问题的分析，但是为了公平，我们在测试中将此优化认为是 Ext4 基准的一部分。

复杂的访问接口 排除掉数据拷贝所占用的时间之后，剩余时间大多数被 *iomap_begin* 和 *iomap_end* 所占用。这两个接口是虚拟文件系统中基于迭代器的访问接口（.read_iter）所必须的。

在虚拟文件系统的框架下，文件系统有两个方式可以实现对文件读取功能：一种是实现简单的 .read 接口；另一种是实现基于迭代器的 .read_iter 接口。

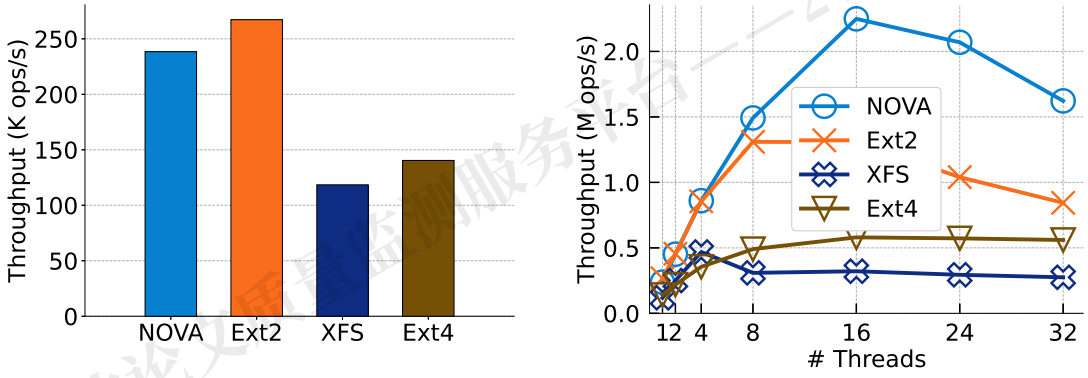
.read 接口是一个非常简单的接口，其接受的参数与用户调用 read 系统调用时所传递的四个参数相对应：要读取的文件、用户提供的数据缓冲区、要读取的数据在文件中的偏移量以及要读取的数据长度。而基于迭代器的 .read_iter 接口则相对复杂，其接受的参数为两个特殊的结构，其中一个（kiocb）用于保存要读取的文件、文件的当前读取位置等信息，另一个为迭代器结构（iov_iter），其中保存了一组数据缓冲区，以及每个缓冲区的长度。虽然 .read_iter 非常复杂，但其既可以用于处理单缓冲区的 read 系统调用，也可以用于处理具有多个缓冲区的读文件请求系统调用，例如 readv 和 io_submit。

如同其他的传统文件系统一样，Ext4 只实现了 .read_iter 接口来完成对文件读的访问。而使用基于迭代器的 .read_iter 接口处理简单的单缓冲区文件读（read）系统调用至少在两方面会带来不必要的开销。首先，为了使用基于迭代器的 .read_iter 接口，虚拟文件系统需要将用户提供的单一缓冲区转换成迭代器结构。而此过程引入和涉及到了大量不必要的辅助结构，造成性能开销。其次，为了使用基于迭代器的 .read_iter 接口，Ext4 还需要提供一个 iomap_begin 函数的实现。Ext4 中现有的 iomap_begin 是一个通用的函数，其除了被文件读取使用之外，还被用于处理文件写请求和缺页中断的处理函数。因此，iomap_begin 的实现要考虑多种情况，这导致在其被用于读取请求处理时，会产生不必要的资源竞争。具体来说，在 Ext4 的 iomap_begin 实现中，Ext4 获取了一个读锁用于检查当前日志系统的状态。在获取读锁时需要更新读写锁中读者计数器，因而会引发内存竞争问题，这是 iomap_begin 函数在图 4–4 中 16 线程时时延大幅增加，性能无法拓展的原因。

而在 NOVA 中，两种接口 .read 和 .read_iter 都被实现了。在处理 read 系统调用时，虚拟文件系统会优先调用 .read 接口。而 .read 接口的实现简单且直接，避免了复杂的 .read_iter 机制带来的性能开销和可伸缩问题，因此更加高效。

优化：使用简单高效的接口（+I）

为了提升 Ext4 在非易失性内存上的性能，我们为 Ext4 实现了 .read 接口，因此在使用 read 系统调用进行文件读取时，虚拟文件系统不会调用复杂的基于迭代器的接口（.read_iter）。在具体实现中，我们重用了 Ext4 中一些现有的函数，如用于计算文件逻辑块号的函数，以较少引入的新代码量。同时，我们用同样的方法实现了更简单的文件写入接口 .write，以避免相同问题对文件写入性能的影响。



(a) 文件数据追加的吞吐量（单线程）　　(b) 文件数据追加的吞吐量

图 4–5 使用传统文件系统和新型文件系统在非易失性内存上向文件中追加写入 4 KB 数据的吞吐量对比。NOVA 在测试中性能最高；Ext2 由于其实现简单，居于第二；Ext 和 XFS 的性能较差，且无法随线程数增高

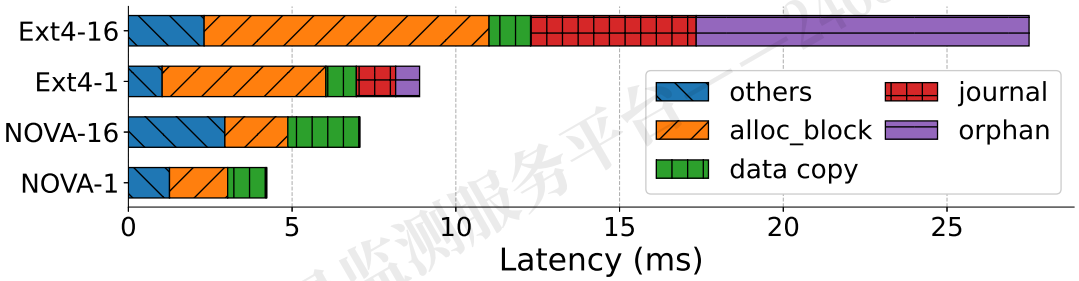


图 4–6 在单线程（图中“-1”后缀）和 16 线程（图中“-16”后缀）情况下，一个文件数据追加写入操作的时延分解。数据块的分配（*alloc_block*），日志（*journal*）和全局孤立 inode 列表（*orphan*）导致 Ext4 的时延过高

4.3.2 文件数据写入操作分析

对于文件写入请求，我们首先测试了 4 KB 数据的追加写入的性能，并将结果展示在图 4–5 之中。在单线程测试中（图 4–5(a)），Ext2 的吞吐量比 NOVA 略高，而随着线程数的增加（图 4–5(b)），Ext2 的吞吐量在 8 个线程时就达到顶峰，且逐渐开始下降。而 NOVA 的伸缩性要优于 Ext2，其在 8 线程时吞吐量超过 Ext2，并在 16 线程时达到吞吐量的顶峰。其他文件系统在单线程时，吞吐量大概为 Ext2 和 NOVA 的一半，且在多线程中吞吐量在 4 线程之后几乎没有增长。

为了进一步探究为何 Ext4 的性能与 NOVA 差距如此之大，我们分别将 NOVA 和 Ext4 在单线程与 16 线程时的时延进行了分解分析。图 4–6 中给出了分解的结

果。在 Ext4 的单线程测试（*Ext4-1*）中，对新文件块的分配的函数（*alloc_block*）占据了大多数的时间。此外占用时间较多的两个部分分别是日志（*journal*）和孤立 inode 列表（*orphan*）。文件块分配、日志和孤立 inode 列表这三部分除了影响单线程的数据之外，还为 Ext4 的多核可伸缩性带来了问题。在 16 线程的时延分解中，这三部分的时延随线程数量的增多而分别变慢，占据了 *Ext4-16* 中的大部分时间。

对于日志机制，与基于 CPU 本地思想而设计的 NOVA 系统不同，Ext4 使用了一个全局的日志系统，在此系统中，有大量的元数据被多个线程共享使用，因此会对 Ext4 的多核可伸缩性造成较大影响。而对于另外两个部分，我们接下来进行深入分析。

保守的数据擦除 Ext4 使用 `ext4_map_blocks` 函数保证文件中需要使用的逻辑块均已经被分配，且被映射到内存空间中。若在之前这些逻辑文件块未被分配，则此函数会分配新的文件块，将其进行清零操作，此后将其加入到该文件的区段状态树中。Ext4 随后便可将用户数据写入到对应文件块中。因此，在处理文件写入的过程中，Ext4 中每个新分配的文件块会被写入两次：一次清零，另一次写入正确的用户数据。此处的清零操作看似多余，但实则对于文件系统数据安全非常关键。由于一旦文件块被加入到区段树中之后，内核中的其他线程就可以通过区段树访问到该文件块，如果在文件块加入到区段树中之前未进行清零操作，则其他并发线程可能会读取到残留在文件块中的旧数据，造成文件系统的数据泄露。然而根据我们的分析，清零操作在空间分配中占据了大量的时间，确实影响了 Ext4 在非易失性内存上的性能，因此如果能够在保证数据安全的情况下，去除清零操作，则可以令 Ext4 在非易失性内存上的文件追加操作性能有所提升。

优化：使用有效数据初始化文件块（+Bi）

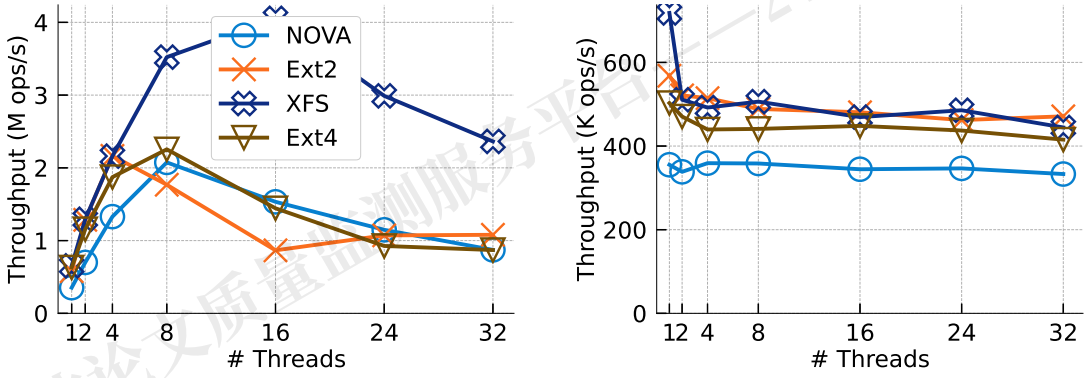
为了解决文件块被两次写入的问题，我们提出在将新分配的文件块插入到区段树中之前，使用用户想要写入的数据对文件块进行初始化，而非进行无意义的清零操作。为了做到这一点，我们稍微打破了 Ext4 中的抽象层次，将额外信息（包括用户提供的缓冲区、文件内偏移量和写入大小）直接传递到 `ext4_map_blocks` 函数之中。因此 `ext4_map_blocks` 在分配文件块时，直接使用用户的数据进行初始化；对未被用户数据覆盖的部分，再进行清零操作，从而在文件写入过程中，每个新分配的文件块最多只会被写入一次（要么写入用户的数据，要么被清零），之后才被加入到文件的区段树中，从而避免了两次写入的开销。在实现中，我们直接将前文中提到的额外信息增加在 `ext4_map_blocks` 这个结构中。由于这个结构原本就是 `ext4_map_blocks` 的参数，因此我们不需要修改所涉及的函数的接口，避免了过多的代码修改。

不可伸缩的孤立 inode 链表 Ext4 维护了一个全局的孤立 inode 链表，用来临时保存那些被删除或文件大小即将改变的文件 inode。若在这些文件被删除或者大小修改被持久化之前，发生崩溃情况，Ext4 可以根据这个全局孤立 inode 链表找到这些文件，以取消或继续完成此前的操作。在文件追加写入时，文件的大小会增加，因此此过程中需要使用到孤立 inode 链表：在追加操作开始之前，文件的 inode 需被插入到全局的孤立 inode 链表中，在追加操作完成之后，该 inode 被从链表中删除。由于孤立 inode 链表全局只有一份，Ext4 使用一个自旋锁（Spinlock）来保护对链表的访问。正是该自旋锁导致 Ext4 在多线程下的性能无法扩展。这个问题在此前的 FxMark 论文[113] 中也有被提及，但该工作并未提出解决方法。

补丁：孤立 inode 文件（+Of）

为了缓解全局孤立 inode 链表造成的资源竞争，Linux 内核开发者 Jan Kara 曾在 2015 年提出了一个补丁[138-139]。该补丁引入了一种新的特殊文件，称为孤立 inode 文件（Orphan File)。孤立 inode 文件是一个固定大小的文件，其中维护了一个可以存放 inode 号的数组。孤立 inode 文件与此前的孤立 inode 链表具有相同的功能，都可记录一组正被删除或者正在被改变大小的文件。当一个文件即将被删除或者被改变大小时，Ext4 将此文件对应的 inode 号保存在孤立 inode 文件中的一个空闲位置上，并在操作结束时将其删除。为了减少资源竞争，不同的 CPU 核心在搜索数组中的空闲的位置时，会从不同位置开始向后搜索。同时，在数组中加入和删除 inode 号时，使用原子指令完成，避免了使用锁的开销与竞争。由于孤立 inode 文件的大小固定，当孤立 inode 中不存在空闲位置时，Ext4 会回落到此前的全局孤立 inode 链表的方法，不过这种情况的发生概率相对较小。我们将此前的孤立 inode 文件补丁移植到新版本内核中的 Ext4 之中，以解决在非易失性内存上 Ext4 在修改文件大小时的性能和可伸缩性问题。

最初当我们发现全局孤立 inode 链表的性能和可伸缩性问题时，我们考虑为每个 CPU 维护一个单独的孤立 inode 链表，并使用多个用于占位的 inode 结点（Dummy Inode）将不同 CPU 上的孤立 inode 链表串联在一起。这样可以在不改变原有链表大结构的前提下，解决可伸缩性问题。然而当我们在邮件列表中发现孤立 inode 文件的补丁之后，我们决定使用现有的孤立 inode 文件补丁。这主要有两方面的原因：首先，虽然孤立 inode 文件补丁未被合入到 Ext4 的主线之中，但其在 e2fsprogs 中的补丁部分已经被标记为接受（Accepted），因此，我们认为社区对此补丁依然是比较接受的态度。其次，孤立 inode 文件补丁相对于我们的多链表方法，更加简单和直观，更容易维护。基于这两点，尽管孤立 inode 文件的补丁实现需要占用一个预留的 inode 号（9 号 inode），并且修改了 `e2mkfs` 和 `e2fsck` 等



