分类号_____ 学号 M201572654

学校代码 10487 密级_____

华中科技大学

硕士学位论文

NVM 文件系统混合日志机制研究

学位申请人： 刘宁倩

学 科 专 业： 计算机系统结构

指 导 教 师： 陈俭喜 副教授

答 辩 日 期： 2018. 05. 22

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering

Research on Hybrid-Logging Mechanism in Non-volatile Main Memory File System

Candidate : Liu Ningqian

Major : Computer Architecture

Supervisor : Chen Jianxi

Huazhong University of Science & Technology

Wuhan 430074, P. R. China

May, 2018

独创性声明

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

学位论文作者签名：刘宇倩

日期：2018年5月25日

学位论文版权使用授权书

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

本论文属于 保密□，　　在_____年解密后适用本授权书。
不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：刘宇倩　　指导教师签名：陈建亭

日期：2018年5月25日　　日期：2018年5月25日

摘 要

随着非易失性内存技术（Non-Volatile Memory，NVM）快速发展，基于非易失性内存的文件系统成为当前的研究热点，其中如何保证文件系统的一致性是非常重要的研究内容。现有的 NVM 文件系统都有其各自的一致性保障机制，但也存在某些不足之处，对系统性能造成了一定的影响。除此之外，大部分 NVM 文件系统都只提供了文件系统级别的一致性保护，没有提供应用程序级别的数据一致性保护，从而导致应用程序必须设计自己的一致性保障机制，冗余的一致性保障机制对上层应用的性能有很大的影响。

针对上述问题，提出了基于 NVM 文件系统的混合日志机制（Hybrid Logging Mechanism，HLM），为文件系统元数据和数据均提供了一致性保证。首先，HLM 根据元数据和数据的不同更新特性以及 undo 日志和 redo 日志的主要开销，解耦了元数据和数据，对元数据和数据分别采取不同的日志机制，平衡了 undo 日志的数据拷贝开销和 redo 日志的日志索引开销；其次，设计了由 B 树和链表构成的二级日志索引结构，降低了读数据时的日志查询开销；此外，设计了最优并行 checkpoint 过程，充分利用系统空闲时间并行进行多个数据块的 checkpoint，对每个数据块的 checkpoint 过程采取优化策略最小化数据刷新量；最后，把事务功能模块抽象成了编程接口，以系统调用的形式提供给上层应用程序使用，提供了应用程序级别的数据一致性保护，消除了应用程序中冗余的一致性保障机制。

实验测试结果表明，混合日志机制在同时保证文件系统的元数据和数据一致性的基础上，能够提高文件系统的事务吞吐率，相对于 PMFS 和 NOVA 分别提高了 63.4% 和 76.3%。

关键词：非易失内存，文件系统，日志，一致性

Abstract

Currently file systems based on Non-Volatile Memory become the research hotspots because of the fast development of NVM technologies. Ensuring the consistency of application data is non-trivial in NVM-based persistent memory systems. Existing file systems have their own consistency mechanisms. However, these consistency mechanisms have their own cons, which degrades system performance. Moreover, most file systems only provide mechanism for protecting their own metadata or data from corruption while ignoring the corresponding protection for application data, providing no consistent update protocols for application data. As a result, most applications need to implement their own consistent protocols to achieve failure consistency, leading to disappointing performance on applications.

In this thesis, we propose a hybrid logging mechanism based on NVM file system. First, we decouple file system metadata and data and use different logging mechanism for metadata and data according to their different updating features and performance overhead of undo logging and redo logging. Second, we design and implement a Two-Level Index consisting of B-tree and list for redo logging, which can improve the performance of file reading. Third, we implement optimized concurrent checkpointing, which allows asynchronous checkpointing to be performed in parallel with minimum data copies. Finally, we offer a series of easy-to-use transaction-based interfaces for application to call, which can ensure the consistency of application data and eliminate redundant consistency mechanism in application level.

Evaluations on Non-Volatile file system which uses hybrid logging mechanism demonstrate that HLM can provide consistency for both metadata and data. Moreover, HLMFS outperforms PMFS and NOVA by up to 63.4% and 76.3%.

Key words：Non-volatile memory，File system，Log，Consistency

目 录

1 绪论

1.1 研究背景和意义

大数据的发展使得人们对数据存储的要求越来越高，例如要求存储系统具备高访问速度，高可靠性，低能耗等特性[1][2]。随着存储技术的发展，新型非易失内存（Non-Volatile Memory，NVM）器件因为其优良的性能越来越受到广泛关注，例如相变存储器（phase change memory，PCM）、铁电存储器（Ferroelectric RAM，FRAM）、忆阻器（Memristor）以及最近提出的 3D-XPoint 技术等。新型非易失存储器件兼具内存和存储器的特性，一方面，NVM 器件具有与 DRAM 相接近的读写访问延迟和字节可寻址的特性，另一方面，它又具有磁盘的非易失的特性[2]。NVM 存储器件与闪存、磁盘、DRAM 等的访问延迟对比图如图 1.1 所示。与磁盘和闪存相比，新型非易失存储器件的随机读写速度很快、能耗低，而且擦写次数远高于闪存，因此非易失存储器件的寿命也比闪存要长[7]。与 DRAM 相比，新型非易失存储器件具有更好的扩展性。由于新型非易失存储器件在访存速度、使用寿命等方面的性能有了很大的提高，因此被视为目前最具有发展潜力的存储器件[3][5][6]。而新型非易失存储器件的特性使得底层存储架构发生改变，从而对文件系统、持久性对象存储以及上层应用都产生了一定的影响，为了更好的利用新型非易失存储器件，基于非易失性内存的文件系统成为当前的研究热点[7]，目前已经出现了很多具有代表性的非易失内存文件系统。

表 1.1 NVM 特性总结

存储设备	PCM	Memristor	STT-RAM	DRAM
写延迟(ns)	150	100	40	15
读延迟(ns)	48	100	32	15
写次数(writes/cell)	10^8	10^{10}	∞	10^{18}
存储密度(um^2/bit)	0.0058	0.0058	0.0074	0.0038

在文件系统的研究中，一致性一直以来都是重点研究内容。文件系统的一致性要

保证当发生系统崩溃或者突然掉电时，系统重启之后数据能够被正确恢复，即数据能恢复到原来的一致性状态[11]。当对一个文件进行更新操作时，会涉及到更新文件的元数据即文件 inode 和文件的实际数据，对文件的元数据和数据的更新操作是不可分割的一个整体，这两部分更新操作全部完成才表示正确完成了一次文件更新[2]。如果在一次文件更新操作过程中发生系统崩溃，导致部分数据已经写入存储设备，而剩余部分还没有写入存储设备，例如，文件的元数据已经更新完成，而数据更新被中断，此时系统将处于不一致的状态，一致性保障机制就是要保证当发生系统崩溃中断了正在进行的文件更新操作时，系统重启后能通过回滚操作将数据恢复到这次更新之前的状态[2][4]。因此，为了保证文件系统的一致性，必须保证文件系统操作的原子性，而文件系统本身是无法保证操作的原子性的，所以必须借助事务来进行保证。事务在保证文件系统的一致性方面发挥着重要作用，事务的原子性要求一个事务中包含的一系列写操作要么全部完成，要么全部不做，把文件系统中一系列写操作包裹在事务中进行，只要保证了事务的原子性就可以保证文件系统操作的原子性，从而保证文件系统的一致性[10]。为了保证事务的原子性，写操作前后的旧数据版本和新数据版本都必须记录下来。文件系统通过一定的机制来保证事务的原子性，从而保证文件系统的一致性，常用的一致性方法包括日志机制[12][13][14]，写时复制（Copy On Write，COW）技术[15]等。然而，由于新型非易失存储器件具有字节可寻址、随机访问与顺序访问速度相接近等异于磁盘的特性，如果把传统的磁盘文件系统的预写日志技术（Write-Ahead Logging，WAL）[16][17]、影子分页技术（Shadow Paging）[22]等一致性保障机制直接用在非易失内存文件系统之上，会导致非常低下的系统性能。

另一方面，现有的 NVM 文件系统的一致性保障机制只能保证文件系统本身的数据的一致性，不能保证运行在文件系统之上的应用程序数据的一致性，因此，应用程序必须设计自己的一致性保障机制来保证它本身的数据的一致性，应用程序通常使用日志机制来保证数据的一致性。然而，由于应用程序不了解底层文件系统的特性，它的一致性保障机制通常是复杂的、容易出错的[11]。此外，大多数应用程序采取的日志机制都是基于传统磁盘的特点而设计的，直接运行在非易失内存文件系统之上，会造成大量冗余的数据拷贝，从而降低应用程序本身的性能[24]。

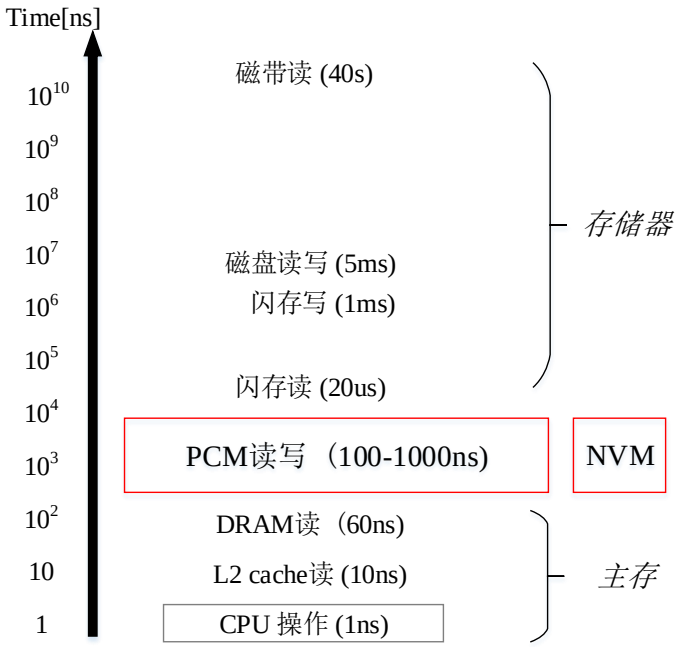


图 1.1 NVM 与其他存储设备访问延迟对比图

基于以上几点，根据 NVM 的特性设计高性能的一致性保障机制来同时保证 NVM 文件系统的元数据和数据的一致性，实现系统崩溃后数据的快速恢复，以及提供应用程序级别的数据一致性保护，从而提高应用程序的整体性能，具有重要的研究意义。

1.2 国内外研究现状

非易失内存技术的发展使得基于非易失内存的文件系统成为当前的研究热点，在基于非易失内存的文件系统的研究中，一致性问题仍然是重点研究内容。当前，具有代表性的 NVM 文件系统有 BPFS[21]、SCMFS[41]、Aerie[42]、PMFS[18]、NOVA[19]等，这些文件系统都有其各自的一致性保障机制。

BPFS 的数据组织形式是树形结构。树的每个节点都是大小为 4KB 的页面，内部节点是索引信息，叶子节点是实际的数据，文件的索引节点存放着文件大小和根指针等元数据信息。BPFS 利用了 NVM 字节可寻址的特性，提出了三种更新策略：就地更新、就地追加写和部分写时复制技术（Short-Circuit Shadow Paging，SCSP），其中部分写时复制技术基于传统的写时复制技术，结合 NVM 字节寻址、就地修改的特性，只对部分数据块进行拷贝和持久化，解决了传统写时复制技术中从修改处到文件系统根的冗

余拷贝的问题，从而减少了大量的数据拷贝开销。SCSP 保证了文件系统的一致性和可靠性，但是需要通过修改硬件来提供支持，可移植性不好。另外，当更新操作覆盖到文件系统树的大部分区域时，SCSP 仍然会导致写放大，影响系统性能。SCSP 虽然优化了传统的写时复制技术，但是也带来了一定的问题。

SCMFS 通过修改操作系统的内存管理模块来管理非易失内存设备，是建立在系统内核地址空间之上的。SCMFS 把文件映射到一块连续的虚拟地址空间中，通过这种管理方式可以使得文件系统的管理更方便简单，对每个文件来说，只需要管理一个文件大小和起始地址。SCMFS 的文件组织结构与传统的文件系统类似，SCMFS 通过采用空间预分配机制对文件进行空间的分配，可以避免频繁调用内存管理函数，从而降低分配空间带来的开销。在一致性方面，SCMFS 没有为元数据和数据提供任何一致性保证。

Aerie 允许上层应用直接访问 NVM 设备，并且允许应用程序自己定义访问 NVM 文件系统的接口，从而消除了传统访问方式中冗余的内核开销。Aerie 提供了一个用户类型的库 libFS 使得文件系统的元数据和数据能够被直接访问，Aerie 把文件系统分成了用户库 libFS 和文件系统服务（File System Service，TFS）两部分，libFS 提供文件系统访问接口。TFS 对元数据使用 redo 日志来进行更新，把新的元数据拷贝到日志中并且持久化，元数据写回后使用 wlflush 进行刷新。当发生系统崩溃，通过日志中记录的元数据来进行系统恢复。TFS 只对元数据进行了一致性保护，没有对数据进行一致性保护。

PMFS 是 NVM 直写的文件系统，消除了传统存储路径上的页高速缓存层，允许应用程序直接对底层 NVM 进行读写访问。在一致性方面，PMFS 只保证元数据的一致性，不保证数据的一致性，文件系统中元数据采取细粒度的 undo 日志。日志结构是一个称为 PMFS-Log 的固定大小的环形区域，头指针和尾指针分别指向有效日志区域的起始和结束为止，日志区域被划分成 64B 大小的日志项，每一个日志项用来记录一次对文件系统元数据的修改。在一个事务中，需要修改元数据时，会为这次事务分配一个日志项，把旧的元数据拷贝到刚分配的日志项中并持久化，然后对元数据进行就地更新，写入新的元数据并且把新写入的元数据持久化。如果在事务执行过程中，发生

系统崩溃或者突然掉电，即事务被中断，那么当系统重启之后，PMFS 就会扫描从头指针到尾指针之间的日志区域，日志区域包含了已经成功提交的事务以及被中断的事务，对于已经成功提交的事务，不做任何操作，而对于被中断的事务，则进行回滚，把日志中记录的旧数据拷贝回相应的区域。对于数据来说，并不记录日志，也没有其他的一致性保障机制来保证数据的一致性。因此，PMFS 只能保证元数据的一致性，不能保证数据的一致性。