(a) 在不同文件中覆盖写入 4 KB 数据　　(b) 在共享文件中覆盖写入 4 KB 数据

图 4–7 使用传统文件系统和新型文件系统在非易失性内存上的不同文件和共享文件中覆盖写入 4KB 数据的吞吐量对比。传统文件系统的性能与 NOVA 相似，或比 NOVA 性能更高。两个测试在单线程时是一样的

工具，但我们依然选择使用该补丁来优化 Ext4 在非易失性内存上的性能。

除了 4 KB 的数据追加写入之外，我们还测试了 4 KB 的数据覆盖写入。我们测试了两种情况，多个线程在不同的文件中进行覆盖写入（图 4–7(a)），以及多个线程覆盖写入同一个文件中的不同位置（图 4–7(b)）。与数据追加不同，在数据覆盖写入测试中，NOVA 的吞吐量与传统文件系统相似，甚至在一些情况下，传统文件系统的性能表现更好。因此，我们跳过了对数据覆盖写入的进一步分析。

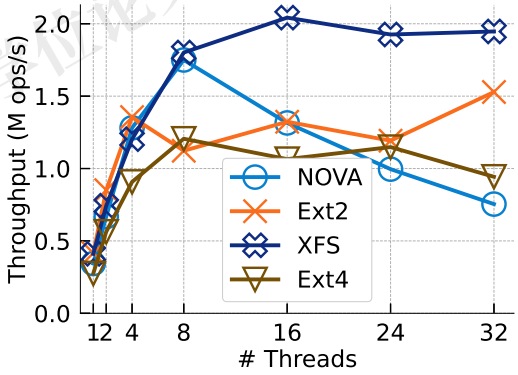


图 4–8 不同文件系统在非易失性内存上同步地覆盖写入 4 KB 数据的吞吐量对比

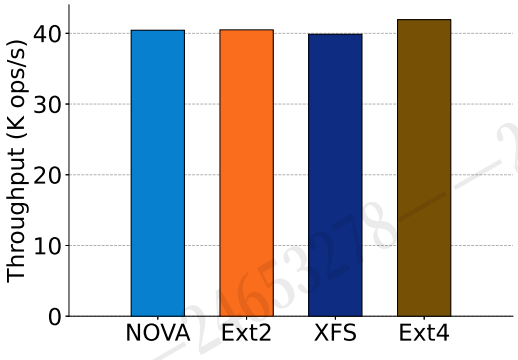


图 4–9 在非易失性内存上的不同文件系统中使用 mmap 接口的 pmemkv 性能对比

低效的日志提交 图 4–8 中展示了不同文件系统在非易失性内存上进行同步数据覆盖写入的性能。每个同步数据覆盖写入操作包括一个 4 KB 的数据覆盖写入（`write`），以及一个 `fsync` 系统调用。我们可以与图 4–7(a) 中的 4 KB 数据覆盖写入操作相对比。可以看出，NOVA 的性能受额外的 `fsync` 系统调用影响最小。这是由于其在文件写入的实现中已经保证了文件系统数据和元数据修改的持久性，因此其在处理 `fsync` 系统调用时什么都不需要做。此处微小的性能差距来自于 `fsync` 系统调用中进行的上下文切换等开销。

其他文件系统的性能则受 `fsync` 操作影响巨大。XFS 在两个测试（数据覆盖写入和同步数据覆盖写入）中均表现出了最高的性能和最好的可伸缩性，然而，同步写入中多出的 `fsync` 操作在 16 线程是导致 XFS 的吞吐量仅为非同步操作的一半。Ext4 的吞吐量同样受 `fsync` 操作影响极大，在有 `fsync` 的同步测试中，其最高吞吐量也将近减半。

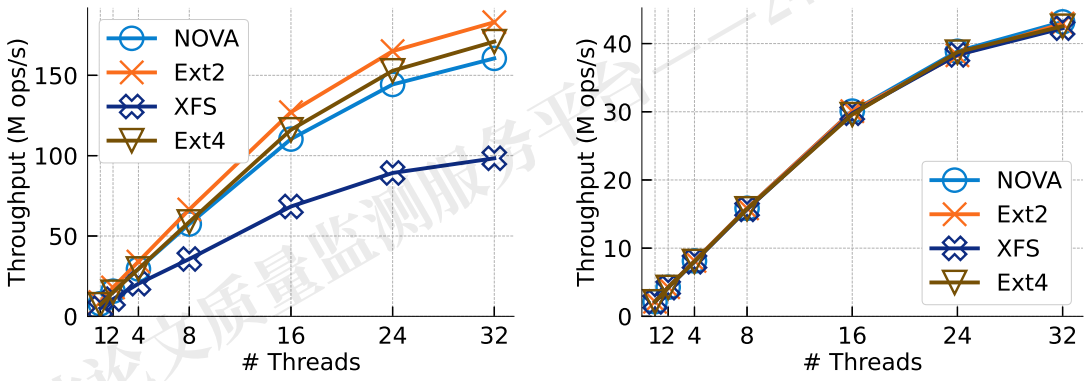
补丁：快速提交（+Fc）

我们发现在最近的 Linux 5.10 的预发行（Release Candidate）版本中，已经具有了一个名为快速提交（Fast Commit）的补丁[140-141]，可以用于优化 `fsync` 操作。

我们将为非易失性内存优化过的快速提交的补丁①移植到我们所使用的 Ext4 之中，以提升 Ext4 在进行 `fsync` 操作时的性能。同时，为了公平地进行对比，在进行最终测试时，我们会将此补丁与我们提出的其他优化区分开。

4.3.3 内存映射操作的性能分析

除了使用文件读写接口（`read` 和 `write`）之外，应用程序还可以使用文件内存映射（`mmap`）机制将非易失性内存映射到用户态空间进行直接使用。为了测试这种情况下不同文件系统的性能差距，我们使用 RocksDB 自带的测试集中的 *fillrandom* 负载，在一个基于文件内存映射机制的非易失性内存键值存储系统 pmemkv[142] 上进行测试。图 4–9 中展示了测试结果：不同文件系统在测试中表现出了非常接近的性能。这是因为 pmemkv 使用文件内存映射的方式访问非易失性内存，此方式旁路了底层的内核文件系统。除了在调用 *mmap* 时和处理缺页中断之外，内核文件系统并不在 pmemkv 的执行路径中，因此文件系统的区别对 pmemkv 的性能影响较小。



(a) 目录中的文件列举　　(b) 路径查询

图 4–10 不同文件系统在非易失性内存上“列举目录中的文件”和“路径查询”（打开一个五层深的文件路径）的吞吐量对比。Ext4 比 NOVA 的性能稍高

4.3.4 文件查询操作的性能分析

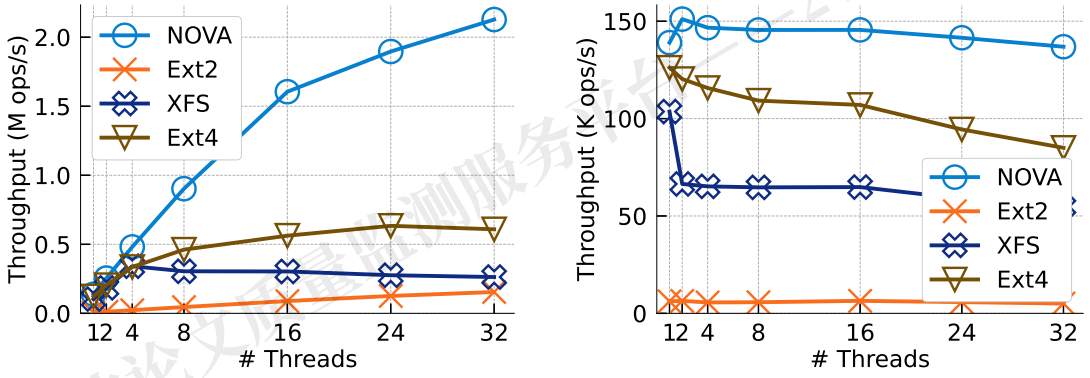
图 4–10 展示了列举目录中文件和路径查询的吞吐量。在目录列举测试中，Ext2 和 Ext4 都比 NOVA 展现出了更好的性能；而在路径查询测试中，所有的文件系统都表现出了相近的性能。这主要是由于虚拟文件系统的目录项缓存（dcache）在内存中缓存了访问过的目录项结构。大多数文件查询请求都在目录项缓存中命中，从而直接被完成；只有少数查询请求在目录项缓存中未命中，才需要文件系统进行处理。在进行文件创建时，同样会进行目录中的文件查询，这类查询将在章节 4.3.5 中分析文件创建时进行讨论。

4.3.5 文件创建操作的性能分析

为了测试不同文件系统在非易失性内存上创建文件的性能，我们使用多个线程在各自不同的目录（图 4–11(a)）以及在同一个目录（图 4–11(b)）中不停地创建空文件。在这两个测试中，NOVA 均取得了最高的吞吐量和可伸缩性，远超过各种传统文件系统。为了分析为何非易失性内存创建文件的性能不如 NOVA，我们分别在单线程测试下和 16 线程下，将一个文件创建操作的时延进行分解。结果展示在图 4–12 中。

低效的目录查找　在创建新文件过程中，文件系统需要首先确认在目标目录中是否已经存在同名文件。如果存在则文件创建操作失败，直接返回错误代码给应用

① https://github.com/harshadjs/linux/tree/fc-pmem-renewed



(a) 在不同目录中创建文件的吞吐量　　(b) 在同一个目录中创建文件的吞吐量

图 4–11 使用传统文件系统和新型文件系统在非易失性内存上的不同目录中和同一个目录中创建文件的吞吐量对比。NOVA 的吞吐量始终超过传统文件系统。两个测试在单线程情况下是相同的

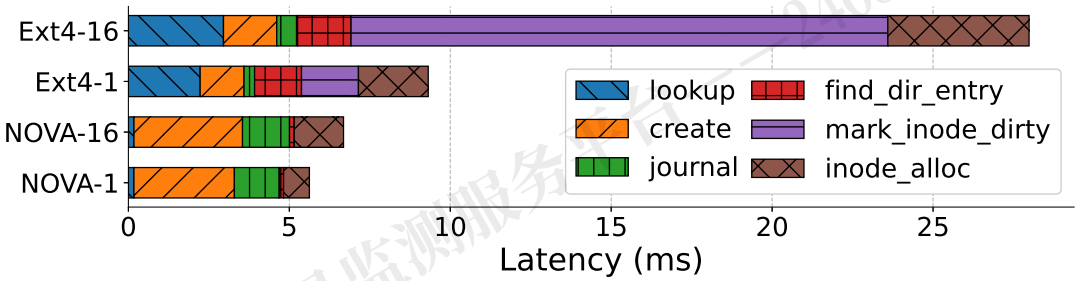


图 4–12 在单线程（图中 “-1” 后缀）和 16 线程（图中 “-16” 后缀）情况下，一个文件创建操作的时延分解

程序。这部分操作在图 4–12 中表现为 *lookup* 部分。在单线程测试中（*Ext4-1* 和 *NOVA-1*）Ext4 的路径查找部分（即 *lookup*）明显比 NOVA 消耗了更多的时间。Ext4 使用哈希树（H-Tree）组织大目录中的目录项信息。哈希树是 B 树的一个变种，其中间节点使用哈希值作为索引，而叶子节点中依然使用线性方式保存目录项信息。尽管哈系树比使用常规文件的存储结构线性保存目录项要更加高效，但是其基于块设备而设计的结构以及在叶子节点中的线性查找依然比不上 NOVA 在普通内存中维护的以文件名的哈希值为键的红黑树的查找性能。

优化：目录项全缓存（+L）

受到 NOVA 等非易失性内存文件系统在普通内存中保存索引的启发，我们为 Ext4 中每个被使用了的目录维护了一个普通内存中的目录项全缓存。在目录项全

缓存中，我们维护了目录中文件名（使用文件名的哈希值进行快速索引）到其 inode 号码的映射，因而目录查找操作可以在内存中高效完成。我们维护的目录项全缓存与虚拟文件系统中的目录项缓存 dcache 不同。虚拟文件系统中的目录项缓存是半缓存，如果在目录项缓存中未找到某个文件名，并不能说明该目录中不存在此文件名，而是需要去存储设备中的目录中进一步查找；而在我们维护的目录项全缓存中，如果未找到某个文件名，则该目录中一定不存在该文件名。为了避免消耗过多的内存，我们使用动态的方式构建和销毁每个目录中的目录项全缓存。

低效的目录项分配 在目录中创建文件时，在确定目录中不存在同名文件之后，Ext4 还需要在哈希树中找到一个空闲的足够大的位置，将新的目录项信息写入到空位之中。在进行寻址时，Ext4 需要从哈希树的叶子节点的开头开始，线性地向后扫描有效目录项之间的空位，直至找到足够大的空位。此过程十分低效，在图 4–12 中被标记为 *find_dir_entry* 部分。

优化：目录项分配提示（+A）

为了提升目录项分配的速度，我们为每个目录维护了一个目录项分配缓存（Dirent-Allocation Cache）。在此缓存中，我们为目录哈希树中的每个叶子节点记录了一个*开始搜索位置*（即对目录项分配的一个提示）。当 Ext4 需要在哈希树的某个叶子节点中分配目录项时，其只需要从此叶子节点的*开始搜索位置*开始往后进行搜索即可，无需从叶子节点的最开头开始搜索。当 Ext4 在某个叶子节点中成功分配了一个目录项空位后，该叶子节点的*开始搜索位置*被更新为该被分配的目录项的末尾。而当 Ext4 在某个叶子节点中删除了一个目录项之后，若被删除的目录项的位置比当前*开始搜索位置*更靠近叶子节点开头，则*开始搜索位置*被更新为此时被删除的目录项的开头。当扫描到叶子节点末尾时，Ext4 需要回到叶子节点的开头继续搜索，直到重新回到*开始搜索位置*时，才表示此叶子节点中的可用空间不足，Ext4 可以进行哈希树的拓展操作。

通过使用*开始搜索位置*作为目录项分配的提示，在文件创建过程中，Ext4 有非常大的概率直接通过*开始搜索位置*找到可以存放新目录项的空位，因而避免线性扫描叶子节点的性能开销。

多余的 inode 修改同步 另外一个造成 Ext4 性能表现不佳的原因是 *mark_inode_dirty*。由于 Ext4 依然使用页粒度的缓冲区管理文件系统中的元数据，当一个 inode 结构中的元数据发生变化时，Ext4 需要调用 `ext4_mark_inode_dirty` 函数将虚拟文件系统 inode 结构中的修改同步到用于写回存储设备的缓冲区中。此后再将缓冲区中的内容（以块为粒度）写回到

存储设备。相对来说，NOVA 在更新文件系统元数据时，直接在非易失性内存上进行，因此不会有类似操作造成性能开销。

由于 Ext4 所使用的日志机制 JBD2 会等待所有正在运行的日志项结束之后，才会进行日志事务提交操作，每个虚拟文件系统的 inode 结构上的修改只需要在日志项结束之前被同步到相应缓冲区中即可。然而 Ext4 中的大多数函数在更新完虚拟文件系统的 inode 结构之后，都会主动的调用一次 `mark_inode_dirty` 将虚拟文件系统 inode 结构中修改同步到缓冲区之上。结果导致在同一个日志项的运行过程中（即在 `journal_start` 到 `journal_stop` 之间），Ext4 可能会多次调用 `mark_inode_dirty`，延长了操作的时延，造成不必要的性能开销。

优化：延迟的 inode 同步操作（+D）

为了去除多余的虚拟文件系统 inode 结构修改的同步操作，我们跟踪每个日志项在运行过程中对虚拟文件系统 inode 的修改操作，并将需要进行同步的 inode 结构在日志项即将结束前一次性、统一地同步到其对应的缓冲区之中。

缓冲区伪共享 除了多余的 inode 修改同步操作之外，`ext4_mark_inode_dirty` 同时还带来了资源竞争问题。与单线程相比，在 16 线程时 Ext4 的 `ext4_mark_inode_dirty` 操作的时延显著增高，说明在该函数执行过程中存在资源竞争问题，且资源竞争问题影响了该函数的可伸缩性。资源竞争问题来源于 Ext4 中的两个设计：第一，为了将虚拟文件系统 inode 结构中的修改同步到缓冲区之中，Ext4 首先需要增加该缓冲区所在页的引用数，之后才能使用该缓冲区。而缓冲区所在内存页的引用数被一个自旋锁所保护，因此，每次修改该引用数时，需要首先获取该自旋锁。第二，由于 Ext4 中每个 inode 结构在存储设备上的大小远小于一个内存页，因此 Ext4 将多个相邻的 inode 结构存放在同一个页之中，以避免空间浪费。基于这两点设计，当多个处于同一个缓冲区页面中的 inode 结构同时需要进行数据同步时，多个线程会在增加缓冲区页面的引用数时在自旋锁上发生访问竞争，从而对 inode 结构的同步操作的性能产生影响。在我们的具体测试中，多个线程不断创建文件。由于 Ext4 在同一个块组（Block Group）中会顺序地分配 inode，因此，多个线程分配得到了位置相邻的 inode 结构。这些 inode 结构大多都使用同一个缓冲区页面用于写回，因此，当多个测试线程同时调用 `ext4_mark_inode_dirty` 函数以完成文件创建时，这些线程会由于在缓冲区页面的自旋锁上的访问竞争而极大地延长 `ext4_mark_inode_dirty` 的完成时间。

优化：分离的 inode 分配（+S）

在前面的描述中，我们提到多个线程会在同一个块组中分配 inode 结构，这是由于 Ext4 对非顶层目录中文件的分配策略是将新的文件与其父目录放在同一个

块组之中。这种分配策略有利于将相关的文件分配在同一个块组中，可以提升文件存储结构的空间局部性（Spatial Locality）。这对于机械磁盘等需要尽量保持顺序访问的存储设备非常有利。而由于非易失性内存的顺序访问时延和随机访问时延相似[143]，因此保持这种空间局部性对于非易失性内存文件系统来说意义不大。而由于这种 inode 分配策略是造成文件创建伸缩性问题的众多原因中的一环，我们改变了 Ext4 在非易失性内存上的 inode 分配策略，以缓解文件创建中的资源竞争。我们为在非易失性内存上的 Ext4 实现了一种新的分配策略：每个线程在为新文件分配 inode 结构时，不用关心父目录所在的块组，而是根据其当前运行在的 CPU 核心的编号，选择对应的块组进行 inode 分配。不同的 CPU 核心使用不同的块组，因此运行在不同 CPU 核心上的线程会在不同的块组中分配 inode 结构，所分配出的 inode 之间也不会共享缓冲区页面，从而不会产生前述的访问竞争问题。

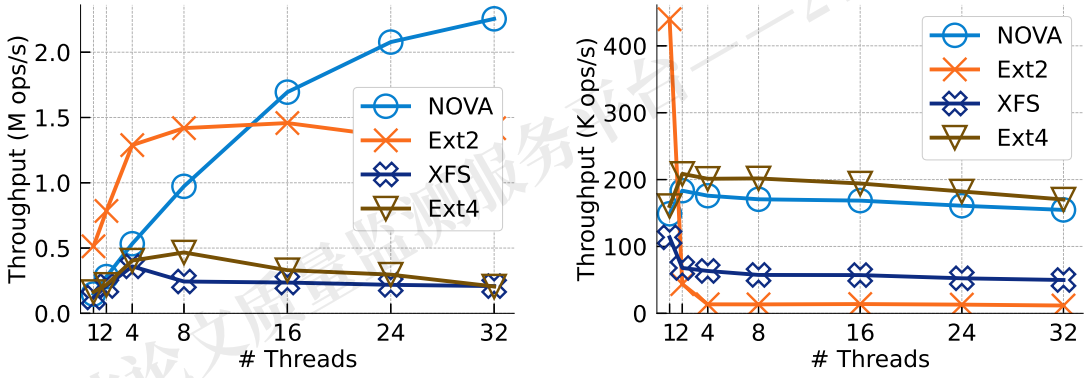
我们的新分配策略与 Ext4 中现有的 Orlov 分配策略[144-146] 类似。Orlov 分配策略会根据文件名的不同，将目录中的文件分散在不同的块组之中，被默认用于 Ext4 顶层目录中的文件分配①。相比于 Orlov 分配策略，我们没有使用文件名，而是直接使用 CPU 核心的编号，对于多核可伸缩性的提升更加直接。

日志空间耗尽 根据我们此前的分解分析，我们并未看到 *journal* 部分是一个明显的性能瓶颈。然而，当我们将前述其他瓶颈进行优化之后，*journal* 部分的性能开销开始逐渐显现，并在多线程情况下成为了影响 Ext4 性能最主要的因素。

通过在日志机制的实现中插入一些调试点，我们发现在测试开始之后，日志空间很快被耗尽，而此后所有的线程在创建新的日志项时，需首先等待后台的日志处理线程完成对检查点的创建以释放出更多的可用日志空间。由于这种情况只有在其他性能瓶颈被优化的足够快之后才会显现，我们在图 4–12 中并未展示此部分。

优化：扩大日志空间（+J）

尽管可以通过优化日志创建检查点的性能以解决日志空间耗尽的问题，但是我们发现这个问题可以通过更简单的扩大日志空间的方法解决。我们在平台中进行了测试，发现在我们的配置中，将日志空间设置成 8 GB 就可以避免在大量元数据修改的情况下由日志空间耗尽造成的等待。



(a) 在不同目录中删除文件的吞吐量　　(b) 在同一个目录中删除文件的吞吐量

图 4–13 使用传统文件系统和新型文件系统在非易失性内存上的不同目录中和同一个目录中删除文件的吞吐量对比

4.3.6 文件删除操作的性能分析

当多个线程在不同目录中进行删除文件操作时（图 4–13）。Ext2 在 8 个线程之前保持着最高的吞吐量，然而 NOVA 具有更好的可伸缩性，其吞吐量随着线程数增加而不断增加，在线程数超过 8 之后，吞吐量超过 Ext2。相比而言，Ext4 和 XFS 两个传统的文件系统吞吐量较低，且随着线程数增加，吞吐量也没有大幅提升。当多个线程在相同的目录中删除文件时（图 4–13(b)），父目录中的互斥锁成为了性能瓶颈，Ext4 比 NOVA 的吞吐量稍有优势，但所有文件系统的吞吐量均无法随线程数而拓展。

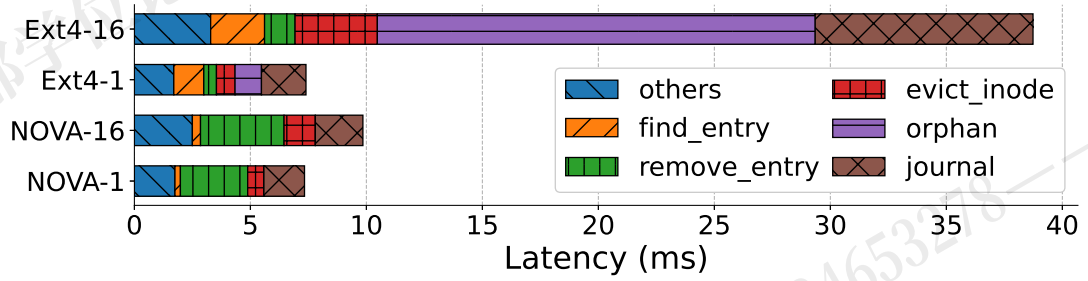
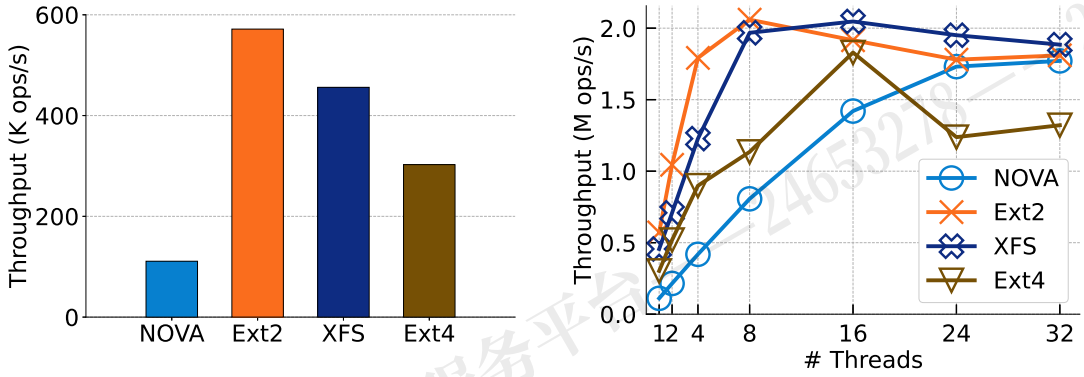


图 4–14 在单线程（图中“-1”后缀）和 16 线程（图中“-16”后缀）情况下，一个删除文件操作的时延分解。全局孤立 inode 链表和日志操作在 16 线程时时延明显增加，性能无法随线程增加而拓展。

① 虽然用户可以通过 `chattr` 命令强制在一个目录中使用 Orlov 分配策略，但是用户需要预先对该目录中所存放的文件的特性有所了解，才能决定是否应该使用 Orlov 分配策略。

为了找出在不同目录中删除文件时 Ext4 的性能瓶颈，我们再次进行了时延分解，在图 4–14 中给出了一个文件删除操作中各部分的时延。对于单线程测试来说，Ext4 的时延与 NOVA 时延相似，这也与图 4–13(a) 中的结果吻合。然而在 16 线程的测试中，Ext4 中删除操作的时延（*Ext4-16*）显著地长于 NOVA（*NOVA-16*）。造成这个结果的主要有两部分：全局孤立 inode 链表操作（*orphan*）和日志操作（*journal*）。由于这两个部分的文件在进行文件创建操作时均已给出了分析，此处不在赘述。

4.3.7 重命名操作性能分析



(a) 重命名操作的吞吐量（单线程）　　(b) 重命名操作的吞吐量

图 4–15 使用传统文件系统和新型文件系统在非易失性内存上的不同目录中移动文件（即重命名）的吞吐量对比

图 4–15 中展示了将文件从一个目录中重命名到另外一个目录中的性能，在单线程和多线程的情况下，传统文件系统的性能均超过 NOVA 或与 NOVA 相当，因此我们跳过对 Ext4 重命名操作的时延分解分析。

4.3.8 优化总结与修改量统计

为了保持 Ext4 现有的各种功能和特性，我们在进行优化时，在不修改 Ext4 的存储格式的情况下，将修改量控制在最少。表 4–2 中给出了每个优化和补丁的代码修改量。

除去两个第三方补丁（快速提交和孤立 *inode* 文件）之外，所有的优化共增加了 854 行代码，减少了 29 行代码。修改总量远小于 Ext4 现有的总代码量（不计 JBD2 部分，约为 4 万行）。此外，我们增加了 92 行代码用于控制各个优化是否启

表 4–2 每个优化和补丁修改的代码行数统计

优化和补丁名	增加的代码行数	删除的代码行数
乐观的机器异常安全的内存拷贝	184	12
使用简单高效的接口	364	2
使用有效数据初始化文件块	70	0
目录项全缓存	105	7
目录项分配提示	63	2
延迟的 inode 同步操作	57	2
分离的 inode 分配	11	4
扩大日志空间	0	0
优化部分修改之和	854	29
控制代码	92	0
快速提交	4403	169
孤立 inode 文件	263	77

用。而在两个第三方补丁中，已经合并到上游分支的*快速提交*补丁包括约 4 千行代码修改，*孤立 inode 文件*补丁则有大概 200 行代码。在我们提出的优化中，部分优化，例如*使用有效数据初始化文件块*，只有几十行代码修改，却可以令 Ext4 在非易失性内存上的性能提升极大。

4.4 性能测试

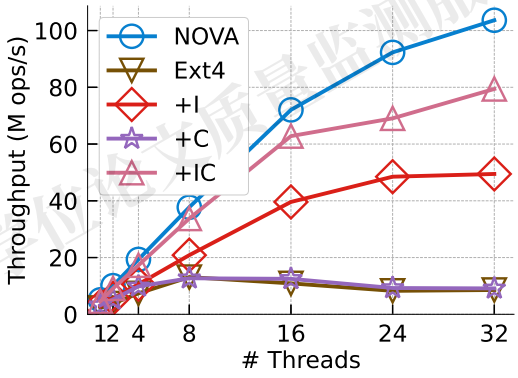
在本章节中，我们将对我们优化过的 Ext4 进行测试，以回答下列问题：

- 在非易失性内存上，传统文件系统（Ext4）的性能是否可以被优化到与新型非易失性内存文件系统（NOVA）相同?
- 本工作中提出的各种优化方案对 Ext4 的性能提升有何效果?
- 本工作中提出的这些优化在综合负载测试中是否依然有效?
- 本工作中提出的这些优化是否可以提升运行在文件系统上真实应用程序的性能?