NOVA 充分利用了 NVM 的快速随机访问的特性，对传统的 log-structure[20]方式进行了优化。NOVA 为每个 inode 维护一个独立的日志，允许多个文件的并发更新，inode 日志中只记录元数据，不记录数据，NOVA 使用 COW 方式更新数据，更新完数据之后，再把相关元数据的修改记录到日志中，日志中的元数据描述了本次事务所做的更新，并且会指向新数据页。NOVA 使用链表对日志进行管理，日志区域是 4KB 日志页面形成的链表。当一次操作涉及到多个 inode 时，则要使用 undo 日志对修改之前的每个 inode 进行记录。NOVA 中数据不使用日志进行记录而是使用 COW 方式更新，这种方式可以避免日志文件太大导致索引开销增大的问题，同时还能减小垃圾回收的开销，但是，COW 方法的粒度是 4KB，即一个数据块的大小，如果只需要更新 1KB 的数据，则需要首先分配一个 4KB 的新数据块，把原数据块中不需要修改的 3KB 拷贝到新块中，然后再写入 1KB 的新数据，由此可知，当数据更新量比较小的时候，使用 COW 方式会造成写放大，对系统性能有很大的影响，还会造成空间的浪费。SNFS[43]针对这个问题进行了优化，SNFS 预留了一块固定大小的数据日志区域，当数据更新量比较小时，采取日志方式进行更新，当数据更新量比较大时，才使用 COW 方式进行更新，SNFS 在保证文件系统的一致性的基础上，解决了小写采取 COW 方式造成的写放大问题，在一定程度上提高了小写的性能。

通过上面的分析可以看出，现有的 NVM 文件系统的一致性保障机制，具体实现方法或有不同，但基本思想都来源于日志机制和写时复制技术。在这些文件系统中，有的文件系统只保证了元数据的一致性，没有保证数据的一致性。有些文件系统虽然能同时保证元数据和数据的一致性，但是没有考虑元数据和数据各自的更新特性，对元数据和数据采取了相同的一致性保障机制，对系统性能产生了影响。此外，上述所有

文件系统都只提供了文件系统级别的一致性保护，没有提供应用程序级别的数据一致性保护，运行于这些文件系统之上的应用程序必须设计自己的一致性保障机制来保证数据的一致性，而应用程序中冗余的一致性保障机制对应用程序本身的性能也产生了很大的影响。

1.3 本文的研究内容

本文针对 NVM 文件系统的一致性问题，进行了一系列的研究。首先调研了具有代表性的 NVM 文件系统的一致性保障机制并且分析了它们各自的优缺点。针对当前 NVM 文件系统的一致性保障机制的不足之处，本文提出了一种混合日志机制来解决文件系统的一致性问题，主要工作包括以下几个方面：

(1) 通过分析元数据和数据的不同的更新特性以及 undo 日志和 redo 日志各自的性能开销，设计实现了混合日志机制，解耦了元数据和数据，元数据和数据分别使用不同的日志机制。平衡了数据拷贝开销和日志索引开销，提高了系统性能。

(2) 根据数据更新量大小的不同，设计实现了混合粒度的 redo 日志机制，当数据更新量比较大的时候采取页粒度的 redo 日志，当数据更新量比较小时采取字节粒度的 redo 日志，减少了持久化区域的大小，从而提高了系统写性能。

(3) 针对 redo 日志读性能低下的情况，设计实现了二级日志索引结构——第一级 B 树，第二级链表，加快了读数据时对日志的查询速度，降低了日志索引开销。

(4) 设计实现了最优并行 checkpoint 过程，充分利用系统空闲时间，并行进行多个数据块的 checkpoint 过程，每个数据块的 checkpoint 过程采取优化的 checkpoint 过程，使得拷贝和持久化的数据量最小化。

(5) 设计实现了基于链表管理的事务结构，系统中正在执行和已经提交的事务都用链表进行管理。系统将事务功能模块抽象为事务编程接口，以系统调用形式供上层应用程序调用，应用程序基于该事务接口进行编程就可以保证本身数据的一致性，系统提供了应用程序级别的一致性保证，从而消除了应用程序中的冗余的一致性保障机制，提高了应用程序的性能。

1.4 论文结构

本文共包含六章：

第一章：绪论，介绍本文的研究背景、当前国内外研究概况和本文的研究内容。

第二章：介绍了 NVM 存储架构，NVM 文件系统的一致性含义及方法，以及本文所采取的 undo 日志和 redo 日志的实现原理和主要流程。

第三章：介绍了混合日志机制的设计，重点介绍了系统框架、日志方式的设计、最优并行 checkpoint 过程的设计以及事务编程接口的设计，其中在日志方式的设计中，主要介绍了混合粒度的 redo 日志的设计和二级日志索引结构的设计。

第四章：介绍了混合日志机制的实现，具体介绍了初始化模块、写操作处理模块、数据写回模块、系统恢复模块以及事务交互模块等系统重要模块的实现细节。

第五章：测试部分，进行了功能测试和性能测试。功能测试主要验证本文提出的混合日志机制是否能保证文件系统的一致性。性能测试使用自己编写的事务吞吐率测试程序和标准测试工具 Filebench1.5 对系统的读写性能和事务处理性能进行了测试，还测试了运行在文件系统之上的实际应用 SQLite 和 Tokyo Cabinet 的性能。

第六章：总结全文工作并展望。

2 相关技术分析

本章主要介绍 NVM 存储架构以及 NVM 文件系统的一致性方法。另外，由于本文的研究是在日志机制的基础上进行的，因此着重介绍日志机制的实现原理和操作流程，并对两种日志方式 undo 日志和 redo 日志进行了分析和对比。

2.1 NVM 存储架构

传统块设备的访问延迟非常高，对系统性能影响很大，为了解决这个问题，在传统的基于块设备的文件系统的 IO 路径上引入了页高速缓存（page cache）和通用块层。然而，传统的基于块设备的存储架构并不适用于 NVM。一方面，NVM 的访问延迟与 DRAM 相接近，在页高速缓存与 NVM 之间冗余的数据拷贝会导致系统性能大大下降；另一方面，通用块层也会导致很大的软件开销。这两方面的开销会大幅度的削弱 NVM 的高性能的优势[31]。

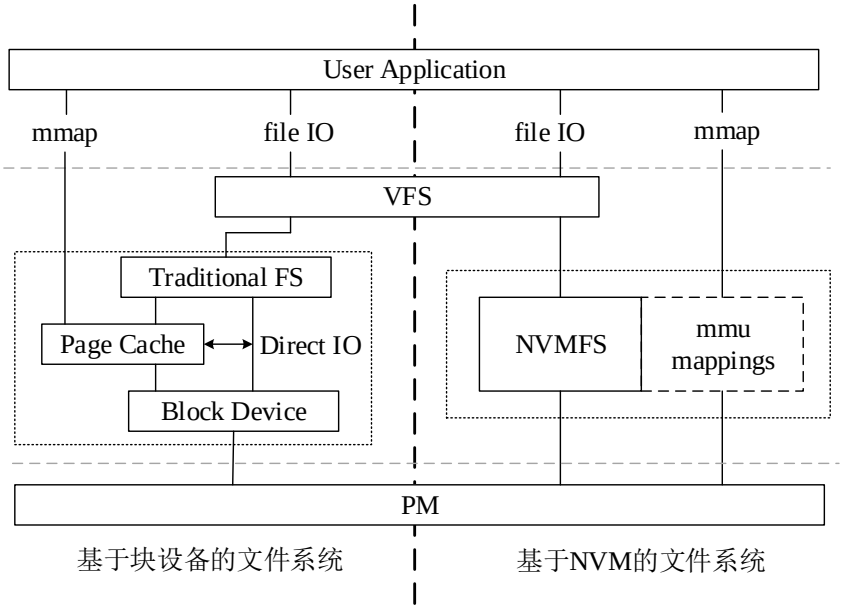


图 2.1 传统存储架构与 NVM 存储架构对比图

为了解决这个问题，现有的 NVM 文件系统都会绕过页高速缓存，允许应用程序对 NVM 设备直接进行读写访问。如图 2.1 所示，BPFS、SCMFS、Aerie、PMFS 和 NOVA

等都是采用的这种存储架构，应用程序的 IO 请求通过虚拟文件系统层、实际的文件系统层之后直接到达 NVM，绕过了传统的存储架构中的页高速缓存和通用块层，避免了多次数据拷贝的开销，能更好的发挥 NVM 的低访问延迟的优势，本文的研究也是基于绕过页高速缓存和通用块层的存储架构进行的。

2.2 NVM 文件系统一致性

新型非易失存储器件具有字节可寻址的特性，因此它可以像 DRAM 一样直接连接在内存总线上并且通过访存指令 load/store 进行访问，同时，它又具有非易失的特性，使得系统能够在内存级别保证数据的持久性，这些新特性使得传统的易失—持久边界从主存与磁盘之间转移到了 CPU Cache 与 NVM 设备之间，从而使得一致性边界也随之转移。如图 2.2 所示，在传统的存储架构中，主要关注的是如何保证主存和磁盘之间数据的一致性，而在新型存储架构中，关注重点变成了 CPU Cache 和 NVM 之间的数据一致性[24][25][26]，一致性边界的改变使得一致性要求也发生了变化。

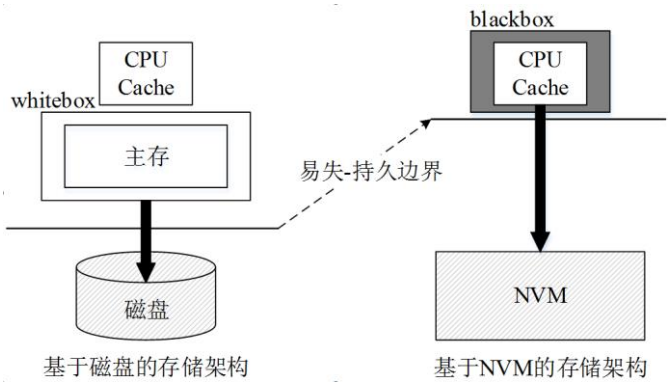


图 2.2 易失—持久边界的变化图

一致性要保证发生系统崩溃后数据能够被正确恢复，由 1.1 节可知，保证文件系统的一致性的前提是保证事务的原子性，此外，要想保证数据被正确恢复，事务中的写操作必须严格按照执行顺序写到存储设备上，因此，必须同时保证事务的原子性和写操作的顺序性才能保证文件系统的一致性[28]。

在传统的磁盘文件系统中，保证事务的原子性的方法有预写日志技术 WAL 和影子分页技术 Shadow Paging，这两种方法的实现原理分别是日志机制和写时复制技术。然

而，把这两种方法直接用于非易失内存文件系统来保证事务的原子性，系统性能会非常低。

一方面，WAL 直接把新数据写到日志区域，然后再把日志区域中的新数据拷贝到数据区域，后面这个过程称为 checkpoint，checkpoint 过程通常不立即进行，因此 WAL 只需要把新数据写到日志区域就完成了写操作。WAL 在以磁盘作为存储设备的系统中性能非常高，因为对日志区域的写操作是顺序性的，对数据区域的写是随机性的，磁盘的顺序写操作的开销比随机写操作的开销要低很多，WAL 用对日志区域的顺序写操作代替了对数据区域的随机写操作，从而避免了磁盘随机写操作带来的高开销。然而，NVM 的顺序写和随机写操作的延迟是相同的，因此 WAL 的优势在 NVM 存储系统中并不明显。

另一方面，Shadow Paging 基于写时复制技术，把新的数据写到数据区域的其他位置，当新数据刷新到存储设备上之后，再把指向旧数据的指针修改为指向新数据，这种方法会导致链式更新，即更新操作沿着文件系统的树形结构逐层向上扩散直到树的根指针，开销很大。针对这个问题，BPFS 提出了基于 NVM 文件系统的短路径影子分页技术 SCSP，SCSP 对小写采取就地更新，因此在文件系统树的任意位置都可以直接进行小数据量的修改，不会扩散到文件系统的根指针，但是 SCSP 方法只有在对单个文件进行原子性的小写操作时有比较好的性能提升，在提供应用程序级别的数据一致性保护时，SCSP 没有明显的性能优势，因为应用程序的一次事务操作可能会涉及到对多个文件进行更新，更新范围会覆盖文件系统树结构的大部分，导致提交操作有可能发生在文件系统树的相同的祖先处，在这种情况下，使用 SCSP 仍然会产生大量冗余的数据拷贝操作，对系统性能产生很大影响[37][38]。

对于写操作的顺序性，现有的处理器为了提高写操作的性能，会在 CPU Cache 处对写操作进行重排序，失序的写操作会导致系统的不一致，因此必须控制写操作的顺序性，由于一致性边界的变化，在基于磁盘的文件系统中，需要关注的是从主存到磁盘的写操作的顺序性，而在基于 NVM 的文件系统中，需要关注从 CPU Cache 到 NVM 设备的写操作的顺序性。在基于磁盘的存储系统中，主存中的数据页由操作系统管理，操作系统可以高效地控制写操作的顺序性。然而 CPU Cache 是由硬件控制的，在基于

NVM 的存储系统中，很难以较低的性能开销控制从 CPU Cache 到 NVM 设备的写操作的顺序性。为了使得从 CPU Cache 到 NVM 设备的写操作严格按照执行顺序进行，必须使用硬件指令 clflush 和 mfence 来进行控制，clflush 是将数据从 CPU Cache 刷回存储设备的指令，mfence 是顺序控制指令，把 clflush 与 mfence 指令结合使用能够控制写操作的顺序性。然而，clflush 和 mfence 指令的平均延迟是 250ns[3]，而 NVM 设备的访问延迟只有几十到几百纳秒，相比之下，使用数据刷新指令 clflush 和顺序控制指令 mfence 对写操作的性能会产生很大的影响，成为整个 NVM 文件系统的性能瓶颈。

通过上述分析可以看出，保证事务的原子性和写操作的顺序性是保证 NVM 文件系统一致性的关键，传统的磁盘文件系统使用 WAL 和 Shadow Paging 来保证事务的原子性，但是这两种方法不适合直接用于 NVM 文件系统。此外，在 NVM 存储系统中，还要特别关注写操作的顺序性，必须使用硬件指令来控制写操作的顺序性，而硬件指令的使用对系统性能也有很大的影响。基于以上两点，必须设计出适合 NVM 文件系统的一致性保障机制。由 1.2 节分析可知，现有的 NVM 文件系统都有其各自的一致性保障机制，这些一致性保障机制的原理都来自日志机制或者写时复制技术，但是这些一致性保障机制存在很多不足之处，例如性能低下、系统崩溃后数据恢复速度太慢、只能提供文件系统级别的一致性保护、有些一致性保障机制只能保证文件系统的元数据的一致性等。针对当前 NVM 文件系统的一致性保障机制存在的这些不足，研究实现高性能的一致性保障机制来保证文件系统的一致性是非常重要的，而在提供应用程序级别的数据一致性保护方面，日志机制比写时复制技术更有优势，因此，本文的研究基于日志机制进行。

2.3 日志机制

根据 2.2 节的描述，为了设计适用于 NVM 文件系统的高性能的一致性保障机制，同时保证元数据和数据的一致性，提供应用程序级别的数据一致性保护，并且实现系统崩溃后数据的快速恢复，本文的研究工作基于日志机制展开。因此，本节主要介绍日志机制的实现原理，介绍了两种日志方式的具体执行过程，并对它们的主要开销进

行了分析和对比。

日志分为两种：undo 日志和 redo 日志。

undo 日志的执行流程如图 2.3 所示，使用 undo 日志的事务执行过程分为四个阶段[39]。第一阶段，把旧数据从原来的位置拷贝到日志中，并且把日志中的旧数据持久化，在 NVM 设备中，持久化操作需要调用硬件指令 clflush 把数据从 CPU Cache 刷新到 NVM 设备上。第二阶段，把新数据从写缓冲区拷贝到原来的数据块中，也就是进行就地更新，在这一阶段，必须保证第一步日志中所有的旧数据全部都持久化完成之后，才能就地更新写入新数据，并且把刚刚写入的新数据持久化。第三阶段，提交事务，提交事务之前必须保证第二步中所有的新数据已经持久化完毕，提交事务的过程主要是修改事务的状态，把事务状态从执行中修改为已完成状态，标志整个事务的结束。第四阶段，在事务成功提交之后，删除记录旧数据的日志，从日志中移除刚才已经提交的事务[40]所包含的日志项。

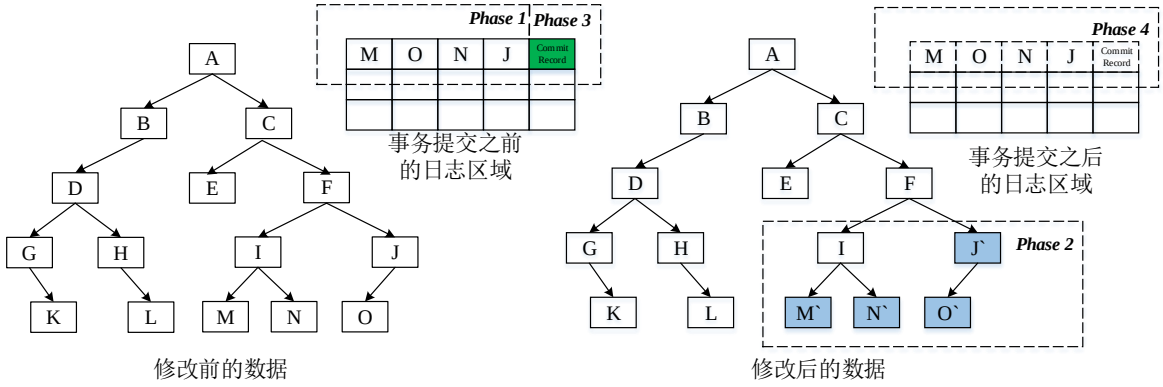


图 2.3 undo 日志的执行流程

redo 日志的执行流程如图 2.4 所示，使用 redo 日志的事务执行过程分为四个阶段[43]。第一阶段，把新数据直接从写缓冲区拷贝到日志区域，此时旧数据仍然在原来的数据块中。第二阶段，提交事务，在提交事务之前必须把所有写入到日志中的新数据持久化，只有日志中所有的新数据都持久化之后才能修改事务的状态，完成事务的提交。第三阶段，为了方便读操作的进行，把日志区域中记录的新数据拷贝到原来的数据块中并且持久化，也就是进行就地更新。第四阶段，删除日志，把刚才已经提交的事务所包含的日志项从日志中移除。第三阶段和第四阶段的操作称为 checkpoint 操作[41]，

checkpoint 操作一般不在事务提交之前进行。

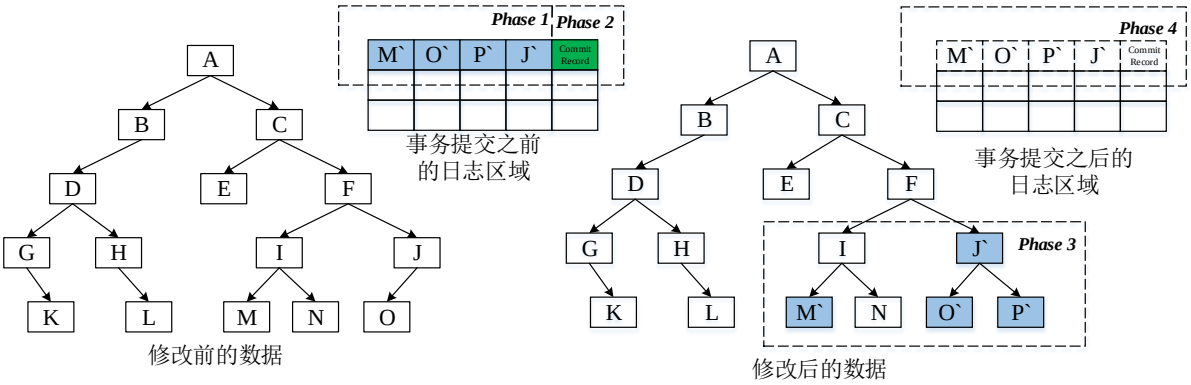
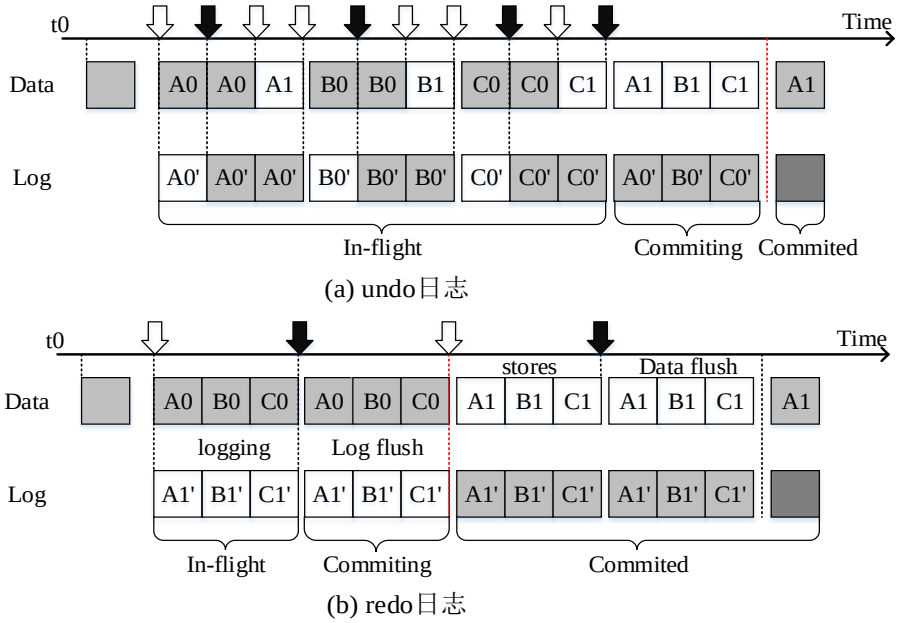


图 2.4 redo 日志执行流程

无论是使用 undo 日志还是 redo 日志，必须充分保证操作的顺序性才能保证日志的正确性[42]。在使用 undo 日志的一次事务流程中，必须保证两个顺序性操作，第一个是必须保证日志中所有的旧数据全部持久化之后，才能进行就地更新写入新数据，第二个是必须保证所有的新数据全部持久化完成之后才能提交事务[44]。而在使用 redo 日志的事务流程中，只需要保证一个顺序性操作，保证写到日志中的新数据全部持久化之后才能提交事务。只有保证了上述操作的顺序性才能保证当发生系统崩溃时，数据能够被正确的恢复，而保证 undo 日志和 redo 日志中操作的顺序性，就必须使用硬件指令 clflush 和 mfence。在 undo 日志中，把旧数据拷贝到日志区域后，在旧数据持久化操作和写入新数据操作之间调用 mfence 指令保证旧数据持久化完成之后再进行新数据的写入，调用 clflush 指令进行旧数据的持久化，在事务提交之前，调用 clflush 指令把写入的新数据持久化。在 redo 日志中，事务提交之前，调用 clflush 指令把日志中的新数据持久化。由于 undo 日志和 redo 日志的执行流程完全不同，因此它们的主要性能开销之处也不尽相同[45]。

图 2.5 表示在一次事务操作中更新三个数据块时 undo 日志和 redo 日志的执行时间轴。A0 表示旧数据块，A0’表示旧数据块在日志中的副本，A1 表示对 A0 进行更新后的新数据块，A1’表示新数据块在日志中的副本，数据块 B、C 同理。图 2.5(a)表示使用 undo 日志的事务执行过程，首先把旧数据（A0，B0 和 C0）写入到日志中（A0’，B0’和 C0’）并且把包含旧数据的日志持久化，然后进行就地更新写入新数据（A1，B1

和 C1）并且把新数据持久化，最后提交事务，如果事务执行过程中发生了系统崩溃，只需要把日志中记录的旧数据拷贝回原来的位置并且持久化，然后删除日志即可。图 2.5(b)是使用 redo 日志的一次事务执行过程，直接把新数据写到日志中（A1’，B1’和 C1’）并且持久化，然后提交事务即可，事务提交之后可以随时进行 checkpoint 操作把日志中保存的新数据拷贝回原来的位置并且持久化，由于旧数据一直在原来的位置没有被修改，如果在事务执行过程中发生系统崩溃，只需要删除被中断事务的日志直接把日志中记录的新数据丢弃即可[46]。



(a) undo日志

(b) redo日志

图 2.5 undo 日志和 redo 日志的时间轴对比

从图 2.5 可以看出 undo 日志和 redo 日志各自的优缺点[47]。Redo 日志的执行时间明显比 undo 日志的执行时间要短，这是因为在事务提交之前，undo 日志需要进行两次拷贝和持久化操作：把旧数据拷贝到日志区域并且持久化，把新数据从写缓冲区拷贝到数据区域并且持久化，即 undo 日志需要“写两次”。而 redo 日志只需要把新数据拷贝到日志区域并且持久化，只需要“写一次”。除此之外，redo 日志只需要保证一次顺序性操作，即事务提交之前日志中所有新数据持久化完毕，而 undo 日志需要进行两次顺序性保证，除了要保证事务提交之前日志中所有新数据持久化完毕，还需要保证每一次就地更新之前日志中的旧数据都持久化完成[38]。如果一个事务中包含多次写操作，redo 日志只需要一次顺序性保证，而 undo 日志需要多次顺序性保证。总体来说，

redo 日志比 undo 日志减少了一次数据持久化的开销。另一方面，对于读操作来说，redo 日志的开销要高于 undo 日志，undo 日志采取就地更新，新数据直接写入原来的位置，因此只需要从数据所在的位置直接读取就可以，而 redo 日志的新数据保存在日志区域，因此每次读数据时都需要查询整个日志区域找到最新版本的数据，对整个日志区域的查询过程开销很大[37]。由上面的分析可以看出，undo 日志比 redo 日志多了一次数据拷贝和持久化的操作，而 redo 日志每次读数据时的日志查询开销非常大，因此 undo 日志的主要开销是数据拷贝开销，redo 日志的主要开销是日志索引开销。

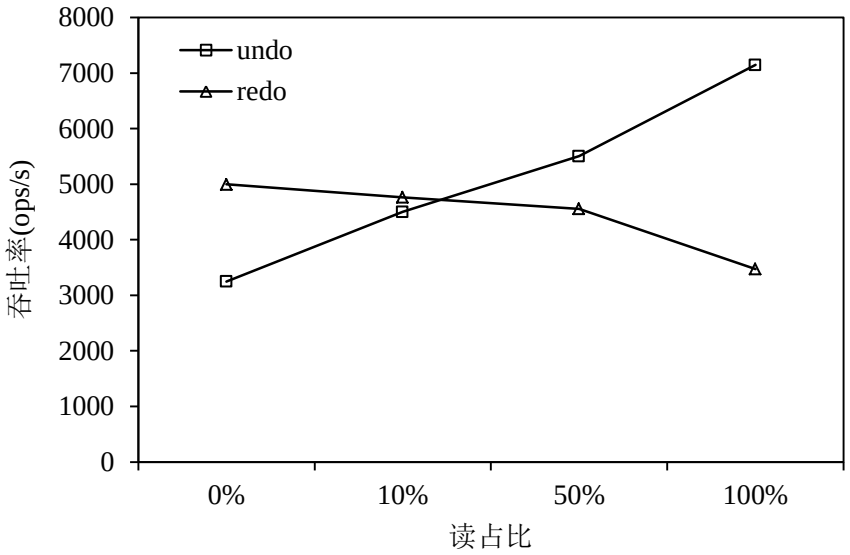


图 2.6 不同读写比例下 undo 和 redo 的吞吐率比较图

为了验证上述分析的正确性，对 undo 日志和 redo 日志进行了实际的测试。测试配置为在每个事务中创建一个文件，然后对文件进行 1000 次读写操作，测试不同读写比例情况下的吞吐率。分别对 undo 日志和 redo 日志进行上述测试，测试结果如图 2.6 所示。由测试结果可以看出，随着读占比的增大，redo 日志的性能越来越低，而 undo 日志的性能越来越高。当读写比例为 0%时，redo 日志比 undo 日志的性能高 53.7%。而当读写比例为 100%时，undo 日志的性能是 redo 日志的 2 倍。当读占比较小时，写操作对性能影响较大，undo 日志存在较大的数据拷贝开销，因此，比 redo 日志的性能低。当读占比较大时，读操作对性能影响较大，此时，redo 日志会带来较大的日志索引开销，因此，比 undo 日志性能低。对 undo 日志和 redo 日志的实际测试验证了上述分析