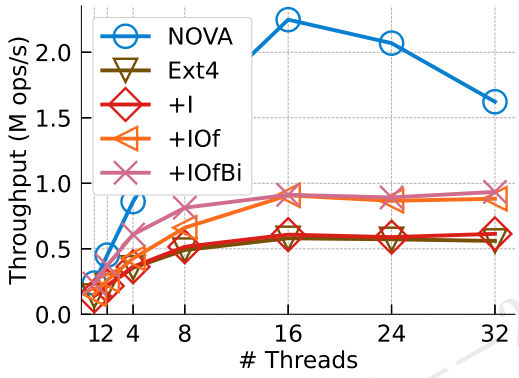
我们在测试中使用的实验平台和环境与章节 4.3 中的配置相同，故不再赘述。

4.4.1 微基准测试

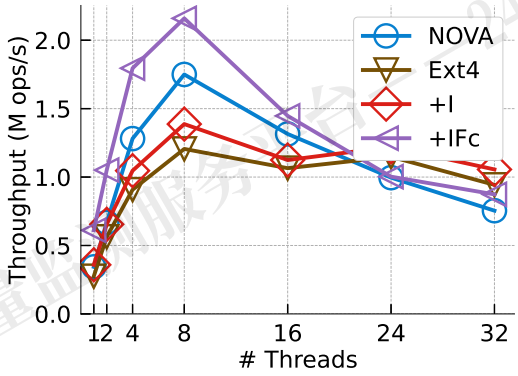
在本小节中，我们将此前章节 4.3 中进行过的 FxMark 测试重新测试一次，并将有代表性的测试负载进行分解分析，以检验我们提出的优化的效果。



(a) 读取 4 KB 文件数据的吞吐量



(b) 追加写入 4 KB 文件数据的吞吐量



(c) 同步地覆盖写入 4 KB 文件数据的吞吐量

图 4–16 传统文件系统和新型文件系统在非易失性内存上进行数据操作的吞吐量

文件数据访问优化的效果 图 4–16 中给出了不同文件系统在非易失性内存上进行 4 KB 文件数据读取、追加和同步写入操作的性能。对于每个测试负载，我们将与其相关的各种优化逐个加入，以分别表现出各个优化的效果。

图 4–17 中展示出三种文件数据访问下单线程的吞吐量。我们从图中可以看出使用简单高效的接口优化（图中的 *+I*）和乐观的机器异常安全的内存拷贝优化（图中的 *+C*）可以提升 Ext4 的数据读取性能。同时，当这两个优化同时被应用时（图中的 *+IC*），其所达到的性能提升，比两种优化各自的性能提升之和更高。在

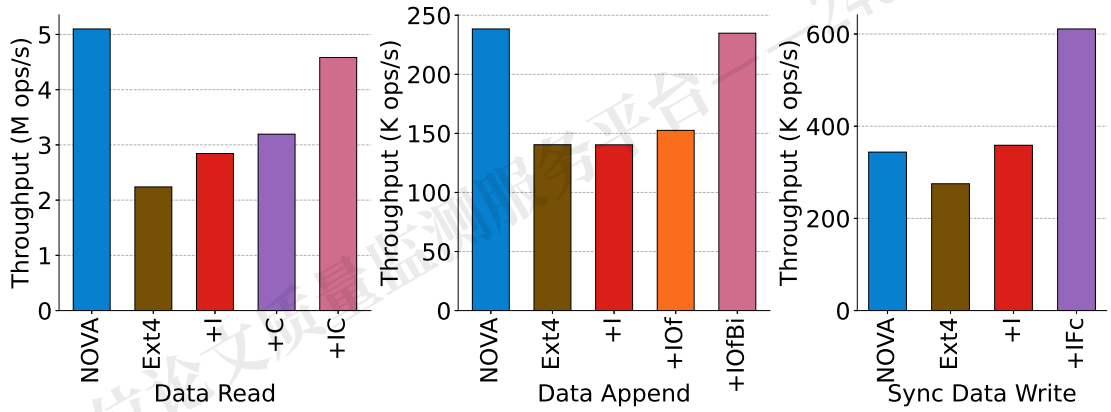


图 4–17 数据读取、追加写入和同步覆盖写入三个测试中的单线程吞吐量

应用了这两个补丁之后，Ext4 的单线程数据读取的吞吐量被提升了 105%，只比 NOVA 相差 10% 左右。

对与图 4–16(a) 中所展示的多线程场景中，使用简单高效的接口优化（*+I*）提高了 Ext4 在多线程下的可伸缩性，然而的乐观的机器异常安全的内存拷贝优化（*+C*）却几乎与原本的 Ext4 性能相同。两个优化的组合方案（*+IC*）的吞吐量在多线程情况下比原有 Ext4 提升最多达 827%，比只应用了乐观的机器异常安全的内存拷贝优化（*+C*）的 Ext4 提升最多到 770%。这种“一加一大于二”的现象源于优化过程中的性能瓶颈的变化。复杂的抽象接口在最初限制了 Ext4 的读取文件数据的性能，因而此时优化机器异常安全的内存拷贝的效果并不明显；而当我们在使用了更加简单高效的接口之后，Ext4 读取文件数据时的性能瓶颈转变为机器异常安全的内存拷贝，此时再使用高效乐观的机器异常安全的内存拷贝，则可以看到明显的吞吐量提升。在超过 16 线程之后，使用了两个优化的 Ext4 的吞吐量提升与 NOVA 相比依然有所差距，因此 Ext4 的吞吐量和可伸缩性依然有被进一步提升的空间。

在数据追加写入操作的单线程测试中（图 4–17），使用简单高效的接口优化（*+I*）带来的性能提升非常有限；孤立 *inode* 文件优化（*+Of*）将单线程吞吐量提高了 9%。在同时应用了使用有效数据初始化文件块优化之后的组合方案（*+IOfBi*）中，Ext4 的吞吐量已经与 NOVA 的吞吐量相同。然而尽管在单线程测试中可以达到相同的性能，组合优化方案（*+IOfBi*）却依然无法像 NOVA 一样随线程增多而展现出足够好的可伸缩性。图 4–18 中给出了 Ext4 优化前后的单线程和 16 线程在进行 4 KB 数据追加写入时的时延分解。单线程情况下，应用了所有优化的 Ext4（图中的 *Ext4+all-1*）时延与 NOVA 的单线程使用非常接近；而在 16 线程下，尽管孤立 *inode* 文件优化已经基本消除了孤立 inode（*orphan* 部分的可伸缩性问题，同时使

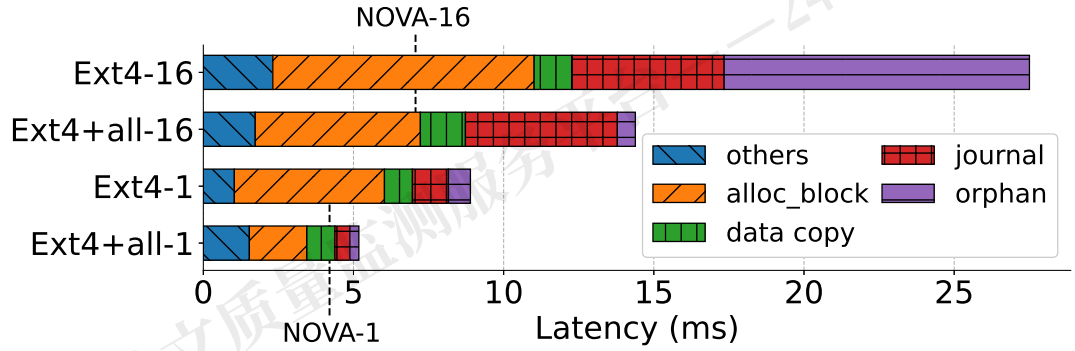


图 4–18 数据追加写入操作中各个优化对时延的优化效果分解

用简单高效的接口优化减小了文件块分配（*alloc_block*）部分的时间，*Ext4+all-16* 的时延依然是 *NOVA-16* 的两倍左右。

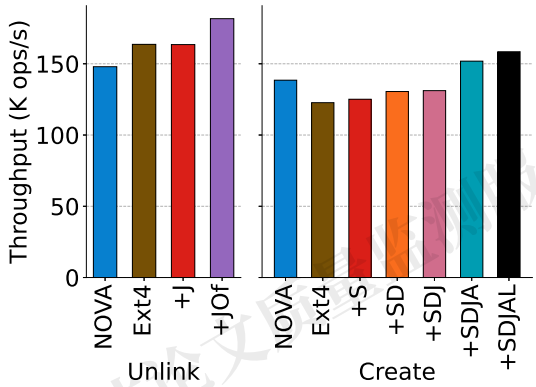
对于同步数据覆盖写入测试，只应用了使用简单高效的接口优化（+*I*）之后，Ext4 的单线程吞吐量已经略高于 NOVA，但其多线程测试下的吞吐量依然不如 NOVA。在同时应用了快速提交优化（+*Fc*）之后，组合方案（+*IFc*）已经全面超过了 NOVA，吞吐量超过 NOVA 最高达 78%。

元数据优化效果 图 4–19 给出了在依次应用各个优化之后 Ext4 元数据操作的吞吐量。多项优化可以提升创建文件操作的性能，包括分离的 *inode* 分配优化（+*S*）、延迟的 *inode* 同步操作优化（+*D*）、扩大日志空间优化（+*J*）、目录项分配提示优化（+*A*）和目录项全缓存优化（+*L*）。

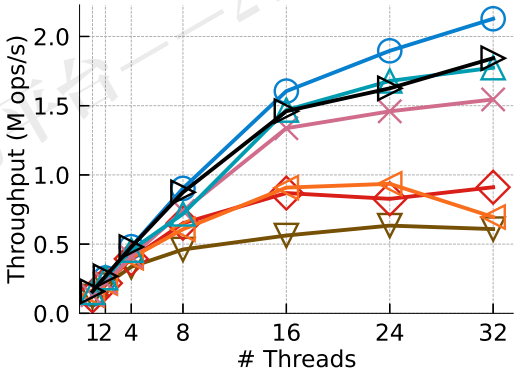
图 4–19(a) 中的单线程测试显示出，逐步应用这些优化可以逐步提升 Ext4 创建文件的吞吐量，最终 Ext4 的吞吐量超过 NOVA 达 14%。在众多优化中，在应用目录项分配提示优化（+*A*）之后的性能提升最明显。图 4–19(b) 中对应的多线程测试结果显示，在不超过 4 个线程的时候，优化过的 Ext4 的吞吐量（+*SDJAL*）始终比 NOVA 要更高，在超过 4 个线程之后，NOVA 的吞吐量反超优化过的 Ext4，但性能差距始终保持在 15% 以内。当多个线程在一个共享目录中创建文件时（图 4–19(b)）性能情况与在不同目录中创建类似。

图 4–20 中进一步展示了各个优化如何降低创建文件操作的时延。时延中的多个耗时部分被极大地优化，如目录查找（*lookup*）部分、目录项分配（*find_dir_entry*）部分和 *mark_indoe_dirty* 部分。因而 Ext4 在非易失性内存上可以达到与 NOVA 相似甚至比 NOVA 更好的性能。

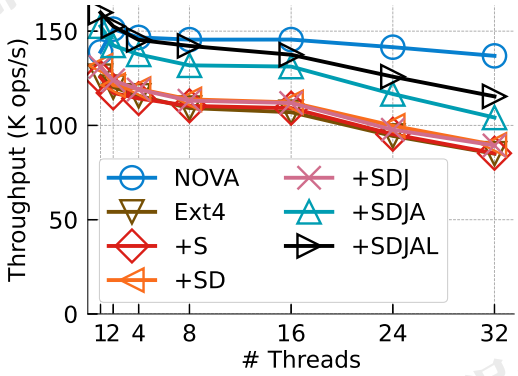
对于文件删除操作，原有的 Ext4 已经在单线程测试中超过了 NOVA。扩大日志空间优化（+*J*）和孤立 *inode* 文件优化（+*S*）进一步提升了 Ext4 的吞吐量，并



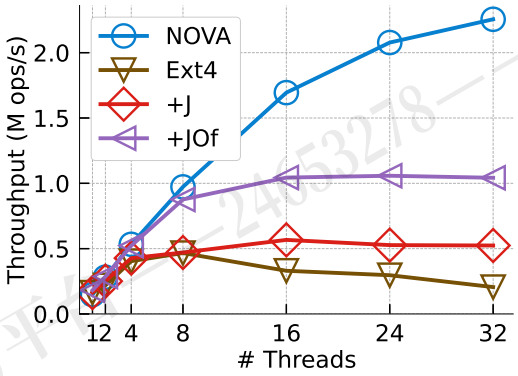
(a) 创建和删除文件操作的单线程吞吐量



(b) 在不同目录中创建文件的吞吐量



(c) 在相同目录中创建文件的吞吐量



(d) 在不同目录删除文件的吞吐量

图 4–19 传统文件系统和新型文件系统在非易失性内存上进行元数据操作的吞吐量

让 Ext4 在多线程测试中（图 4–19(d)）吞吐量可以一直提升至 8 个线程，但超过 8 个线程之后优化的 Ext4 性能（*+JOf*）不再增加。我们将如何在此情况下进一步对 Ext4 进行优化留作未来工作。

4.4.2 Filebench

为了进一步展示传统文件系统在非易失性内存上的性能提升，我们进一步测试了 Filebench 测试集中的四个综合负载，包括 FileServer、Varmail、WebServer 和 WebProxy。我们使用三种 Ext4 与 NOVA 进行对比。在 Linux 5.8.1 版本中原生的 Ext4（*Ext4*）基础上，*+P* 额外增加了两个第三方补丁快速提交（*+Fc*）和孤立 *inode* 文件（*+Of*）。*+PC* 在 *+P* 的基础上，应用了乐观的机器异常安全的内存拷贝优化（*+C*）；*+all* 则进一步将所有其他优化应用到 Ext4 中。在下文的测试中，我们将以 *+PC* 作为默认的对比基准。

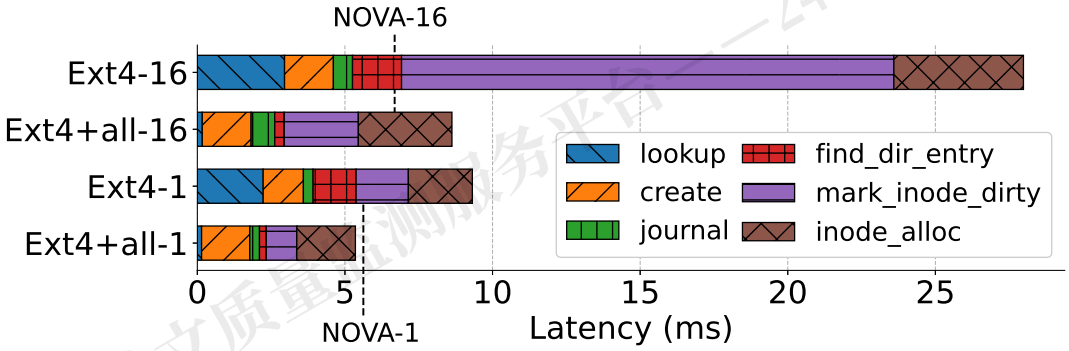


图 4–20 创建文件测试中各个优化对于操作时延的优化效果分解

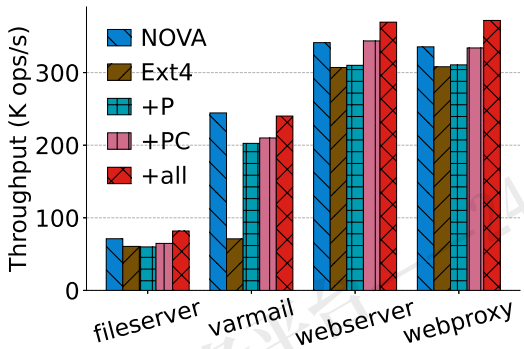


图 4–21 在 Filebench 各个负载上不同文件系统的单线程吞吐量。应用了所有优化之后 Ext4 在单线程测试中性能超过 NOVA

图 4–21 展示了 4 个综合负载的单线程吞吐量。应用了所有优化的 Ext4（*+all*）比基准（*+PC*）分别提升了 26%、13%、8% 和 11%。与 NOVA 进行比较，*+all* 在 FileServer 负载测试中提升了 15%，在 WebServer 负载测试中提升了 8%，在 WebProxy 负载测试中提升了 11%。在 Varmail 中，*+all* 的吞吐量只比 NOVA 小 2.16%。在 Varmail 负载中有频繁的 `fsync` 系统调用，因此在 *+P* 中开始引入的快速提交优化带来了最大的性能提升。在其他负载中，大部分性能提升来自于本文中提出的其他优化。

图 4–22 中给出了多线程下的测试结果。总体来看，*+all* 表现出了与 NOVA 相近或者更好的性能与可伸缩性。在 FileServer 测试负载中，具有所有优化的 *+all* 始终保持最高的性能。其性能比 *+PC* 高最多 112%、比 NOVA 高最多 27%。使用有效数据初始化文件块优化由于消除了两次写入问题，在 FileServer 负载中带来的性能提升最大。

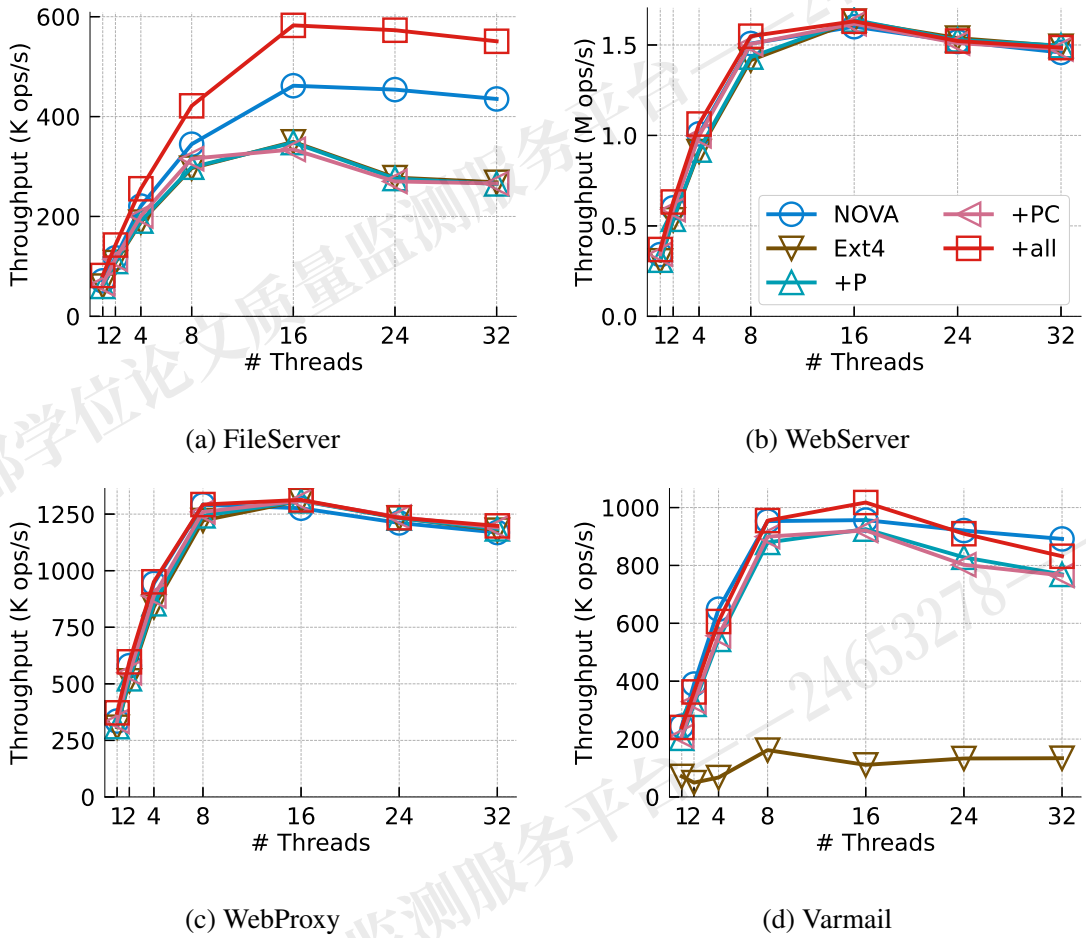
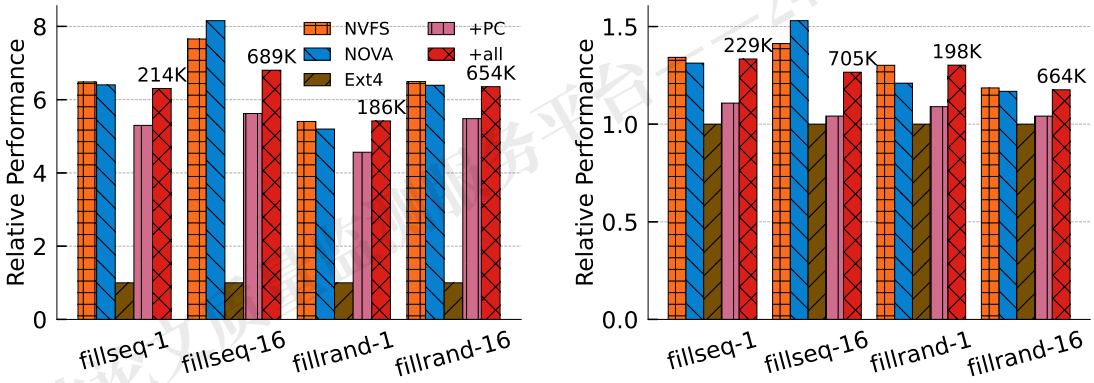


图 4–22 在 Filebench 各个负载上不同文件系统的吞吐量和可伸缩性

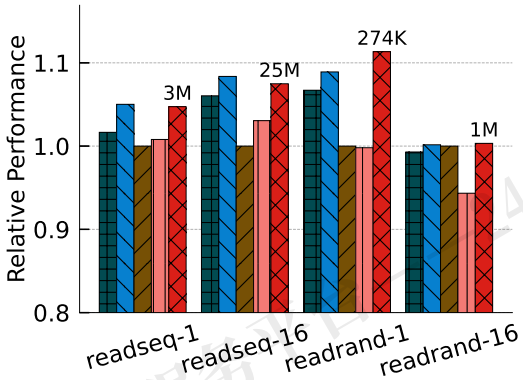
在 WebServer、WebProxy 和 Varmail 中，*+all* 的性能始终与 NOVA 相似。其中快速提交优化显著地提高了 Varmail 测试负载中 Ext4 的性能，其他优化对 Ext4 的性能提升也有所贡献。

4.4.3 RocksDB

为了表现出我们的优化在实际应用程序中的效果，我们使用一个非常流行的持久化键值存储系统 RocksDB[135, 147] 作为测试应用程序。我们使用其自带的 db_bench 测试工具进行性能测试。测试的负载包括同步版本和非同步版本的顺序插入（*fillseq*）、随机插入（*fillrand*）、顺序读取（*readseq*）和随机读取（*readrandom*）。在此测试中，除了对比 NOVA 之外，我们还引入了另外一个新型非易失性内存文件系统 NVFS[148]。NVFS 是由红帽公司员工实现，发布在 Linux 社区中的另外一个非易失性内存文件系统[149]，在一定程度上可以代表工业界研发的新型非易失



(a) RocksDB 测试的相对吞吐量（同步操作） (b) RocksDB 测试的相对吞吐量（非同步操作）



(c) RocksDB 测试的相对吞吐量（读取操作）

图 4–23 在非易失性内存上使用不同文件系统时 RocksDB 的相对性能。原生的 Ext4 文件系统实现是相对性能的基准（即其性能为 1.0）。应用了本文中所有优化的 Ext4 取得了与 NOVA 和 NVFS 两个新型非易失性内存文件系统相似或者更高的性能

性内存文件系统。

图 4–23 中给出了测试结果。与应用了第三方补丁的 Ext4（+*PC*）相比，应用了所有优化的 Ext4（+*all*）带来了平均 19% 的性能提升。而与原生的 Ext4 相比，+*all* 将 Ext4 的性能提升了最高达 6.3 倍。相比于 NOVA 来说，+*all* 在单线程测试中有平均 3.1% 的更高性能，在 16 线程测试中，+*all* 的性能大约是 NOVA 性能的 83% 到 101%。对于读取相关的负载，+*all* 的性能平均超出 +*PC* 9%，超过 NOVA 平均 4%。

4.5 其他相关工作

用户态非易失性内存文件系统 由于非易失性内存可以在用户态进行访问，近年来出现了许多用户态文件系统[38-39, 132-134, 150]。SplitFS[132] 和 Libnvmmio[133] 将文件数据映射到用户态内存地址空间中，并通过访问映射后的地址来处理文件系统的文件数据请求，以避免系统调用和虚拟文件系统开销。这些工作依赖于底层内核态非易失性内存文件系统来管理文件元数据和提供到物理非易失性内存的文件内存映射，因此，本工作在理论上也可以提升这些文件系统的使用性能。Strata[39] 将非易失性内存作为一个用户态持久日志区域，用于记录对文件系统的操作日志，并通过内核中的文件系统对操作日志进行整理，以将数据持久化到不同设备。CrossFS[150] 利用非易失性内存持久地存储文件描述符队列，以发挥实现在固件中的文件系统在执行文件系统操作时的高并发性。这些文件系统利用非易失性内存加速混合存储的性能，Ext4pm 则关注在只使用非易失性内存的场景。

文件系统性能测试 由于文件系统的复杂性和文件系统负载的多样性，对文件系统进行测试十分复杂[113, 151-154]。研究人员提供了多种测试工具[63, 113, 155-160] 用于测试文件系统的性能。FxMark[113] 关注于对文件系统的可伸缩性进行测试。其提出了一系列微基准测试工具来压力测试文件系统中的不同部分。本工作使用这些微基准测试工具检测传统文件系统在非易失性内存上运行时的性能瓶颈，并进行对应的性能优化。Filebench[63] 是一个文件系统性能测试工具。其可以根据一种特定的语法生成不同的文件系统工作负载，并对文件系统性能进行综合性测试。本工作使用 Filebench 作为综合测试对提出的优化的有效性进行测试。

文件系统可伸缩性 为了充分利用多核处理器和存储设备的高并发性，许多工作致力于提供文件系统的可伸缩性。ScaleFS[161-162] 将内存文件系统与磁盘文件系统解耦，利用内存中的高速并发数据结构处理文件请求。文件系统操作被记录在 CPU 本地的日志中，并在进行 `fsync` 操作时写回存储设备。SpanFS[119] 由一组微文件系统服务组成，并在这些文件系统服务之上提供了一个了全局视图。

在非易失性内存文件系统的可拓展性方面，pNOVA[163] 在 NOVA 的实现中引入了文件范围锁，允许同一个文件的不同部分被并发访问。FastMap[164] 通过减少同步开销优化文件内存映射访问中的可伸缩性。Xu 等人[165] 在非易失性内存上测试了文件系统的可伸缩性，并提出了非易失性内存感知的日志系统 JDD，允许细粒度的 CPU 本地撤销日志（Undo Log）。在其测试中，JDD 可以提升 Ext4 在非易失性内存上的性能，但提升后的性能依然与 NOVA 有较大差距。由于无法获得源