的正确性，undo 日志的主要开销是数据拷贝开销，redo 日志的主要开销是日志索引开销。

2.4 本章小结

本章主要介绍了 NVM 存储架构，与传统存储架构进行了对比。同时还介绍了保证文件系统一致性的关键和基本方法，分析了传统的磁盘文件系统的一致性保障机制不适用于 NVM 文件系统的原因，以及现有的 NVM 文件系统的一致性保障机制的不足之处。由于本文的研究内容是基于日志机制进行的，因此本章重点描述了日志机制的实现原理，着重分析了两种日志方式 undo 日志和 redo 日志的执行流程和它们各自的主要性能开销，为第三章混合日志机制的设计做铺垫。

3 混合日志机制的设计

本章主要介绍混合日志机制 HLM 的设计。根据日志机制的基本原理，结合 NVM 的特点以及文件系统元数据和数据的更新特性，本章提出了一种适用于 NVM 文件系统的高性能的混合日志机制。

本章 3.1 节介绍了系统框架，描述了整体的设计思路。3.2 节主要介绍了混合日志机制的日志方式以及提高文件系统读性能的二级日志索引结构。3.3 节主要介绍了降低数据拷贝和持久化开销，提高系统整体性能的策略——最优并行 checkpoint 过程。3.4 节主要介绍了事务管理方法和提供应用程序级别一致性保护的事务编程接口。

3.1 系统框架设计

由前两章的描述可知，NVM 文件系统和传统的磁盘文件系统的一致性方法都是日志机制或写时复制技术。但是由于 NVM 异于磁盘的新特性以及存储架构的变化，使得传统磁盘文件系统的一致性方法 WAL 和 shadow paging 不适合直接用于 NVM 文件系统，因此必须研究出适合于 NVM 文件系统的一致性方法，然而现有的 NVM 文件系统的一致性保障机制存在各种不足，它们或不能同时保证元数据和数据的一致性，或没有考虑元数据和数据的各自的更新特性，对元数据和数据采取了相同的一致性保障机制，导致系统性能非常低下，不能实现系统崩溃后数据的快速恢复。此外，现有的 NVM 文件系统都只提供了文件系统级别的一致性保护，没有提供应用程序级别的数据一致性保护，上层应用必须设计自己的一致性保障机制来保证数据的一致性，冗余的一致性保障机制也会对系统性能产生很大的影响。

针对上述问题，本文设计了混合日志机制来同时保证 NVM 文件系统的元数据和数据的一致性，实现系统崩溃后数据的快速恢复，在此基础上，通过减少持久化操作中数据刷新量的大小以及把刷新操作并行化等策略，来提高系统的整体性能。此外，通过设计事务编程接口来实现文件系统与应用程序的事务交互，提供应用程序级别的数据一致性保护，提高上层应用的读写性能。

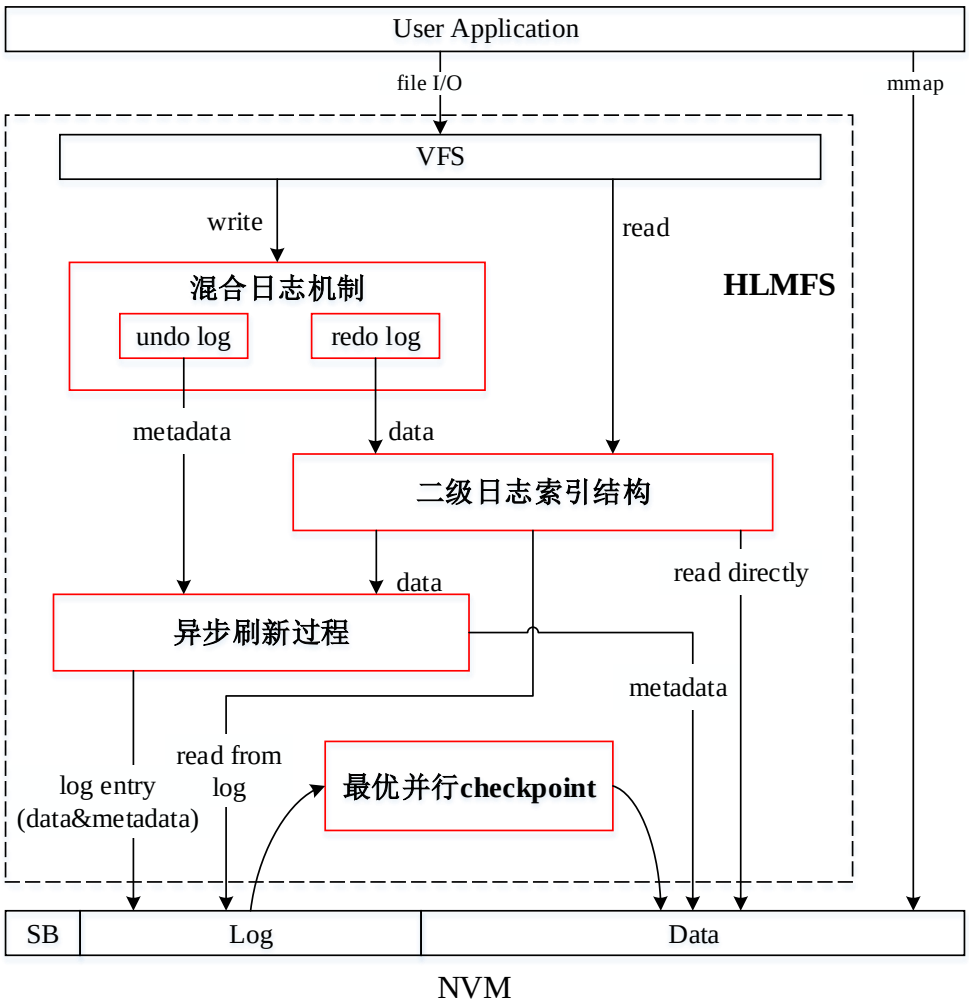


图 3.1 系统整体架构图

系统整体框架如图 3.1 所示。通过设计混合日志机制来同时保证元数据和数据的一致性，其中，根据元数据和数据的不同更新特性以及 undo 日志和 redo 日志的主要性能开销，解耦元数据和数据，分别对元数据和数据采取 undo 日志和 redo 日志。通过对 redo 日志构建二级日志索引结构来提高日志查询速度，从而提高文件系统读性能。通过设计最优并行 checkpoint 过程把刷新操作并行化，并且减少了持久化操作中数据刷新量的大小，同时充分利用系统空闲时间进行从日志区域到数据区域的数据写回处理，提高了系统的整体性能。混合日志机制在保证文件系统的元数据和数据的一致性的前提下，通过上述几种策略提高了文件系统的读写性能，并且通过快速恢复机制实现了系统崩溃后数据的快速恢复。此外，通过把事务创建、事务提交等功能模块抽象为事

务编程接口，以系统调用的方式供上层应用程序调用，提供了应用程序级别的数据一致性保护，同时提高了上层应用的性能。

3.2 混合日志方式的设计

通过 2.2 节的分析可以知道，undo 日志有两次数据持久化的过程，一次是持久化日志中记录的旧数据，另一次是持久化更新后的新数据，而 redo 日志只有一次数据持久化的过程，redo 日志直接把新数据记录到日志中，只需要在事务提交之前把日志中的新数据持久化即可，因此，undo 日志比 redo 日志多了一次“写操作”，即多了一次数据拷贝和持久化开销。另一方面，redo 日志的新数据都记录在日志中，因此每次读取数据的时候，都要查询整个日志区域寻找最新版本的数据，而 undo 日志的新数据就在原来的位置，每次读数据时直接在原来的数据块中读取即可，因此对于读操作来说，redo 日志比 undo 日志多了日志查询开销。

为了平衡数据拷贝开销和日志查询开销，设计了混合日志机制，解耦了文件的元数据和数据，根据元数据和数据的不同的更新特性，对元数据和数据分别采取不同的日志机制。

文件系统中元数据和数据的更新具有不同的特点。对于元数据来说，每次写操作的更新量通常都非常小，基本上只会涉及到一个文件系统数据结构大小的更新，例如一个 inode 结构体的大小，因此，元数据采取字节粒度的日志方式。但是，如果元数据采取字节粒度的 redo 日志，那么对于每个字节都要建立日志索引，会导致非常大的日志索引开销，除此之外，每次读取数据之前都要首先读取相关元数据信息，元数据需要频繁读取，每次读取元数据都需要查询日志，会进一步增大开销。因此，元数据采取字节粒度的 undo 日志，虽然 undo 日志比 redo 日志多了一次数据拷贝开销，但是元数据较小的更新量使得冗余的数据拷贝开销是可以接受的，而且采取 undo 日志之后，每次元数据的更新都是就地更新，新的元数据都是直接写到原来的位置，对元数据的频繁读取操作提供了很大的便利。

与元数据不同，对于数据来说，每次写操作的更新量都不是固定不变的，每次更

新操作的数据量大小小到几个字节，大到几个数据页的大小。当数据更新量非常大的时候，使用 undo 日志比使用 redo 日志增加的一次数据拷贝开销也会非常大，会大大降低系统性能。为了消除大量冗余的数据拷贝开销，数据使用 redo 日志。除了降低数据拷贝开销之外，redo 日志还通过优化的 checkpoint 过程避免了写放大，进一步降低了持久化操作的开销。

此外，日志粒度的选择也会对系统性能产生影响，不合适的日志粒度会造成写放大问题，因此，针对数据更新量不固定的情况，采取混合的字节粒度和页粒度的 redo 日志。当数据更新量比较小的时候，采取字节粒度的 redo 日志，新数据直接拷贝到日志项中。当数据更新量比较大时，字节粒度的 redo 日志不再适用，因为需要分配很多个日志项来保存数据，而每个日志项除了包含要记录的数据之外，还包含元数据字段，这些多余的元数据字段会造成写放大，增大持久化开销。而页粒度的 redo 日志，把数据保存在日志页中，只使用一个日志项来记录元数据信息和日志页的地址，减少了持久化操作的数据量，避免了写放大，可以提高系统性能，因此，当数据更新量比较大时，采取页粒度的 redo 日志。

整个系统的日志方式既包含 undo 日志，又包含 redo 日志。系统根据元数据和数据的不同更新特性以及 undo 日志和 redo 日志的主要开销，解耦了元数据和数据，分别设计了适用于元数据和数据的不同的日志方式和日志粒度，元数据采取字节粒度的 undo 日志，数据采取混合的字节粒度和页粒度的 redo 日志，同时保证了元数据和数据的一致性，也在最大程度上降低了数据持久化操作的开销。此外，解耦元数据和数据的设计使得在大数据量更新操作中，数据采取页粒度的 redo 日志，新数据保存在数据区域，从而不再占用大量日志空间。因此，数据更新量不再受限于固定大小的日志区域，从而使得文件系统能够支持大事务的执行，在使用混合日志机制的文件系统中，大数据量更新的事务的性能也非常好。

3.2.1 日志空间布局

文件系统的布局如图 3.2 所示，整个文件系统区域由超级块、日志项区域、混合的

日志页和数据页区域三部分组成。超级块中包含了文件系统元数据信息，各区域的起始地址和大小都在超级块中保存。

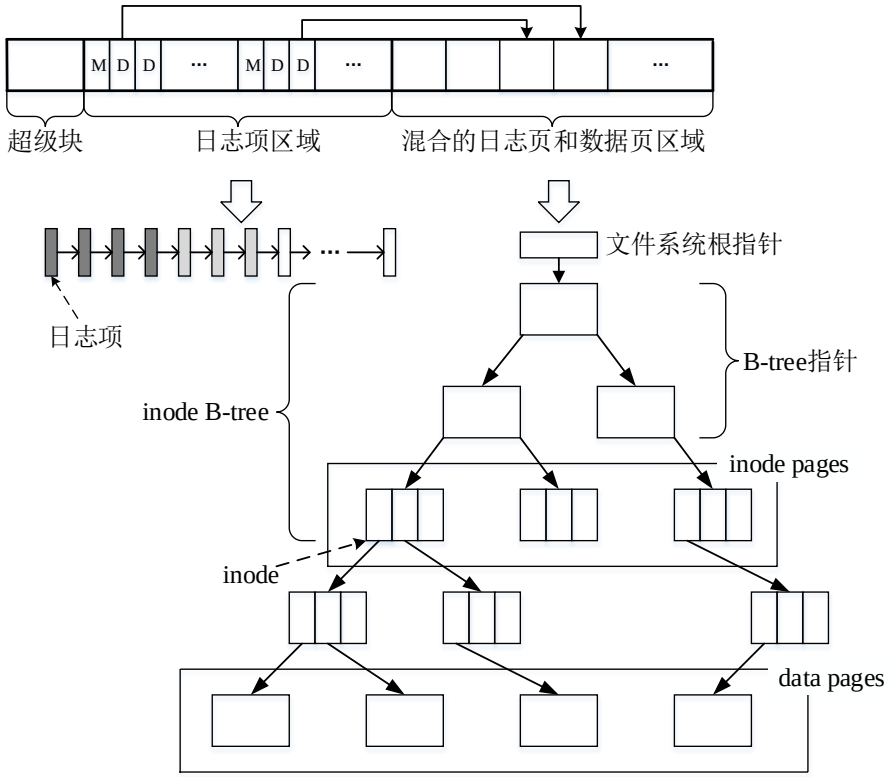


图 3.2 日志空间布局

日志项区域被划分为大小相等的若干个日志项，由于持久化操作是以 cacheline 为单位进行刷新的，因此，为了便于刷新操作的进行，日志项大小设置为与 cacheline 大小相同。为了便于日志项的管理，日志项以链表形式进行组织，日志项的分配沿着链表进行，空闲日志项的起始地址和空闲日志项的数量都保存在超级块中，每次分配日志项都从第一个空闲的日志项开始进行分配。元数据使用字节粒度的 undo 日志项记录更新前的旧数据，数据使用字节粒度的 redo 日志项和页粒度的 redo 日志项记录新数据，这三种类型的日志项都来自日志项区域。