代码，我们并未对 JDD 进行测试，但我们认为其效果与我们所应用的快速提交补丁相似。与这些工作相比，本文中的工作注重于发现 Ext4 在非易失性内存上的性能瓶颈，并对其进行优化，在修改尽量小的情况下将 Ext4 的性能提升至与 NOVA 等新型非易失性内存相似甚至更好的程度。这些相关工作对文件范围锁、文件内存映射访问、虚拟文件系统等方面的优化，可以应用在我们优化过的 Ext4 中，进一步提升其在非易失性内存上的性能。

4.6 本章小结

新型非易失性内存文件系统尚未成熟，使用 Ext4 等成熟的传统文件系统管理新型非易失性内存是一种比较切实可靠的方案。然而即使在“直接访问”的优化模式下，传统文件系统在非易失性内存上的性能与新型非易失性内存文件系统差距依然较大，限制了非易失性内存性能的发挥。在本工作中，我们对非易失性内存上运行的传统文件系统 Ext4 进行性能测试和分析，并通过与专为非易失性内存设计的新型非易失性内存文件系统 NOVA 进行性能对比，找出了导致传统文件系统在非易失性内存上性能不佳的 10 个性能问题。我们将这些性能问题总结为三个根本原因，并提出相应的优化策略。根据这些策略，我们找到一些第三方补丁以缓解部分性能问题；而对其余的性能问题，我们为每个问题有针对性地提出了相应的优化方案。最终，我们优化过的文件系统 Ext4pm 在对 Ext4 进行不到 1,000 行修改的情况下，将使用第三方补丁优化过的 Ext4 的性能进一步提高 22%。在测试中，Ext4pm 达到了与 NOVA 相近的性能甚至更高的性能；在对真实应用 RocksDB 的测试中，Ext4pm 的性能比 NOVA 最多高 8%。

第五章　工作总结与展望

5.1　工作总结

非易失性内存是一种新型存储设备。其将传统大容量存储设备的持久性与普通内存的高性能和字节粒度寻址的特性相结合，将传统的“CPU 缓存–内存–存储”存储层次改变为“CPU 缓存–存储（非易失性内存）”，正在为计算机存储系统带来一场新的变革。研究非易失性内存的高效使用方法，能够提升系统软件和应用程序的存储性能，提高应用程序使用存储的灵活性和便捷性，对于大数据时代背景下的海量、高速数据存储功能的实现与提升具有重大意义。

非易失性内存的出现，为计算机系统中最常用的存储系统之一——文件系统提供了一个新的发展契机和方向。如何设计和实现文件系统，充分发挥非易失性内存的性能与各种特性，逐渐成为学术界和工业界的研究热点话题。新型非易失性内存文件系统研究首先在内核空间中展开。大量现有的文件系统设计方法和机制在非易失性内存中被重新设计和优化，以符合非易失性内存的访问特点，并发挥其高性能。然而，由于非易失性内存使用 LOAD、STORE 等 CPU 指令进行访问，CPU 缓存可能会将 CPU 对非易失性内存的写入乱序写回到非易失性内存之中，从而造成文件系统崩溃一致性被破坏。现有的新型文件系统通过使用同步的 CLFLUSH 等缓存行刷除指令保证文件系统崩溃一致性，导致系统调用的时延增长，降低了文件系统的性能。**如何在保证文件系统崩溃一致性的情况下，避免同步缓存刷除指令对文件系统性能的影响，是进一步提升内核态非易失性内存文件系统性能过程中的重要问题**。由于非易失性内存可以在用户态访问，用户态非易失性内存文件系统可以避免系统调用、上下文切换和虚拟文件系统的开销，也逐渐成为研究热点。然而由于用户态文件系统与应用程序处于同一个内存空间之中，为了保护文件系统的安全性、完整性和一致性，现有的用户态非易失性内存文件系统无法直接修改文件系统元数据，这导致用户态文件系统的元数据更新依然需要进入内核空间或通过通讯由可信的进程进行，影响了用户态文件系统的性能。**如何在保障文件系统安全性和完整性的同时，赋予用户态非易失性内存文件系统对元数据的完全控制，是为了完全发挥非易失性内存性能所需要研究的核心问题**。虽然大量新型非易失性内存文件系统被提出，这些新型文件系统需要多年的积累才能够逐渐成熟和被应用到实际生产之中。因此，针对非易失性内存的特点对成熟的传统文件系统进行优化，是将非易失性内存快速应用到实际生产生活中最可靠的方法之一。现有的“直接访问”优化虽然能够提升 Ext4、XFS 等成熟的传统文件系统在非易失

性内存上的性能，然而在实际应用过程中，其性能依然与为非易失性内存设计的新型内核态文件系统有较大差距。**如何探索导致性能差距的根本问题，并在尽量少改动的情况下去除现有成熟传统文件系统在非易失性内存上的性能瓶颈，是实际应用非易失性内存时需要解决的关键问题**。为了解决上述三个问题，本文对非易失性内存上的文件系统设计进行了具体研究，内容包括：

- 提出了**新型内核态异步非易失性内存文件系统 SoupFS**，通过推迟文件系统操作的持久性保证，在保证崩溃一致性的情况下，消除了关键路径上的同步缓存行刷除指令。SoupFS 结合了软更新技术与非易失性内存的特点，使用软更新技术将文件系统修改的持久化交给后台异步线程处理，从而在关键路径中消除了持久化和同步缓存刷除操作。同时，SoupFS 利用非易失性内存的可字节寻址特性，设计使用了更高效的文件系统组织结构，并与软更新技术中指针粒度的依赖性追踪相配合，简化了软更新技术中原本非常复杂的依赖追踪和依赖保证过程。为了解决软更新技术中双视图技术对页缓存的依赖，SoupFS 还提出了基于指针的双视图，在同一份结构上通过不同指针形成两种不同的视图，以支持元数据的异步持久化、依赖追踪和依赖保证。在性能测试中，SoupFS 显著地降低了文件系统操作的处理时延，提供了更高的文件系统操作吞吐量。
- 提出了**新型用户态非易失性内存文件系统框架 Treasury**，在保障文件系统安全性的同时，允许用户态文件系统直接修改文件系统元数据。通过对应用程序的数据文件进行调研，本文发现应用程序倾向于以相似的文件权限保存其文件，同时这些文件的权限很少被修改。基于此，本文提出了一个新的抽象 Coffer，用于管理一组具有相同访问权限的非易失性内存资源。通过使用 Coffer 抽象，本文设计了 Treasury 框架，将非易失性内存的保护和管理分开：内核中的模块 KernFS 负责以 Coffer 为粒度对非易失性内存进行强保护，同时赋予用户态非易失性内存文件系统 μFS 对 Coffer 内的文件系统结构（包括数据和元数据）的完全管理权。本文还结合了内存保护键等机制，解决由用户态对 Coffer 的完全管理权带来的潜在误写和恶意攻击问题，进一步增强对文件系统结构的保护和隔离。Treasury 框架支持动态链接程序不经修改直接运行在高性能的用户态非易失性内存文件系统之上，为实现各种不同结构的用户态非易失性内存文件系统提供了便利。在 Treasury 框架中实现的示例用户态非易失性内存文件系统 ZoFS，在基准测试和现实应用程序上的测试中均表现出更优的性能，在部分测试中对文件的访问性能达到了非易失性内存访问带宽的上限。

- 提出了**为非易失性内存优化的传统文件系统 Ext4pm**，解决成熟传统文件系统 Ext4 在非易失性内存上的性能瓶颈问题。通过对成熟的传统文件系统 Ext4 在非易失性内存上进行测试和分析，并与新型非易失性内存文件系统 NOVA 进行对比，本文识别出 Ext4 在非易失性内存上的性能问题，并将问题归纳为三个造成性能瓶颈的根本原因：过度抽象、非相称的实现、不可伸缩的设计。针对每个性能问题，本文找到了对应的第三方补丁或提出了对应的优化方案，解决 Ext4 在非易失性内存上的性能问题。通过不足 1,000 行的优化修改，为非易失性内存优化过的文件系统 Ext4pm 在微基准测试、综合基准测试和现实应用测试中均接近或超出新型非易失性内存文件系统的性能。

本文中所提出的三个文件系统适用于不同的使用场景。内核态异步非易失性内存文件系统 SoupFS 较为均衡。其既通过成熟的系统调用接口实现了完整的文件系统操作，又针对非易失性内存的特点设计了全新的结构，能较好地发挥非易失性内存的性能。用户态非易失性内存文件系统框架 Treasury 针对于对性能要求极高的场景，其设计和实现避免了系统调用、上下文切换和虚拟文件系统的开销，能够最大程度地发挥非易失性内存的性能优势。同时，Treasury 允许应用程序定制化文件系统的设计，能够有针对性地提升应用程序使用非易失性内存存储数据的效率。基于成熟传统文件系统、为非易失性内存优化的 Ext4pm 适用于对可靠性、稳定性要求较高的场景。其仅对 Ext4 的文件操作的处理流程进行了优化，修改的代码量少，且其不改变 Ext4 的存储布局，兼容为 Ext4 设计的各种管理、调优和监控工具。本文中使用的设计和优化思想，同样可以拓展到其他新型存储介质和其他场景中，为大数据时代高速文件系统的设计实现、优化和演化提供了新的方法和思路。

5.2 未来工作展望

5.2.1 持久的 CPU 缓存对文件系统设计的影响

在研究人员对非易失性内存文件系统进行研究的同时，非易失性内存技术也在不断发展。在英特尔最新的服务器平台中，新增了 eADR 特性[166]。在拥有该特性的服务器平台中，数据的持久化区域被进一步扩大——CPU 缓存进入到了持久化区域之中。换句话说，在此前，数据写入到 CPU 缓存后如果发生断电情况，数据依然可能会发生丢失。而在支持 eADR 的服务器平台中，数据一旦写入到 CPU 缓存中，即使发生了断电，服务器平台可以保证其有足够的备用电量将 CPU 缓存中全局可见的数据写回到持久的非易失性内存之中。这种 eADR 机制相当于将 CPU

缓存变为了持久存储。对于非易失性内存文件系统来说，其不再需要担心 CPU 缓存会将写入进行乱序，从而也不必再使用 CLFLUSH 等缓存刷除指令进行数据的强制写回。在这种情况下，如何对现有的非易失性内存文件系统进一步优化，是继续进行研究的方向之一。

5.2.2 新型用户态文件系统框架的完善与深入研究

本文设计的新型用户态非易失性内存文件系统框架 Treasury 为高性能用户态非易失性内存文件系统的设计和实现提供了便利。但其功能尚不够完善，设计和实现尚不够成熟。因此，在未来工作中，将首先根据现有文件系统接口的语义对 Treasury 的实现进行完善，覆盖并解决更多的边界问题。此外，根据数据库系统、键值存储系统等常用应用程序的特点，可以设计并实现专用的用户态非易失性内存文件系统（即 Treasury 中的 μFS），提升这些应用程序的性能。另一方面，由于 FUSE 文件系统框架的流行，大量的文件系统使用 FUSE 的接口实现在用户态。为了拓展 Treasury 的应用场景，可以实现 Treasury 对 FUSE 接口的兼容层，从而允许为 FUSE 框架设计的文件系统直接运行在 Treasury 框架中，并取得更好的性能。基于此，还可将现有的分布式文件系统客户端使用 Treasury 进行优化，或在 Treasury 中设计和实现与现有分布式文件系统兼容的全新高性能客户端。这将会是 Treasury 的未来发展方向之一。

5.2.3 新型文件系统和存储抽象

非易失性内存的访问接口与普通内存类似，因此，应用程序在使用存储功能时，不必再局限于使用传统的文件接口。在此背景下，系统软件（或文件系统）如何将非易失性内存以一种全新的、更高效、更符合非易失性内存特性的抽象暴露给新型应用程序进行使用，也是一个值得继续研究的话题。尤其在服务器平台支持 eADR 特性之后，CPU 缓存不再是影响一致性的障碍，新型应用程序可以更加简单地对非易失性内存进行使用，这对于一种对应用程序更加灵活，同时更利于系统软件进行管理的新型抽象提出了更加急切的需求，将会是存储系统未来发展的一个研究方向。

参考文献

[1] PConline. 攻克相变存储 PCM 芯片比当前闪存快 100 倍[Z]. https://servers.pconline.com.cn/news/1107/2457951.html. 2011.

[2] STMicroelectronics. Phase-Change Memory (PCM)[Z]. https://www.st.com/content/st_com/zh/about/innovation---technology/PCM.html.

[3] EverSpin. Spin-transfer Torque MRAM Technology[Z]. https://www.everspin.com/spin-transfer-torque-mram-technology.

[4] EETimes. Freescale begins selling 4-Mbit MRAM[Z]. https://www.eetimes.com/freescale-begins-selling-4-mbit-mram/. 2006.

[5] McGrath D. Intel, Samsung Describe Embedded MRAM Technologies[Z]. https://www.eetimes.com/intel-samsung-describe-embedded-mram-technologies/. 2018.

[6] Strukov D B, Snider G S, Stewart D R, et al. The missing memristor found[J]. Nature, 2008, 453(7191): 80-83.

[7] 新华网. 开启电子学新纪元的钥匙——忆阻器材料成 IT 基础研究新焦点[Z]. http://www.xinhuanet.com/tech/2018-07/04/c_1123074966.htm. 2018.

[8] Intel. Intel and Micron Produce Breakthrough Memory Technology[Z]. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/. 2015.

[9] Smith R. Intel Announces Optane Storage Brand For 3D XPoint Products[Z]. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products. 2015.

[10] Evangelho J. Intel And Micron Jointly Unveil Disruptive, Game-Changing 3D XPoint Memory, 1000x Faster Than NAND[Z]. https://hothardware.com/news/intel-and-micron-jointly-drop-disruptive-game-changing-3d-xpoint-cross-point-memory-1000x-faster-than-nand. 2015.

[11] Statt N. IBM's phase-change memory is faster than flash and more reliable than RAM[Z]. https://www.theverge.com/2016/5/17/11693054/ibm-phase-change-memory-breakthrough-ram-flash-storage. 2016.

[12] Sebastian A. Keep it Simple: Towards Single-Elemental Phase Change Memory[Z]. https://www.ibm.com/blogs/research/2018/07/phase-change-memory/. 2018.

[13] Clarke P. Samsung preps 8-Gbit phase-change memory[Z]. https://www.eetimes.com/samsung-preps-8-gbit-phase-change-memory/. 2011.

[14] MRAM-info. Samsung[Z]. https://www.mram-info.com/mram_memory_makers/samsung.

[15] MRAM-info. MRAM memory makers[Z]. https://www.mram-info.com/companies/mram_memory_makers.

[16] RRAM-info. RRAM companies: the comprehensive list[Z]. https://www.rram-info.com/companies.

[17] Tallis B. Intel Introduces Optane SSD DC P4800X With 3D XPoint Memory[Z]. https://www.anandtech.com/show/11208/intel-introduces-optane-ssd-dc-p4800x-with-3d-xpoint-memory. 2017.

[18] Tallis B. Intel Launches Optane Memory M.2 Cache SSDs For Consumer Market[Z]. https://www.anandtech.com/show/11227/intel-launches-optane-memory-m2-cache-ssds-for-client-market. 2017.

[19] Beeler B. Intel Optane SSD DC P4800X Review[Z]. https://www.storagereview.com/review/intel-optane-ssd-dc-p4800x-review. 2018.

[20] Intel. 英特尔 ® 傲腾 ™ 持久内存[Z]. https://www.intel.cn/content/www/cn/zh/products/memory-storage/optane-dc-persistent-memory.html.

[21] Beeler B. Intel Optane DC Persistent Memory Module (PMM)[Z]. https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm.

[22] Intel. 英特尔 ® 傲腾 ™ 数据中心级持久内存合作伙伴：华为[Z]. https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/huawei-partner-video.html.

[23] Intel. 英特尔 ® 傲腾 ™ 数据中心级持久内存合作伙伴：阿里巴巴[Z]. https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/alibaba-partner-video.html.

[24] Intel. 英特尔 ® 傲腾 ™ 数据中心级持久内存合作伙伴：腾讯[Z]. https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/tencent-partner-video.html.

[25] Intel. HPE and Intel® Optane™ DC Persistent Memory[Z]. https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/hpe-partner-video.html.

[26] Intel. Lenovo and Intel® Optane™ DC Persistent Memory[Z]. https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/lenovo-partner-video.html.

[27] Intel. Dell EMC and Intel® Optane™ DC Persistent Memory[Z]. https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/dell-partner-video.html.

[28] Intel. 大数据应用技术：至强 ® 平台和傲腾 ™ 持久内存定义内存数据库的未来[Z]. https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/accenture-partner-whitepaper.html.

[29] Boden N. Available first on Google Cloud: Intel Optane DC Persistent Memory[Z]. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory.

[30] Bablani G. Introducing new product innovations for SAP HANA, Expanded AI collaboration with SAP and more[Z]. https://azure.microsoft.com/en-us/blog/introducing-new-product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/.

[31] Strandberg K. Building an ecosystem for heterogeneous memory supercomputing[Z]. https://www.nextplatform.com/2020/07/27/building-an-ecosystem-for-heterogeneous-memory-supercomputing/.

[32] Rosenblum M, Ousterhout J K. The Design and Implementation of a Log-Structured File System[J/OL]. ACM Trans. Comput. Syst., 1992, 10(1): 26-52. https://doi.org/10.1145/146941.146943. DOI: 10.1145/146941.146943.

[33] Condit J, Nightingale E B, Frost C, et al. Better I/O Through Byte-addressable, Persistent Memory[C/OL]. in: SOSP '09: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. Big Sky, Montana, USA: ACM, 2009: 133-146. http://doi.acm.org/10.1145/1629575.1629589. DOI: 10.1145/1629575.1629589.

[34] Dulloor S R, Kumar S, Keshavamurthy A, et al. System Software for Persistent Memory[C/OL]. in: EuroSys '14: Proceedings of the Ninth European Conference on Computer Systems. Amsterdam, The Netherlands: ACM, 2014: 15:1-15:15. http://doi.acm.org/10.1145/2592798.2592814. DOI: 10.1145/2592798.2592814.

[35] Ou J, Shu J, Lu Y. A High Performance File System for Non-Volatile Main Memory[C]. in: European Conference on Computer Systems. 2016.

[36] Xu J, Swanson S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories[C/OL]. in: FAST'16: Proceedings of the 14th Usenix Conference on File and Storage Technologies. Santa Clara, CA: USENIX Association, 2016: 323-338. http://dl.acm.org/citation.cfm?id=2930583.2930608.

[37] Xu J, Zhang L, Memaripour A, et al. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System[C/OL]. in: SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China: ACM, 2017: 478-496. http://doi.acm.org/10.1145/3132747.3132761. DOI: 10.1145/3132747.3132761.

[38] Volos H, Nalli S, Panneerselvam S, et al. Aerie: Flexible File-system Interfaces to Storage-class Memory[C/OL]. in: EuroSys '14: Proceedings of the Ninth European Conference on Computer Systems. Amsterdam, The Netherlands: ACM, 2014: 14:1-14:14. http://doi.acm.org/10.1145/2592798.2592810. DOI: 10.1145/2592798.2592810.

[39] Kwon Y, Fingler H, Hunt T, et al. Strata: A Cross Media File System[C/OL]. in: SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China: ACM, 2017: 460-477. http://doi.acm.org/10.1145/3132747.3132770. DOI: 10.1145/3132747.3132770.

[40] Corbet J. Supporting filesystems in persistent memory[Z]. https://lwn.net/Articles/610174/. 2014.

[41] Wilcox M. Support ext4 on nv-dimms[Z]. http://lwn.net/Articles/588218/. 2014.

[42] Fuchsia[Z]. https://fuchsia.dev/fuchsia-src.

[43] David F M, Chan E M, Carlyle J C, et al. CuriOS: Improving Reliability through Operating System Structure[C]. in: OSDI'08: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, California: USENIX Association, 2008: 59-72.

[44] Ford B, Hibler M, Lepreau J, et al. Microkernels Meet Recursive Virtual Machines[C/OL]. in: OSDI '96: Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation. Seattle, Washington, USA: Association for Computing Machinery, 1996: 137-151. https://doi.org/10.1145/238721.238769. DOI: 10.1145/238721.238769.

[45] Klein G, Elphinstone K, Heiser G, et al. SeL4: Formal Verification of an OS Kernel[C/OL]. in: SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. Big Sky, Montana, USA: Association for Computing Machinery, 2009: 207-220. https://doi.org/10.1145/1629575.1629596. DOI: 10.1145/1629575.1629596.

[46] Levin R, Cohen E, Corwin W, et al. Policy/Mechanism Separation in Hydra[C/OL]. in: SOSP '75: Proceedings of the Fifth ACM Symposium on Operating Systems Principles. Austin, Texas, USA: Association for Computing Machinery, 1975: 132-140. https://doi.org/10.1145/800213.806531. DOI: 10.1145/800213.806531.

[47] Shapiro J S, Smith J M, Farber D J. EROS: A Fast Capability System[C/OL]. in: SOSP '99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles. Charleston, South Carolina, USA: Association for Computing Machinery, 1999: 170-185. https://doi.org/10.1145/319151.319163. DOI: 10.1145/319151.319163.

[48] Storage Performance Development Kit[Z]. https://spdk.io.

[49] The GNU C Library (glibc)[Z]. https://www.gnu.org/software/libc/.

[50] Oracle. Package java.io[Z]. https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html.

[51] Oracle. Package java.nio.file[Z]. https://docs.oracle.com/javase/7/docs/api/java/nio/file/package-summary.html.

[52] Ganger G R, Patt Y N. Metadata Update Performance in File Systems[C]. in: OSDI '94: Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation. Monterey, California: USENIX Association, 1994: 5-es.

[53] McKusick M K, Ganger G R, et al. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem[C]. in: USENIX ATC, FREENIX Track. 1999: 1-17.

[54] Ganger G R, McKusick M K, Soules C A, et al. Soft updates: a solution to the metadata update problem in file systems[J]. ACM Transactions on Computer Systems (TOCS), 2000, 18(2): 127-153.

[55] McKusick K. Journaling Soft Updates[C]. in: BSDCan. 2010.

[56] Ganger G R, McKusick M K, Soules C A N, et al. Soft Updates: A Solution to the Metadata Update Problem in File Systems[J/OL]. ACM Trans. Comput. Syst., 2000, 18(2): 127-153. https://doi.org/10.1145/350853.350863. DOI: 10.1145/350853.350863.

[57] McKusick M K, Roberson J. Journaled soft-updates[J]. BSDCan, Ottawa, Canada, 2010.

[58] Seltzer M I, Ganger G R, McKusick M K, et al. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems.[C]. in: USENIX ATC, General Track. 2000: 71-84.

[59] McKusick M K, Neville-Neil G, Watson R N. The Design and Implementation of the FreeBSD Operating System[M]. 2nd. Addison-Wesley Professional, 2014.

[60] NetBSD. Significant changes from NetBSD 1.4 to 1.5[Z]. http://www.netbsd.org/changes/changes-1.5.html. Accessed: 2020-09-01.

[61] OpenBSD. Mount — mount file systems[Z]. https://man.openbsd.org/mount.8. Accessed: 2020-09-01.

[62] Liu R, Chen H. SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability[C]. in: Proceedings of the Asia-Pacific Workshop on Systems. 2012: 15.

[63] Filebench - A Model Based File System Workload Generator[Z]. https://github.com/filebench/filebench.

[64] Tarasov V, Zadok E, Shepler S. Filebench: A flexible framework for file system benchmarking[J]. USENIX; login, 2016, 41(1): 6-12.

[65] Katcher J. PostMark: A New File System Benchmark[C]. in: 1997.

[66] Hitz D, Lau J, Malcolm M A. File System Design for an NFS File Server Appliance.[C]. in: USENIX winter: vol. 94. 1994.

[67] Kustarz E. ZFS-The Last Word in File Systems[J]. \group _ end :http://www.opensolaris.org/os/community/zfs/, 2008.

[68] Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-tree filesystem[J]. ACM Transactions on Storage (TOS), 2013, 9(3): 9.

[69] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 26-52.

[70] Seltzer M, Bostic K, McKusick M K, et al. An implementation of a log-structured file system for UNIX[C]. in: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings. 1993: 3-3.

[71] Konishi R, Amagai Y, Sato K, et al. The Linux implementation of a log-structured file system[J]. ACM SIGOPS Operating Systems Review, 2006, 40(3): 102-107.

[72] Min C, Kim K, Cho H, et al. SFS: random write considered harmful in solid state drives.[C]. in: FAST. 2012: 12.

[73] Rumble S M, Kejriwal A, Ousterhout J. Log-structured memory for dram-based storage[C]. in: Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14). 2014: 1-16.

[74] Lee C, Sim D, Hwang J, et al. F2FS: A new file system for flash storage[C]. in: 13th USENIX Conference on File and Storage Technologies (FAST 15). 2015: 273-286.

[75] 陈游旻, 朱博弘, 韩银俊, 等. 一种持久性内存文件系统数据页的混合管理机制[J]. 计算机研究与发展, 2020, 57(02): 281-290.

[76] Hagmann R. Reimplementing the Cedar file system using logging and group commit[C]. in: SOSP. ACM, 1987.

[77] Silicon Graphics International Corp. XFS: A High-performance Journaling File System[Z]. http://oss.sgi.com/projects/xfs. 2012.

[78] Chidambaram V, Pillai T S, Arpaci-Dusseau A C, et al. Optimistic crash consistency[C]. in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 228-243.

[79] 陈波, 陆游游, 蔡涛, 等. 一种分布式持久性内存文件系统的一致性机制[J]. 计算机研究与发展, 2020, 57(03): 660-667.

[80] Prabhakaran V, Rodeheffer T L, Zhou L. Transactional Flash[C]. in: OSDI. 2008: 147-160.

[81] Chidambaram V, Sharma T, Arpaci-Dusseau A C, et al. Consistency without ordering[C]. in: FAST. 2012: 9.

[82] Prabhakaran V, Bairavasundaram L N, Agrawal N, et al. IRON file systems[C]. in: SOSP '05: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. ACM, 2005: 206-220.

[83] Frost C, Mammarella M, Kohler E, et al. Generalized file system dependencies[C]. in: SOSP. 2007: 307-320.

[84] PM C, WT N, Chandra S, et al. The Rio File Cache: Surviving Operating System Crashes[C]. in: ASPLOS. 1996.

[85] Wu M, Zwaenepoel W. ENVy: A Non-Volatile, Main Memory Storage System[C]. in: ASPLOS. 1994.

[86] Wu X, Reddy A. SCMFS: a file system for storage class memory[C]. in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. 2011: 39.

[87] Sha E, Chen X, Zhuge Q, et al. Designing an Efficient Persistent In-Memory File System[C]. in: IEEE Non-Volatile Memory System and Applications Symposium. 2015: 1-6.

[88] Venkataraman S, Tolia N, Ranganathan P, et al. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory[C/OL]. in: FAST'11: Proceedings of the 9th USENIX Conference on File and Stroage Technologies. San Jose, California: USENIX Association, 2011: 5-5. http://dl.acm.org/citation.cfm?id=1960475.1960480.

[89] Yang J, Wei Q, Chen C, et al. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems[C/OL]. in: FAST'15: Proceedings of the 13th USENIX Conference on File and Storage Technologies. Santa Clara, CA: USENIX Association, 2015: 167-181. http://dl.acm.org/citation.cfm?id=2750482.2750495.

[90] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight Persistent Memory[C/OL]. in: ASPLOS XVI: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. Newport Beach, California, USA: ACM, 2011: 91-104. http://doi.acm.org/10.1145/1950365.1950379. DOI: 10.1145/1950365.1950379.

[91] Zuo P, Hua Y, Wu J. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory[C]. in: OSDI'18: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. Carlsbad, CA, USA: USENIX Association, 2018: 461-476.

[92] Nam M, Cha H, Choi Y r, et al. Write-optimized dynamic hashing for persistent memory[C]. in: 17th {USENIX} Conference on File and Storage Technologies ({FAST} 19). 2019: 31-44.

[93] Lepers B, Balmau O, Gupta K, et al. Kvell: the design and implementation of a fast persistent key-value store[C]. in: Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 447-461.