字节粒度的日志项中包含日志项的元数据和数据信息，而页粒度的日志项中只包含元数据信息，数据信息需要保存在日志页中，日志页来自于混合的日志页和数据页区域。混合的日志页和数据页区域包含日志页（log page）和数据页（data page）两种

类型，页大小都是 4KB，其中数据页包含文件的实际数据，数据页以 B-tree 形式组织。日志页则由页粒度的 redo 日志项进行管理，在页粒度的 redo 日志项的元数据区域有一个指向日志页的指针，通过这个指针，可以找到具体的日志页，日志页中一般包含着新数据。

3.2.2 二级日志索引结构

数据采取 redo 日志方式，每次写入的新数据都会在日志区域保留一个数据版本，如果同一个数据块被修改过多次，那么在日志区域就会包含这块数据的多个数据版本，导致每次读数据都需要查询整个日志区域来寻找最新版本的数据，读数据时的查询开销非常大。

为了降低读数据时的日志查询开销，设计了二级日志索引结构（Two-Level Log Index，TLLI），用来快速索引最新版本的数据。redo 日志的二级索引结构如图 3.3 所示，TLLI 把同一个数据块的不同版本的数据所在的日志项用链表组织，称为版本链表。二级日志索引结构由两部分组成：第一级索引是 B 树，与数据索引的组织形式保持一致，使得数据索引的构建与日志索引的构建同时进行，简化了日志索引的构建过程，每个文件都会建立一个日志 B 树结构，用于索引版本链表的链表头，日志 B 树的根指针保存在文件的 inode 结构体中；第二级索引是版本链表，同一个数据块的不同版本的数据所在的日志项按照提交顺序插入到链表中，链表头部是最新版本的数据所在的日志项，链表尾部是较旧版本的数据所在的日志项。如图 3.3 所示，版本链表 L1 中的节点对应数据块 D1 的各种版本的数据，日志项 LE1_1 中记录的是最新版本的数据，日志项 LE1_2 中记录的是较旧版本的数据。数据块 D2 对应的版本链表为空，表明数据块 D2 没有被修改过或者已经完成了 checkpoint 操作。

二级日志索引结构的空间开销和性能开销都非常低。一方面，每一个 4KB 的数据块只需要 8B 的空间大小用于第一级 B 树索引，因此，二级日志索引结构占用的空间非常小。另一方面，由于二级日志索引结构占用空间很小，可以直接在 DRAM 中进行构建，所以索引结构的构建和查询开销都很小。与原来遍历整个日志区域查询最新版

本的数据方式相比，使用二级日志索引结构加快了日志查询过程，在很大程度上提高了日志查询效率，从而提高了整个系统的读性能。此外，二级日志索引结构还对提高 checkpoint 过程的性能有很大的帮助。

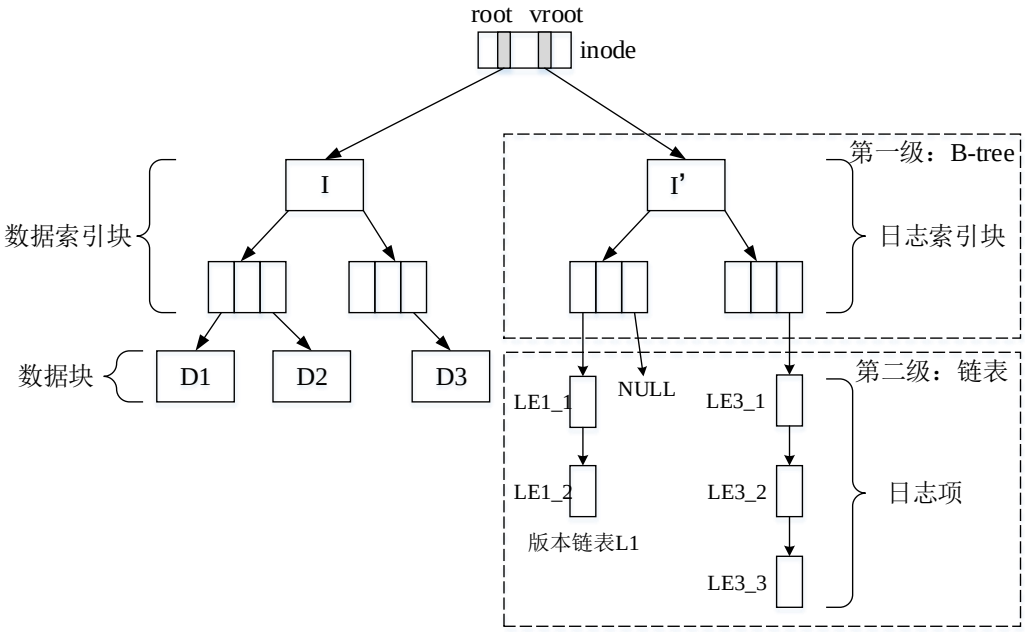


图 3.3 二级日志索引结构

3.3 最优并行 checkpoint 的设计

通过第 2 章的分析可知，将数据持久化到 NVM 上需要调用刷新指令 clflush，在整个写操作处理流程中，刷新操作的开销对系统写性能的影响是最大的。因此，必须降低刷新操作的开销才能提高文件系统的写性能。而降低数据刷新操作开销的方法有两种：一种是让每次刷新操作的数据量最小化，另一种是让刷新操作并行化，提高事务处理效率。

checkpoint 过程要求把日志区域中已提交事务的 redo 日志项中记录的所有新数据全部拷贝回原来的数据块中并持久化，在这个过程中存在大量的刷新操作，对系统性能有很大的影响。在进行 checkpoint 时，必须保证不能让较旧版本的数据覆盖较新版本的数据，否则就会导致数据的不一致，解决这个问题的常用方法是采取同步 checkpoint 过程或者严格按照事务提交顺序进行异步 checkpoint。同步 checkpoint 过程

是在事务提交之前进行 checkpoint，即把新数据写到日志项中并持久化之后，立即把日志项中的新数据拷贝回原来的位置并持久化，同步 checkpoint 过程在事务提交之前增加了一次拷贝和持久化操作，同样会带来很大的持久化开销。因此，一般采取异步 checkpoint 过程，然而传统的异步 checkpoint 过程必须严格按照事务提交顺序，从最旧的数据版本到最新的数据版本，依次把日志项中记录的数据拷贝回原来的数据块并且持久化，如果有多个日志项所记录的修改是针对同一块区域进行的，那么就会对这块区域进行多次拷贝和持久化操作，而实际上只需要把最新的数据版本拷贝回去并且持久化即可，因此严格按照提交顺序执行的异步 checkpoint 操作会引入多余的拷贝和持久化操作，造成写放大，也会带来较大的持久化开销。同步 checkpoint 过程和传统的异步 checkpoint 过程的性能开销都非常大。

基于上述分析，提高异步 checkpoint 过程的性能非常重要，因此设计了最优并行 checkpoint 过程，充分利用系统的空闲时间进行 checkpoint 操作，事务提交之前不立即进行 checkpoint 操作，可以大幅度的提高系统性能。通过设置一个阈值，当系统中空闲可用的日志项的数量低于这个阈值时，唤醒 checkpoint 线程，让 checkpoint 线程在后台并行地进行 checkpoint 操作。

每个数据块对应着由不同事务提交的多个数据版本，而针对 redo 日志设计的二级日志索引结构使得同一个数据块的多个数据版本被组织到同一个版本链表中，每个数据块都分别对应各自的版本链表，因此，对于不同数据块的最优 checkpoint 过程并行进行。

对每个数据块的 checkpoint 过程执行数据刷新量最小化的 checkpoint 过程，优化的 checkpoint 过程通过三个策略来最小化 checkpoint 过程中的数据刷新量：(1) 重新构建版本链表，释放包含冗余数据的日志项；(2) 合并多次刷新操作，把对版本链表中每个日志项的数据拷贝、刷新新数据、释放日志项的操作流程转变为把版本链表中所有日志项中的新数据全部拷贝完后，只进行一次刷新操作，最后释放整个版本链表；(3) 对页粒度的日志项，以修改指针的方式代替整个数据块的拷贝和持久化，把数据刷新量从一个数据块的大小降低为一个指针的大小。这三个策略解决了 checkpoint 过程中的写放大问题，使得数据刷新量最小化。

3.4 事务编程接口的设计

系统中的事务有两种类型，正在运行的事务和已经提交的事务。如图 3.5 所示，为了方便对事务进行管理，在超级块中管理两个链表，一个是正在运行的事务链表，另一个是已经提交的事务链表，在进行更新操作之前，首先要创建一个事务，把新创建的事务加入到正在运行的事务链表中，当更新操作结束之后提交事务，把成功提交的事务从正在运行的事务链表移除，插入到已经提交的事务链表。由于链表操作非常简单，因此这两个链表的设计对事务管理提供了极大的便利性。

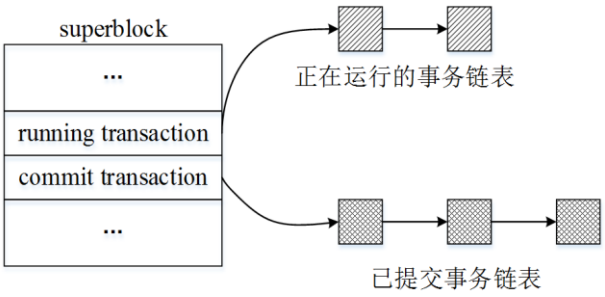


图 3.5 事务管理图

事务操作主要包括创建事务、提交事务和终止事务。为了提供应用程序级别的数据一致性保护，把事务功能模块抽象成了事务编程接口，以系统调用的方式供上层应用程序使用，相关的事务编程接口包括：hlmfs_new_txn()、hlmfs_commit_txn()和 hlmfs_abort_txn()，分别表示创建事务、提交事务和终止事务。在应用程序中，对文件进行更新操作前调用 hlmfs_new_txn()创建一个事务，更新操作完成后，调用 hlmfs_commit_txn()提交这个事务或者调用 hlmfs_abort_txn()删除这个事务，用这种方法代替应用程序中复杂的一致性保障机制。为了让文件系统能够识别每次写操作属于哪个事务，创建事务时会生成一个事务号，每个事务号都标志着唯一的一个事务，把事务中每次写操作与相关事务号进行绑定即可，这样在事务结束之前，每个与事务号绑定了的写操作都属于本次事务。为了结束一个事务，在 hlmfs_commit_txn()中将所有新数据持久化完成后，再进行修改事务状态、把事务从正在运行的事务链表转移到已提交事务链表等操作，或者调用 hlmfs_abort_txn()回滚事务中所进行的所有更新。

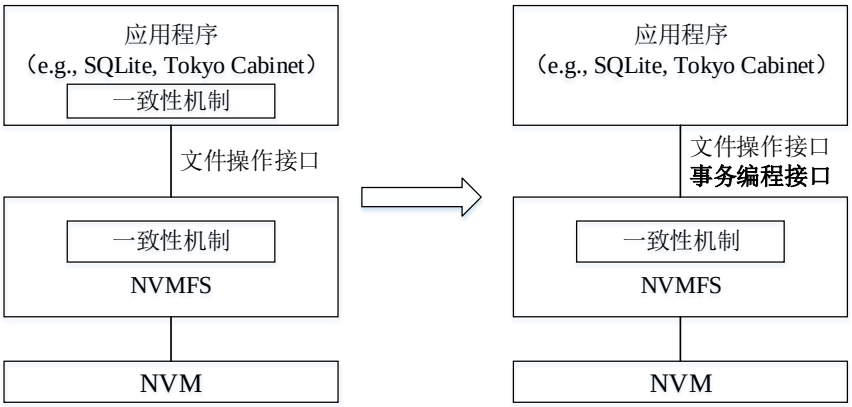


图 3.6 提供事务编程接口后的变化图

如图 3.6 所示，应用程序直接调用上述事务编程接口进行编程就可以保证数据的一致性，不需要再设计自己的一致性保障机制，因此可以消除上层应用中冗余的一致性保障机制，从而提高了上层应用的读写性能。

3.5 本章小结

本章主要介绍了保证 NVM 文件系统一致性的高性能混合日志机制 HLM。混合日志机制的设计思想主要有以下几点：(1) 解耦元数据和数据，根据元数据和数据的更新特性，分别为元数据和数据设计合适的日志机制来保证文件系统的一致性；(2) 通过对 redo 日志构建二级日志索引结构来提高文件系统读性能；(3) 通过设计最优并行 checkpoint 过程来降低持久化操作的开销，从而提高文件系统写性能；(4) 将事务功能模块抽象为事务编程接口，以系统调用形式提供给上层应用使用，消除了应用程序中冗余的一致性保障机制，提高了应用程序的性能。混合日志机制能够同时保证元数据和数据的一致性，保证系统崩溃后数据的快速恢复，同时采取一些策略提高了文件系统的读写性能，还提供了应用程序级别的数据一致性保障，消除了上层应用中冗余的一致性保障机制，提高了上层应用的事务处理性能。

4 混合日志机制的实现

本章主要介绍 NVM 文件系统的混合日志机制的具体实现，根据第 3 章对混合日志机制设计的描述，主要阐述以下几个模块的具体实现细节：

(1) 初始化模块。负责整个文件系统区域的初始化，尤其是各个区域的划分以及日志区域的初始化；

(2) 写操作处理模块。负责处理写操作过程中的元数据和数据的日志过程，以及二级日志索引结构的构建；

(3) 数据写回模块。负责在系统空闲时将日志区域中的有效数据写回到原来的数据区域；

(4) 系统恢复模块。负责在发生系统崩溃后将文件系统恢复到一致性状态，采取了快速恢复机制；

(5) 事务交互模块。负责文件系统与上层应用的交互，提供事务编程接口。

4.1 系统初始化模块

整个文件系统区域划分为三部分：超级块、日志项、混合的日志页和数据页。超级块主要用来管理日志区域和数据区域，日志区和数据区的元数据都存放在超级块中，在系统挂载时，进行超级块中信息的更新。

在超级块之后初始化一块固定大小的区域作为日志项区域，日志项区域的起始地址和大小都保存在超级块中。日志项区域被划分为大小相等的日志项，为了便于刷新操作的进行和提高刷新操作的效率，每个日志项大小设定为与刷新操作的基本单位 cacheline 大小相同，为 64B。日志项区域中的日志项用链表进行管理，初始化时，把日志项区域中相邻的日志项链接起来，形成链表。日志项的分配沿着链表进行。第一个空闲日志项的地址、最后一个空闲日志项的地址以及空闲日志项的个数都记录在超级块中，每次分配日志项时查询超级块中的这几个相关字段来进行分配，成功分配日志项之后更新超级块中的相关字段。

4.2 写操作处理模块

对一个文件进行写操作时，通常会涉及到对文件的元数据和数据的更新。因此，写操作处理模块分为两部分：一部分是对文件元数据的更新处理，另一部分是对文件实际数据的更新处理。

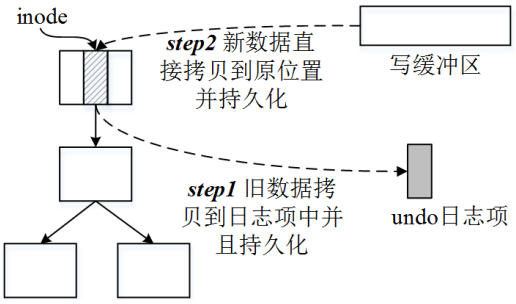
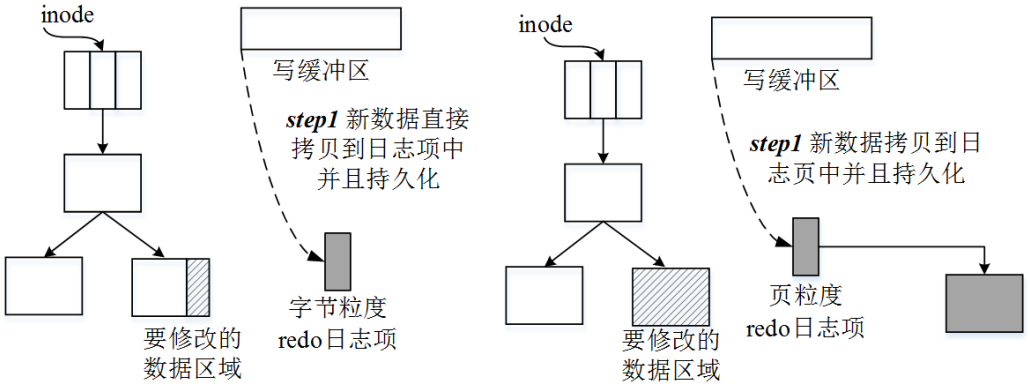


图 4.1 元数据更新过程

元数据的更新过程如图 4.1 所示，文件的元数据都在文件的 inode 结构体中，元数据采取字节粒度的 undo 日志，因此，首先从日志项区域分配 undo 日志项，undo 日志项结构前面的字段是日志项的元数据信息，用来描述日志项中所记录的数据的地址和大小等，后面的数据区域用来存放需要记录的数据。把旧的元数据复制到 undo 日志项的数据字段，然后把这个包含旧的元数据的日志项持久化，即调用 clflush 指令刷新到 NVM 设备上。然后就地更新，写入新的元数据，之后再把新的元数据持久化，这样就完成了元数据的更新。

数据的更新过程如图 4.2 所示，数据采取混合的字节粒度和页粒度 redo 日志。当数据更新量比较小的时候，采取字节粒度的 redo 日志，字节粒度的 redo 日志项的结构，与 undo 日志项类似，前面的字段是日志项的元数据信息，后面的 data 区域用来存放需要记录的数据，更新过程如图 4.2(a)所示，分配字节粒度的 redo 日志项，然后把缓冲区中的新数据直接复制到 redo 日志项的 data 字段，然后把日志项持久化。当数据更新量比较大的时候，采取页粒度的 redo 日志，页粒度的 redo 日志项结构中只包含元数据信息，不再有存放数据的字段，分配一个日志页，日志页的地址记录在页粒度的 redo 日志项中，需要记录的数据存放在日志页中，更新过程如图 4.2(b)所示，分配了页粒度的 redo 日志项和日志页之后，把写缓冲区中的新数据直接复制到日志页中，然后把日

志项和日志页都持久化，之后数据更新就完成了。



(a) 字节粒度的数据更新　　(b) 页粒度的数据更新

图 4.2 数据更新过程

在数据更新过程中，还会进行二级日志索引结构的构建。第一级日志索引 B 树的构建与数据索引的构建同步进行，当数据索引构建好之后，第一级日志索引也就构建好了，日志索引 B 树的根指针与数据索引 B 树的根指针相同，都存放在文件的 inode 结构体中。第二级日志索引链表的构建是在数据更新过程中进行的，对一个数据块的更新操作会产生包含新数据的 redo 日志项，根据数据块的逻辑块号可以找到数据块对应的版本链表的链表头，把 redo 日志项直接插入到版本链表的链表头，每次对数据块的更新操作产生的 redo 日志项都通过上述过程插入到版本链表中，由此形成第二级日志索引链表。二级日志索引结构构建好之后，在对文件进行读操作时，根据 inode 结构体中第一级日志索引 B 树的根指针的值找到第一级日志索引 B 树，再根据要读取的数据块的逻辑块号找到对应的版本链表，如果版本链表为空，则表明要读取的数据所在的数据块没有被修改过或者这个数据块已经完成了 checkpoint 操作，日志中不包含这个数据块的内容，直接到原数据块中读取即可。如果版本链表不为空，则表明这个数据块被修改过并且还没有进行 checkpoint 操作，日志中包含这个数据块的最新版本的数据，最新的数据都在对应的版本链表的日志项中，由于版本链表的链表头中记录的是比较新的数据，而链表尾记录的是比较旧的数据，因此从链表尾开始依次把日志项中记录的数据拷贝回原来的数据块中，然后直接从原来的数据块中读取所需要的数据就可以了，在这个过程中，只进行数据拷贝不进行刷新，在保证读取到正确数据的前

提下，引入了非常小的拷贝开销，保证了读操作的高效性。

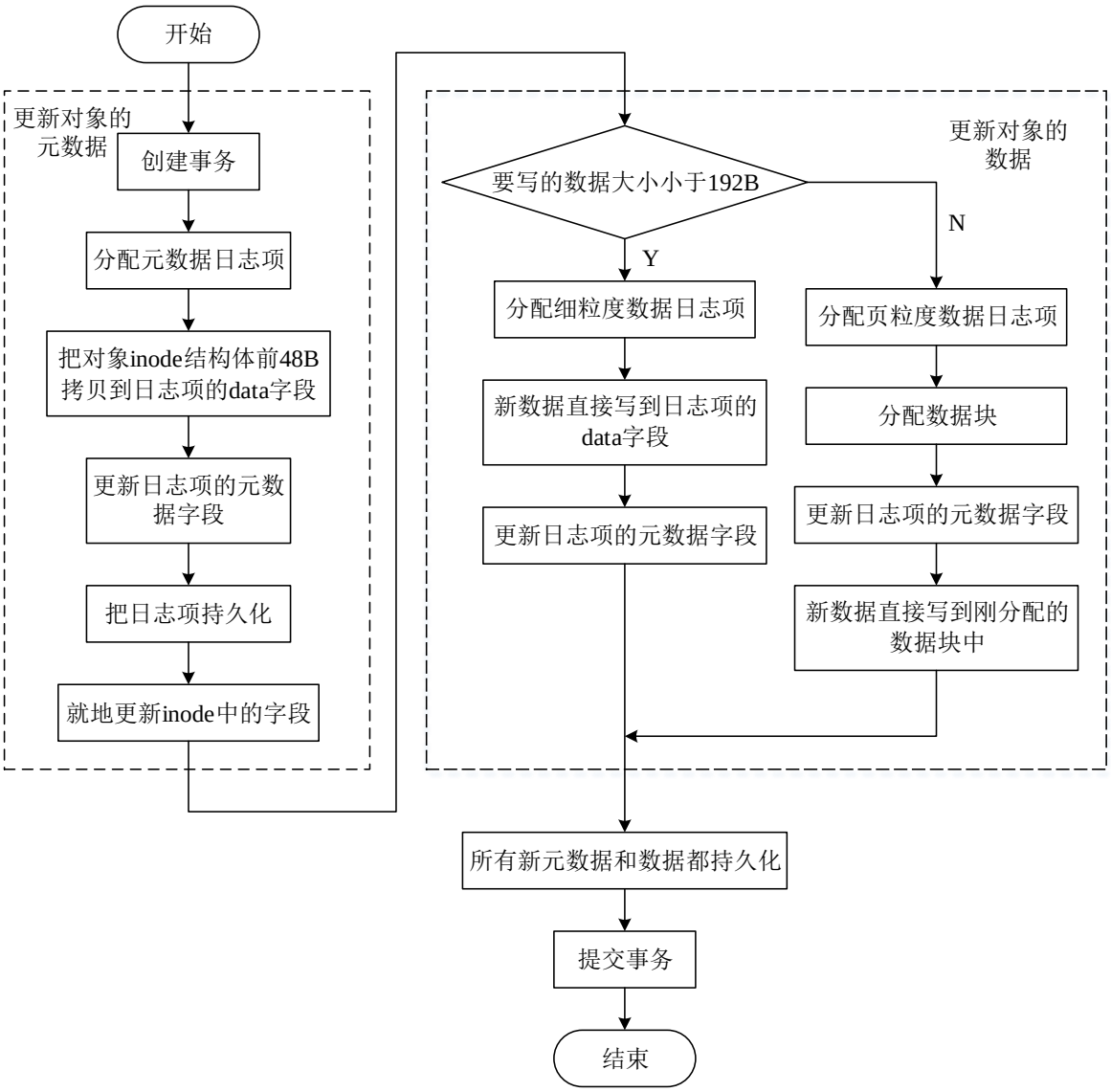


图 4.4 文件写操作流程

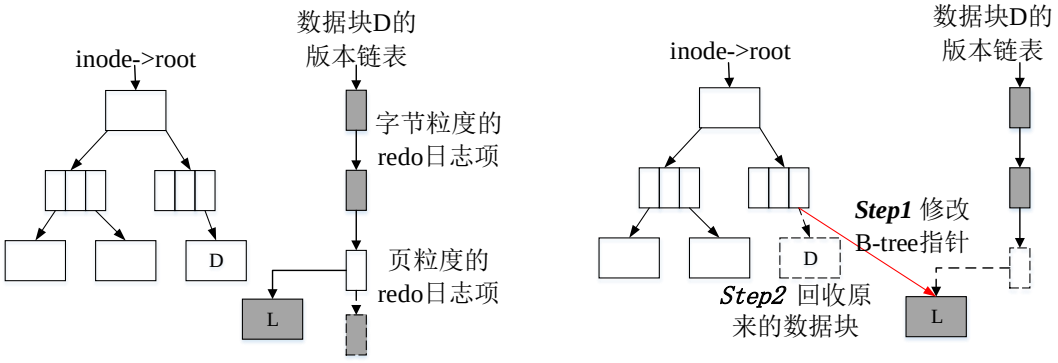
对一个文件进行写操作的完整流程如图 4.4 所示。在对文件进行更新之前先创建事务。在具体的更新过程中，首先对文件的元数据进行处理，具体的就是对元数据进行日志记录和更新操作，把拷贝后的新数据所在的区域加入待刷新链表。其次对文件的实际数据进行处理，新数据直接拷贝到日志项中，把包含新数据的日志项所在的区域加入待刷新链表，同时对当前文件构建二级日志索引结构，把刚生成的 redo 日志项插入到对应的版本链表中。最后，在待刷新链表中所有区域都刷新完成后，提交事务，

对文件的更新操作就成功完成了。

4.3 数据写回模块

为了避免日志空间不足造成系统错误，需要把 redo 日志项中记录的新数据写回相应的数据区域并且把写回的新数据持久化，也就是 checkpoint 过程。

系统挂载时，会创建一个清理线程，每当系统空闲时，唤醒清理线程，在后台进行日志区域的清理操作，在清理已提交事务的 redo 日志项时，进行数据写回操作。数据写回操作需要借助二级日志索引结构进行，由于不同的数据块分别对应各自的版本链表，不同数据块的版本链表之间不会互相干扰，因此对不同数据块的 checkpoint 分别由多个线程并行进行。



(a) 重新构建版本链表　　(b) 页粒度的redo日志项的checkpoint过程

图 4.5 优化的 checkpoint 过程

对同一个数据块的 checkpoint 过程采取数据刷新量最小化的 checkpoint 过程。优化的 checkpoint 过程在二级日志索引结构的基础上按照事务提交顺序进行，二级日志索引结构的第二级版本链表，链表头的 redo 日志项中包含的是较新版本的数据，链表尾的 redo 日志项中包含的是较旧版本的数据，checkpoint 过程如图 4.5 所示。

进行 checkpoint 过程前，首先要重新构建版本链表，从保存最新数据的链表头日志项开始遍历，移除被覆盖的包含较旧数据版本的日志项，例如在图 4.5(a)中，页粒度的 redo 日志项所指向的日志页中包含了一整块的新数据，与下面所有的 redo 日志项中的数据相比，这个日志页中的数据是比较新的版本，并且在更新范围上覆盖了下面 redo 日志项的更新范围，因此沿着版本链表这个日志项下面的所有 redo 日志项中的数据都

是较旧的版本，都是无效的，直接把下面的日志项标记为无效状态即可。重新构建版本链表消除了版本链表中部分冗余的 redo 日志项，在一定程度上减少了冗余的数据拷贝和持久化操作。