[94] Lee S K, Mohan J, Kashyap S, et al. Recipe: Converting concurrent dram indexes to persistent-memory indexes[C]. in: Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 462-477.

[95] Chen Y, Lu Y, Yang F, et al. FlatStore: An efficient log-structured key-value storage engine for persistent memory[C]. in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 1077-1091.

[96] Kimura H. FOEDUS: OLTP engine for a thousand cores and NVRAM[C]. in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. 2015: 691-706.

[97] Pelley S, Chen P M, Wenisch T F. Memory persistency[C]. in: ISCA. ACM, 2014: 265-276.

[98] Kolli A, Pelley S, Saidi A, et al. High-Performance Transactions for Persistent Memories[C]. in: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. 2016: 399-411.

[99] Kustarz E. MySQL[Z]. https://www.mysql.com. 2008.

[100] Kustarz E. PostgreSQL: The World's Most Advanced Open Source Relational Database[Z]. https://www.postgresql.org. 2008.

[101] Kustarz E. DokuWiki[Z]. https://www.dokuwiki.org/dokuwiki. 2008.

[102] Oracle. Other MySQL Documentation: Example Databases[Z]. https://dev.mysql.com/doc/index-other.html. 2019.

[103] PostgreSQL. Sample Databases[Z]. https://wiki.postgresql.org/wiki/Sample_Databases. 2018.

[104] Gilbey J. Debian Bug report logs - #853972[Z]. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=853972. 2017.

[105] SNIA. FSL-Dedup Traces[Z]. http://iotta.snia.org/traces/5228. 2019.

[106] Tarasov V, Mudrankit A, Buik W, et al. Generating Realistic Datasets for Deduplication Analysis[C/OL]. in: USENIX ATC'12: Proceedings of the 2012 USENIX Conference on Annual Technical Conference. Boston, MA: USENIX Association, 2012: 24-24. http://dl.acm.org/citation.cfm?id=2342821.2342845.

[107] SNIA. MobiGen Traces[Z]. http://iotta.snia.org/traces/5189. 2019.

[108] Jeong S, Lee K, Lee S, et al. I/O Stack Optimization for Smartphones[C/OL]. in: USENIX ATC'13: Proceedings of the 2013 USENIX Conference on Annual Technical Conference. San Jose, CA: USENIX Association, 2013: 309-320. http://dl.acm.org/citation.cfm?id=2535461.2535499.

[109] Corbet J. Memory protection keys[Z]. https://lwn.net/Articles/643797/. 2015.

[110] Park S, Lee S, Xu W, et al. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)[C/OL]. in: USENIX ATC '19: Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference. Renton, WA, USA: USENIX Association, 2019: 241-254. http://dl.acm.org/citation.cfm?id=3358807.3358829.

[111] Intel. Intel® 64 and IA-32 Architectures Software Developer Manuals[Z]. https://software.intel.com/en-us/articles/intel-sdm. 2019.

[112] Edge J. The ZUFS zero-copy filesystem[Z]. https://lwn.net/Articles/756625/. 2018.

[113] Min C, Kashyap S, Maass S, et al. Understanding Manycore Scalability of File Systems[C/OL]. in: USENIX ATC '16: Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference. Denver, CO, USA: USENIX Association, 2016: 71-85. http://dl.acm.org/citation.cfm?id=3026959.3026967.

[114] Google. LevelDB[Z]. https://github.com/google/leveldb. 2019.

[115] SQLite. SQLite Home Page[Z]. https://www.sqlite.org/index.html. 2019.

[116] The Transaction Processing Council. TPC-C Benchmark V5.11[Z]. http://www.tpc.org/tpcc/. 2019.

[117] Henson V, van de Ven A, Gud A, et al. Chunkfs: Using Divide-and-conquer to Improve File System Reliability and Repair[C/OL]. in: HOTDEP'06: Proceedings of the 2Nd Conference on Hot Topics in System Dependability - Volume 2. Seattle, WA: USENIX Association, 2006: 7-7. http://dl.acm.org/citation.cfm?id=1251014.1251021.

[118] Lu L, Zhang Y, Do T, et al. Physical Disentanglement in a Container-based File System[C/OL]. in: OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Broomfield, CO: USENIX Association, 2014: 81-96. http://dl.acm.org/citation.cfm?id=2685048.2685056.

[119] Kang J, Zhang B, Wo T, et al. SpanFS: A Scalable File System on Fast Storage Devices[C/OL]. in: USENIX ATC '15: Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference. Santa Clara, CA: USENIX Association, 2015: 249-261. http://dl.acm.org/citation.cfm?id=2813767.2813786.

[120] Zhan Y, Conway A, Jiao Y, et al. The Full Path to Full-path Indexing[C/OL]. in: FAST'18: Proceedings of the 16th USENIX Conference on File and Storage Technologies. Oakland, CA, USA: USENIX Association, 2018: 123-138. http://dl.acm.org/citation.cfm?id=3189759.3189771.

[121] Gough V. EncFS: an Encrypted Filesystem for FUSE[Z]. https://github.com/vgough/encfs. 2019.

[122] Libfuse. SSHFS[Z]. https://github.com/libfuse/sshfs. 2019.

[123] Hat G. Gluster: Storage for you cloud[Z]. https://www.gluster.org/. 2019.

[124] Szeredi M. Fuse: Filesystem in userspace[Z]. http://fuse.sourceforge.net. 2005.

[125] Peter S, Anderson T. Arrakis: A Case for the End of the Empire[C/OL]. in: HotOS'13: Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems. Santa Ana Pueblo, New Mexcio: USENIX Association, 2013: 26-26. http://dl.acm.org/citation.cfm?id=2490483.2490509.

[126] Peter S, Li J, Zhang I, et al. Arrakis: The Operating System is the Control Plane[C/OL]. in: OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Broomfield, CO: USENIX Association, 2014: 1-16. http://dl.acm.org/citation.cfm?id=2685048.2685050.

[127] Peter S, Li J, Woos D, et al. Towards High-performance Application-level Storage Management[C/OL]. in: HotStorage'14: Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems. Philadelphia, PA: USENIX Association, 2014: 7-7. http://dl.acm.org/citation.cfm?id=2696578.2696585.

[128] Engler D R, Kaashoek M F, O'Toole J, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management[C/OL]. in: SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. Copper Mountain, Colorado, USA: ACM, 1995: 251-266. http://doi.acm.org/10.1145/224056.224076. DOI: 10.1145/224056.224076.

[129] Boyd-Wickizer S, Chen H, Chen R, et al. Corey: An Operating System for Many Cores[C/OL]. in: OSDI'08: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, California: USENIX Association, 2008: 43-57. http://dl.acm.org/citation.cfm?id=1855741.1855745.

[130] Hedayati M, Gravani S, Johnson E, et al. Hodor: Intra-process Isolation for High-throughput Data Plane Libraries[C/OL]. in: USENIX ATC '19: Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference. Renton, WA, USA: USENIX Association, 2019: 489-503. http://dl.acm.org/citation.cfm?id=3358807.3358849.

[131] Vahldiek-Oberwagner A, Elnikety E, Duarte N O, et al. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)[C/OL]. in: 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, 2019: 1221-1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[132] Kadekodi R, Lee S K, Kashyap S, et al. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory[C/OL]. in: SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019: 494-508. https://doi.org/10.1145/3341301.3359631. DOI: 10.1145/3341301.3359631.

[133] Choi J, Hong J, Kwon Y, et al. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface[C/OL]. in: 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 2020: 1-16. https://www.usenix.org/conference/atc20/presentation/choi.

[134] Kim J, Soh Y J, Izraelevitz J, et al. SubZero: Zero-Copy IO for Persistent Main Memory File Systems[C/OL]. in: APSys '20: Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems. Tsukuba, Japan: Association for Computing Machinery, 2020: 1-8. https://doi.org/10.1145/3409963.3410489. DOI: 10.1145/3409963.3410489.

[135] Facebook. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage[Z]. https://github.com/facebook/rocksdb/. 2008.

[136] documentation T L K. The Linux Journalling API[Z]. https://www.kernel.org/doc/html/latest/filesystems/journalling.html. Accessed: 2020-08-01. 2008.

[137] Kustarz E. Linux ChangeLog-5.9.4[Z]. https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.9.4. 2020.

[138] Kara J. Ext4: Speedup ext4 orphan inode handling[Z]. https://patchwork.ozlabs.org/project/linux-ext4/patch/1429198977-5637-3-git-send-email-jack@suse.cz/. 2015.

[139] Kara J. Ext4 Filesystem Scaling[Z]. https://events.static.linuxfound.org/sites/events/files/slides/ext4-scaling.pdf. 2015.

[140] Corbet J. The rest of the 5.10 merge window[Z]. https://lwn.net/Articles/834504/. 2020.

[141] Shirwadkar H. Add fast commits in Ext4 file system[Z]. https://lwn.net/Articles/834483/. 2020.

[142] Pmem.io. Pmemkv[Z]. https://pmem.io/pmemkv/. 2008.

[143] Izraelevitz J, Yang J, Zhang L, et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module[Z]. 2019. arXiv: arXiv:1903.05714v3 [cs.DC].

[144] Orlov G. Directory Allocation Algorithm For FFS[Z]. https://web.archive.org/web/20080131082712/http://www.ptci.ru/gluk/dirpref/old/dirpref.html. 2008.

[145] Corbet J. The Orlov block allocator[Z]. https://lwn.net/Articles/14633/. 2002.

[146] Ts'o T. Orlov block allocator for ext3[Z]. https://lwn.net/Articles/14447/. 2002.

[147] Kustarz E. RocksDB[Z]. https://rocksdb.org. 2008.

[148] Patocka M. NVFS INTERNALS[Z]. https://people.redhat.com/~mpatocka/nvfs/INTERNALS. 2020.

[149] Patocka M. [RFC] nvfs: a filesystem for persistent memory[Z]. https://lore.kernel.org/lkml/alpine.LRH.2.02.2009140852030.22422@file01.intranet.prod.int.rdu2.redhat.com/. 2020.

[150] Ren Y, Min C, Kannan S. CrossFS: A Cross-layered Direct-Access File System[C/OL]. in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020: 137-154. https://www.usenix.org/conference/osdi20/presentation/ren.

[151] Traeger A, Zadok E, Joukov N, et al. A Nine Year Study of File System and Storage Benchmarking[J/OL]. ACM Trans. Storage, 2008, 4(2). https://doi.org/10.1145/1367829.1367831. DOI: 10.1145/1367829.1367831.

[152] Tarasov V, Bhanage S, Zadok E, et al. Benchmarking File System Benchmarking: It *IS* Rocket Science[C]. in: HotOS'13: Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems. Napa, California: USENIX Association, 2011: 9.

[153] Agrawal N, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Towards Realistic File-System Benchmarks with CodeMRI[J/OL]. SIGMETRICS Perform. Eval. Rev., 2008, 36(2): 52-57. https://doi.org/10.1145/1453175.1453184. DOI: 10.1145/1453175.1453184.

[154] Agrawal N, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Generating Realistic Impressions for File-System Benchmarking[J/OL]. ACM Trans. Storage, 2009, 5(4). https://doi.org/10.1145/1629080.1629086. DOI: 10.1145/1629080.1629086.

[155] Axboe J. Flexible I/O Tester[Z]. https://github.com/axboe/fio. 2008.

[156] Kustarz E. IOzone Filesystem Benchmark[Z]. http://www.iozone.org. 2008.

[157] Kustarz E. File and Storage System Benchmarking Portal[Z]. https://fsbench.filesystems.org. 2008.

[158] Kustarz E. DBENCH[Z]. https://dbench.samba.org. 2008.

[159] Wright C P, Joukov N, Kulkarni D, et al. Auto-Pilot: A Platform for System Software Benchmarking[C]. in: ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference. Anaheim, CA: USENIX Association, 2005: 53.

[160] Anderson E, Kallahalla M, Uysal M, et al. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking[C]. in: FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies. San Francisco, CA: USENIX Association, 2004: 45-58.

[161] Clements A T, Kaashoek M F, Zeldovich N, et al. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors[C/OL]. in: SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. Farminton, Pennsylvania: Association for Computing Machinery, 2013: 1-17. https://doi.org/10.1145/2517349.2522712. DOI: 10.1145/2517349.2522712.

[162] Bhat S S, Eqbal R, Clements A T, et al. Scaling a File System to Many Cores Using an Operation Log[C/OL]. in: SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China: Association for Computing Machinery, 2017: 69-86. https://doi.org/10.1145/3132747.3132779. DOI: 10.1145/3132747.3132779.

[163] Kim J H, Kim J, Kang H, et al. PNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers[C/OL]. in: APSys '19: Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems. Hangzhou, China: Association for Computing Machinery, 2019: 1-7. https://doi.org/10.1145/3343737.3343748. DOI: 10.1145/3343737.3343748.

[164] Papagiannis A, Xanthakis G, Saloustros G, et al. Optimizing Memory-mapped I/O for Fast Storage Devices[C/OL]. in: 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, 2020: 813-827. https://www.usenix.org/conference/atc20/presentation/papagiannis.

[165] Xu J, Kim J, Memaripour A, et al. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks[C/OL]. in: ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Providence, RI, USA: Association for Computing Machinery, 2019: 427-439. https://doi.org/10.1145/3297858.3304077. DOI: 10.1145/3297858.3304077.

[166] Intel. EADR: New Opportunities for Persistent Memory Applications[Z]. https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html. 2021.

攻读学位期间发表（或录用）的学术论文

[1] 第三作者. EI 国际会议论文, 2020.

[2] 第一作者. CCF A 类国际会议论文, 2019.

[3] 第二作者. CCF A 类国际会议论文, 2019.

[4] 第一作者. CCF A 类国际会议论文, 2017.

[5] 第一作者. EI 国际会议论文, 2016.

攻读学位期间获得的科研成果

[1] 第二发明人，中国发明专利，专利申请号 XXXXXXXXXXXX.X

[2] 第一发明人，中国发明专利，专利申请号 XXXXXXXXXXXX.X

[3] 第二发明人，中国发明专利，专利申请号 XXXXXXXXXXXX.X

[4] 第二发明人，中国发明专利，专利申请号 XXXXXXXXXXXX.X

[5] 第二发明人，中国发明专利，专利申请号 XXXXXXXXXXXX.X