在进行具体的 checkpoint 操作时，由于版本链表中的节点就是按照事务提交顺序插入的，因此直接沿着链表把其中的 redo 日志项中记录的数据拷贝回原来的数据块中即可，把版本链表的所有 redo 日志项的拷贝操作完成之后，再对原来的数据块进行刷新操作，刷新完成后，释放版本链表中所有的 redo 日志项和日志页，与原来顺序性的对版本链表中的每个日志项分别进行拷贝、持久化、释放相比，采取这种方法只需要刷新一次，避免了冗余的持久化操作。

对于页粒度的 redo 日志项，没有采取把日志页中记录的数据拷贝回原来的数据块并且持久化的方法，而是把指向原来数据块的指针修改为指向这个日志页并且把修改后的指针内容持久化，如图 4.5(b)所示，这种方法进一步减少刷新的数据量的大小，把需要刷新的数据量从 4KB 的大小降低为 8B 的指针大小。

4.4 系统恢复模块

系统崩溃后数据的快速恢复对于文件系统的可靠性来说非常重要。当发生系统崩溃之后，日志区域的日志项处于以下三种状态中的一种：空闲状态、已经提交状态和未提交状态，其中已经提交状态的日志项属于成功提交的事务，而未提交状态的日志项属于发生系统崩溃时被中断的事务。为了实现数据的快速恢复，只对未提交的日志项进行回滚操作，对于已经提交日志项的写回操作在 checkpoint 过程进行，为此还要对已经提交的日志项进行索引重构。

系统恢复流程如图 4.6 所示。系统重启后的主要操作包括以下两步：

(1) 对已经提交但还未写回的日志项重新构建二级日志索引结构。遍历整个日志区域找到状态为已经提交（TXN_COMMITED）的日志项的日志项，事务的状态是用它的日志项链表中第一个日志项的状态标识的，而属于一个事务的所有日志项都用链表链接，由此可以构建出属于每一个已提交事务的日志项链表。每个数据块对应着一个

版本链表，同样可以构建出属于每个数据块的版本链表。遍历数据区域的数据块时可以构建出第一级日志索引 B 树，这样，整个文件系统中已经提交但是还未写回的日志项的二级日志索引结构就重新构建好了，日志项中的数据写回操作在 checkpoint 过程中进行。

(2) 对系统崩溃时被中断事务进行回滚。系统重启后首先遍历整个日志区域的日志项找到状态为 TXN_RUNNING 的日志项，由于事务的状态是用它的第一个日志项的状态标识的，因此，找到了状态为 TXN_RUNNING 的日志项，就找到了状态为 TXN_RUNNING 的事务的第一个日志项，状态为 TXN_RUNNING 的事务就是发生系统崩溃时被中断的事务，由于每个事务中使用到的所有日志项用另外一个链表进行了链接，因此遍历这个链表就可以遍历到属于被中断事务的所有日志项。对被中断事务中所有日志项进行回滚操作，在一个事务的日志项链表中，链表头是比较旧的数据，而链表尾是比较新的数据，因此回滚操作要从链表尾开始，依次向前进行。当遍历到一个日志项时，首先根据日志项的 status 字段判断该日志项是 undo 日志项还是 redo 日志项，如果是 undo 日志项，那么日志项中记录的数据是事务开始之前的旧数据，把日志项中的旧数据拷贝回原来的位置然后持久化即可。如果是 redo 日志项，那么日志项中记录的是新数据，则不需要进行任何拷贝操作，只需要把日志项释放就可以了。当把事务的所有日志项的回滚操作完成之后，把事务从事务链表中移除，然后释放事务占有的全部资源。重复上述回滚操作，直到把整个日志区域的所有日志项都遍历完，把所有被中断事务涉及到的数据都恢复完毕为止。

在经过上述恢复操作之后，发生系统崩溃时已提交事务的日志项的二级日志索引结构就被构建好了，在系统空闲时进行 checkpoint 操作即可把最新数据写回。系统崩溃时被中断的事务中所有 undo 日志中的旧数据按照正确的顺序被写回。因此，所有数据都恢复到了原来的一致性状态。这种恢复机制只对被中断事务进行回滚处理，对已经提交的事务只构建索引，能够实现系统崩溃后数据的快速恢复，数据恢复时间大大降低，提高了整个系统的性能。

上述恢复过程是对被中断的事务进行了恢复，除此之外，还要考虑当发生系统崩溃时正在进行 checkpoint 的情况，根据 3.3 节的描述，如果一个数据块对应多个版本的

数据，那么在进行 checkpoint 时，所有版本的数据按照正确的顺序都拷贝完成之后，再对数据块进行刷新，刷新完后才释放数据块对应的版本链表。如果正在进行拷贝操作时发生系统崩溃，由于还没有对数据块进行刷新，因此，拷贝之后的数据会丢失，但是数据块对应的版本链表还没有释放，所以，系统重启之后，就回到了 checkpoint 之前的状态，此时数据是处于一致的状态的。如果正在进行刷新操作时发生系统崩溃，此时已经拷贝过来的但是还未刷新的数据会丢失，但是此时的版本链表还未进行释放，因此可以重新进行 checkpoint 操作，checkpoint 操作完成后，数据就回到了一致的状态。因此，如果发生系统崩溃时，正在进行 checkpoint 过程，数据也都能通过一定的操作回到一致性的状态。

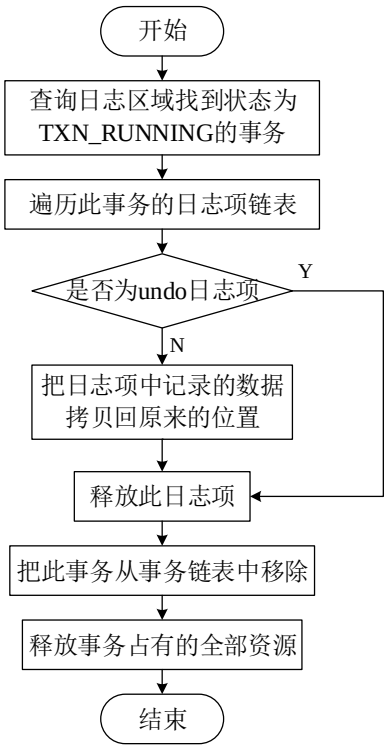


图 4.6 系统崩溃后的恢复流程

4.5 事务交互模块

文件系统与上层应用程序的交互通过事务编程接口和文件操作接口进行。事务编程接口与 read()、write()等文件操作接口类似，都是通过系统调用的方式提供给应用程

序使用。

应用程序通过调用 hlmfs_new_txn()函数通知文件系统创建一个新事务，文件系统执行创建新事务的操作，申请一个事务结构体，并生成对应的事务号，初始化事务结构体中的字段，把事务状态修改为正在运行（TXN_RUNNING），然后把这个事务插入正在运行的事务链表，当在文件更新操作中分配了日志项之后，更新事务结构体中的第一个日志项的地址、日志项的数量等字段。应用程序通过调用 hlmfs_commit_txn()来提交当前事务，文件系统检查待刷新链表中的所有区域是否都已经刷新完毕，如果还存在没有刷新的区域，则等待所有区域都刷新完成之后，把事务状态修改为已提交（TXN_COMMITED），然后把当前事务移动到已提交事务链表中。应用程序还可以通过调用 hlmfs_abort_txn()函数通知文件系统终止当前事务的运行，对当前正在运行的事务进行回滚，如果使用 undo 日志，则把日志中记录的旧数据拷贝回原来的位置并且持久化，如果使用 redo 日志，则直接删除日志项，文件系统通过回滚当前事务中所有已经完成的更新操作来终止当前事务。

通过上述三个事务编程接口可以实现文件系统与上层应用的事务交互，上层应用通过在写操作前后调用 hlmfs_new_txn()和 hlmfs_commit_txn()就可以保证应用程序数据的一致性，因此，可以消除应用程序自带的一致性保障机制，从而减少了上层应用中大量冗余的数据拷贝和持久化操作，在很大程度上降低了刷新开销，极大的提高了应用程序本身的读写性能。

4.6 本章小结

本章介绍了混合日志机制的实现细节。主要介绍了系统的几个重要的功能模块的具体实现，其中重点介绍了写操作处理模块中元数据和数据的日志处理过程、二级日志索引结构的构建过程的实现，还介绍了数据写回时最优并行 checkpoint 过程的实现以及系统崩溃后数据快速恢复的具体实现。除此之外，还介绍了事务交互模块中文件系统提供给上层应用的事务编程接口，主要介绍了事务创建和事务提交函数的具体操作过程。

5 测试与评估

前面两章分别介绍了混合日志机制的设计以及混合日志机制的实现，本章主要对采取混合日志机制的文件系统 HLMFS 进行测试。主要进行功能和性能两方面的测试。功能测试主要测试混合日志机制的正确性，即混合日志机制是否能保证系统的一致性。性能测试主要对 HLMFS 的读写性能、事务处理性能、系统崩溃后数据的恢复时间以及运行在 HLMFS 之上的实际应用程序 SQLite 和 Tokyo Cabinet 的性能进行了测试，并且与 PMFS、NOVA 等文件系统的性能进行了对比和分析。

5.1 测试说明

(1) 硬件

CPU：Intel(R) Xeon(R) CPU E5-2620 0 @2.00GHz

内存：Kingston KVR16E11 8GB DDR3-1333MHz

硬盘型号：Western Digital WD1003FBYX 1TB

(2) 软件

操作系统：CentOS-6.5　内核版本：Linux 4.3.6

测试软件：事务吞吐率测试程序，Filebench1.5，TPC-C，tctreetest

测试软件说明：事务吞吐率测试程序和 Filebench1.5[45]用来测试文件系统的读写性能和事务处理性能。上层应用选取数据库 SQLite[46]和 Tokyo Cabinet[47]，分别使用标准测试工具 TPC-C 和 Tokyo Cabinet 自带的测试脚本 tctreetest 来测试他们的性能。

表 5.1 测试环境信息

CPU	Intel(R) Xeon(R) CPU E5-2620 0 @2.00GHz
Memory	Kingston KVR16E11 8GB DDR3-1333MHz
HDD	Western Digital WD1003FBYX 1TB
OS	CentOS-6.5
Implementation Kernel Version	Linux 4.3.6

由于实际的 NVM 设备还没有市场化，而 NVM 的读写性能接近 DRAM，因此本系统使用 DRAM 来模拟 NVM 设备。在 8GB 的内存空间中预留 6GB 给 NVM 使用。

在进行写延迟模拟时，采用论文 Mnemosyne[40]中用 DRAM 模拟 PCM 的方式，将 DRAM 模拟成 PCM 设备，通过在持久化指令 clflush 后添加 PCM 与 DRAM 的写延迟差来进行模拟，本文添加 150ns 的延迟来模拟 PCM。由于 NVM 具有比 DRAM 更低的写带宽，因此，还需要进行带宽模拟，通过 DRAM 中的热控制功能来对带宽进行限制[48]，利用 Intel Xeon 处理器中的热控制寄存器 THRT_PWR_DIMM_[0:2]，以编程方式调节每个通道上的带宽，使用 setpci 命令对热控制寄存器进行控制，NVM 带宽默认设置为 DRAM 带宽的 1/8。

5.2 功能测试

功能测试主要对混合日志机制的正确性进行测试，即测试混合日志机制是否能保证系统的一致性。由于使用内存模拟 NVM 设备，但是内存具有易失性，发生系统崩溃数据就会丢失，因此不能引入真实的系统崩溃例如掉电来进行功能测试。所以，要模拟系统崩溃来进行测试，通过在数据库 SQLite 中插入用户级的程序错误来模拟系统崩溃，主要引入两种类型的用户级程序错误，一种失败点，另一种是随机的程序终止点。在程序错误被触发后，卸载文件系统，但是不清理日志区域，由此来模拟掉电过程。然后，重新挂载文件系统，由此来模拟系统重启，在挂载时会进行系统恢复操作，挂载成功后，利用 SQLite 自带的一致性检测工具来检测数据的一致性，SQLite 中的一致性检测工具是 PRAGMA integrity_check，这个命令可以对整个数据库的完整性进行检查。

首先，在 SQLite 中插入一系列的失败点 exit(2)，主要插入在事务提交之前，当程序执行到错误语句处时，就会立即终止执行，导致事务不能正常提交。其次，在执行用户程序 TPC-C 时随机的终止 TPC-C 的运行，TPC-C 是测试数据库性能的工具。当上述程序错误被触发后，用户程序会不正常的退出，系统崩溃后使用 SQLite 自带的一致性检查工具进行检查，检查结果如图 5.1 所示，由于事务没有正常提交，此时数据库的

数据是处于不一致状态的。重新挂载系统之后，系统会对被中断的事务进行恢复操作，当系统成功挂载之后，表示恢复操作已经完成，此时，再使用 PRAGMA integrity_check 对数据库的一致性进行检查，检查结果如图 5.2 所示，结果显示数据库此时处于一致的状态。在功能性测试中，共进行了 20 组失败点测试和 200 组随机的用户程序终止测试，在这些测试中，经过系统恢复后的数据库都能通过一致性检测工具的测试。因此，对系统进行功能测试的测试结果表明，混合日志机制是可以保证文件系统数据的一致性的。

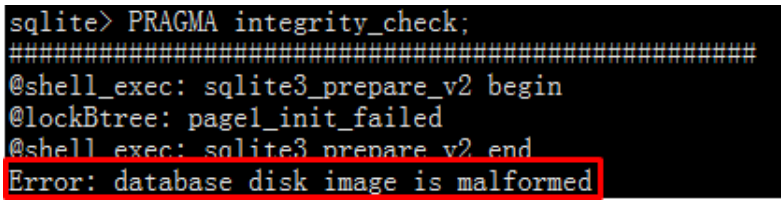


图 5.1 系统恢复之前 SQLite 一致性检查结果

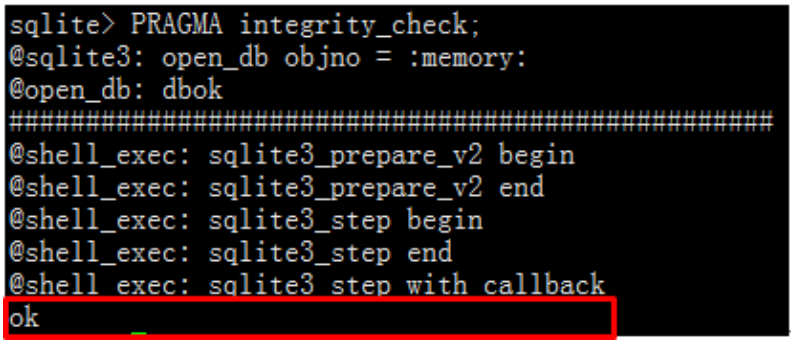


图 5.2 系统恢复之后 SQLite 一致性检查结果

5.3 性能测试

在性能测试中，主要进行了三种类型的测试。第一种测试是使用自己编写的事务吞吐率测试程序，主要对系统的事务处理性能，系统对写区域大小、NVM 写延迟以及系统空闲时间的敏感性，系统崩溃后的恢复时间进行了测试。第二种测试是使用标准测试工具 Filebench1.5，对负载为 fileserver 时的系统吞吐率进行了测试。除此之外，还对运行在文件系统之上的两种实际的应用程序 SQLite 和 Tokyo Cabinet 的事务吞吐率进行了测试。

表 5.2 性能测试所涉及到的文件系统

文件系统	描述
HLMFS	使用混合日志机制的文件系统
HLMFS-NAF	使用混合日志机制但是不用并行刷新方法的文件系统
HLMFS-NC	不使用任何一致性保障机制的文件系统
PMFS	元数据使用 undo 日志机制，数据用 COW 方式更新
NOVA	优化的 log-structure 方式保证一致性的系统

性能测试除了对使用混合日志机制的文件系统 HLMFS 进行测试之外，还对其他几种 NVM 文件系统进行了对比测试和分析，如表 5.2 所示，进行对比测试的文件系统包括 HLMFS-NAF、HLMFS-NC、PMFS 和 NOVA。HLMFS-NAF 是使用混合日志机制但是不使用并行刷新方法的文件系统，HLMFS-NC 是不使用任何一致性保障机制的文件系统，PMFS 只能保证元数据的一致性，对元数据采取 undo 日志机制，数据使用 COW 方式更新，NOVA 是使用优化的 log-structure 方式来保证系统一致性的。在性能测试中，使用这几种文件系统作为 HLMFS 的对照组，与 HLMFS 进行对比测试和分析。

5.3.1 事务吞吐率测试程序的测试结果和分析

事务吞吐率测试程序是每个线程执行 10000 次事务，在每个事务中，从 5000 个 16MB 的文件中随机选择四个文件执行 iosize 为 64B 到 16KB 的随机写操作。在本章的测试结果图中，txn/s 表示每秒执行的事务次数，即事务吞吐率。

使用事务吞吐率测试程序对系统进行了以下三部分的测试，第一部分对使用混合日志机制的系统的整体性能进行了测试，主要测试了单线程和多线程情况下的事务吞吐率以及系统的读性能。第二部分对系统的敏感性进行测试，主要测试了系统对写大小、系统空闲时间以及 NVM 延迟的敏感性。第三部分对系统奔溃后的恢复时间进行了测试。

(1) 系统整体性能测试

在单线程情况下运行上述测试程序，测试结果如图 5.3 所示。由测试结果可以看出

基于混合日志机制的文件系统 HLMFS 的事务吞吐率比 PMFS、NOVA、HLMFS-NAF 分别提高了 63.4%、76.3%、24.1%。与不使用任何一致性保障机制的文件系统 HLMFS-NC 相比，性能仅仅降低了 4.2%。HLMFS 的性能提升得益于三个方面：(1) 数据使用 redo 日志机制使得每次进行数据更新时的拷贝和持久化开销与 undo 日志相比减少很多，而根据数据更新量的大小设计的混合的字节粒度和页粒度的 redo 日志避免了写放大，也在一定程度上降低了持久化开销；(2) 最优并行 checkpoint 过程使得多个数据块的写回操作并行进行，并且每个数据块的 checkpoint 过程使用优化的 checkpoint 过程，使得数据拷贝和持久化操作的数据量达到最小，并且充分利用了系统空闲时间进行 checkpoint，把对主线程的事务处理的影响最小化；(3) 并行刷新过程使得一个事务中的数据处理操作与刷新操作并行进行，降低了事务执行的时间，提高了事务吞吐率。HLMFS 相对于 HLMFS-NAF 的性能有明显提升，说明并行刷新方法能够在一定程度上提升系统的事务处理性能，HLMFS-NAF 相对于 PMFS 和 NOVA 也有明显的性能提升，说明日志方式的设计和针对 redo 日志的最优并行 checkpoint 方法的设计都能提升文件系统的性能。总的来说，这三个方面共同作用使得 HLMFS 的事务吞吐率较另外几种文件系统相比有明显提升。

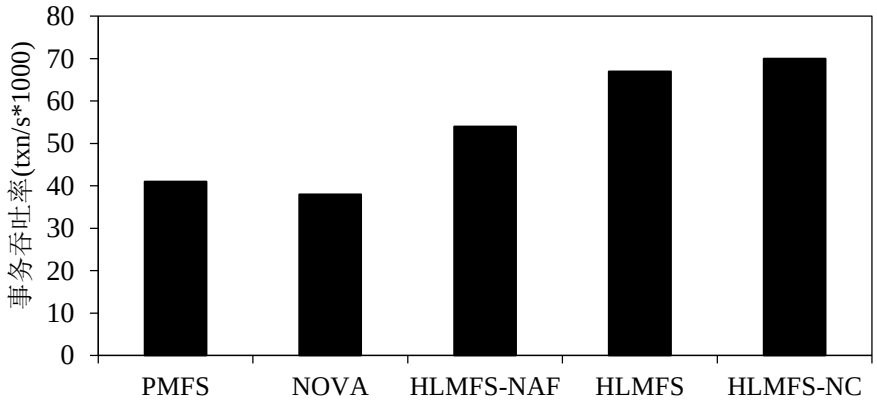


图 5.3 单线程事务吞吐率测试结果

进一步测试了多线程情况下的事务吞吐率，为了测试最优并行 checkpoint 过程对性能提高的影响，除了 PMFS、NOVA、HLMFS-NAF、HLMFS-NC 之外，还与 HLMFS-TAC 进行了测试和对比，HLMFS-TAC 是传统的异步 checkpoint 过程，严格按照提交顺序把每个版本的数据拷贝回原来的数据块并且持久化。

多线程测试结果如图 5.4 所示。由图可以看出，HLMFS 的事务吞吐率随着线程数的增加在不断的增长。当线程数为 1 的时候，HLMFS 相对于 PMFS、NOVA、HLMFS-NAF、HLMFS-TAC 分别提高了 63.4%、76.3%、24.1%和 8.1%。当线程数为 4 到 6 的时候，HLMFS-TAC 的事务吞吐率基本保持不变，这是因为当线程数增大的时候，需要进行 checkpoint 的数据量也会增大，而 HLMFS-TAC 采取传统的异步 checkpoint 过程，所有的数据都严格按照提交顺序依次拷贝回原来的数据块中，每次 checkpoint 都会造成写放大，因此限制了性能的增长。由图还可以看出，随着线程数的增加，HLMFS 和 HLMFS-NC 之间的性能差距越来越大，这是因为当线程数比较少的时候，需要进行 checkpoint 的数据量也比较小，后台的 checkpoint 线程不会对进行事务处理操作的主线程产生较大的影响，然而当线程数比较大的时候，需要进行 checkpoint 的数据量也会增大，此时系统性能就会被 checkpoint 过程所影响，即此时的系统性能主要取决于需要进行持久化的数据量的大小。但是，当线程数为 6 的时候，HLMFS 的性能仍然比 PMFS、NOVA、HLMFS-NAF 要高，分别高出 82.4%、76.1%和 43.5%。由图还可以看出，随着线程数的增加，HLMFS 相对于 HLMFS-NAF 的性能提升比也在不断增加，这与机器所能提供的物理资源相关，具体原因将在 5.3.2 节 Filebench 测试部分进行分析。总的来说，HLMFS 的性能相较于其他几个系统来说都有明显提升，与不使用任何一致性保障机制的系统 HLMFS-NC 相比，差距也不是太大。

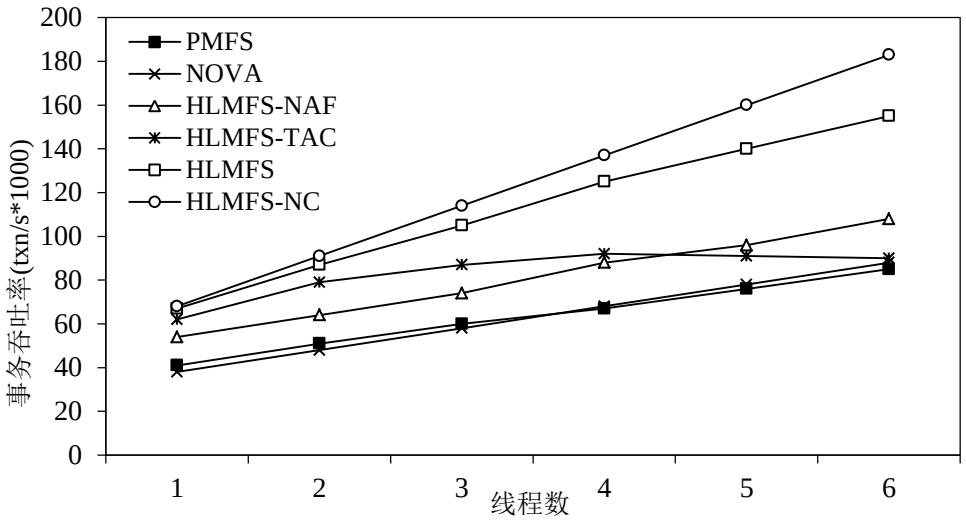


图 5.4 多线程事务吞吐率测试结果

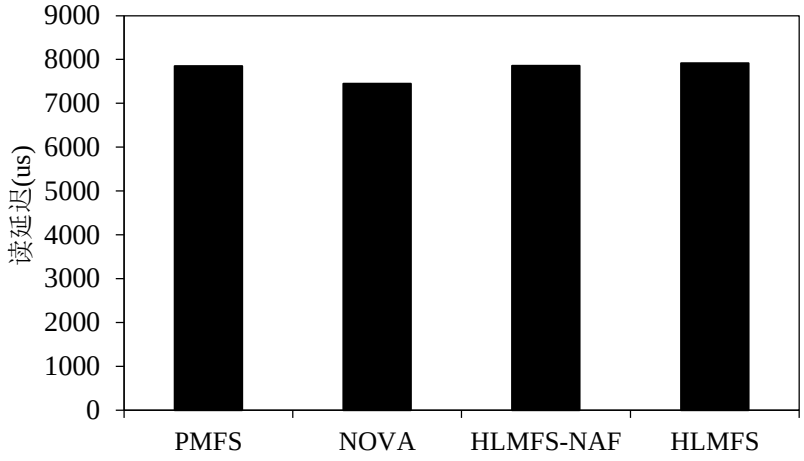


图 5.5 系统读性能测试结果

系统读性能的测试结果如图 5.5 所示，由测试结果可以看出，HLMFS 的读性能与 PMFS、NOVA、HLMFS-NAF 的读性能相当。这是因为在当前的测试环境中，HLMFS 能够充分利用系统空闲时间进行 checkpoint，所以在进行读操作时，日志区域的数据都已经写回到原来的数据区域了，所以大部分的读取操作都直接在数据区域进行，因此，HLMFS 的读性能与其他几个文件系统相当。

(2) 系统敏感性测试

首先测试了在写大小分别为 64B、128B、512B、1KB、4KB、16KB 时的事务吞吐率，并且与 PMFS、NOVA、HLMFS-NAF、HLMFS-NC 进行了对比和分析。不同写大小时的事务吞吐率测试结果如图 5.6 所示。

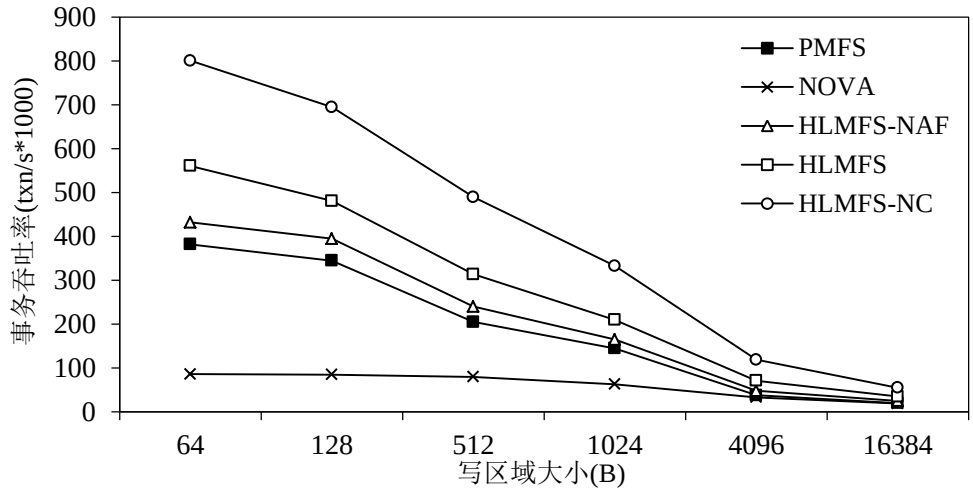


图 5.6 不同写大小时的事务吞吐率

由图可以观察到，当写区域的大小增加时，HLMFS 相对于 PMFS 和 NOVA 的性能提升比也在增加。例如，当写大小为 64B 时，HLMFS 的事务吞吐率相对于 PMFS 提升了 46.9%，是 NOVA 的 6.5 倍，这是因为 NOVA 对数据的更新采取的是页粒度的 COW，当数据更新量很小的时候，会造成写放大，因此当写区域大小小于 4KB 时，NOVA 的性能都非常低。当写区域大小为 16KB 时，HLMFS 相对于 PMFS 和 NOVA 分别提升 75%和 84.2%，原因是当写区域较小的时候，软件开销的影响会比较大，软件开销主要有系统调用、二级日志索引结构的开销，因此性能提升较少。而当写区域比较大的时候，系统的主要开销为持久化开销，此时软件开销占比较小，因此，当写区域比较大的时候，性能提升较为明显。

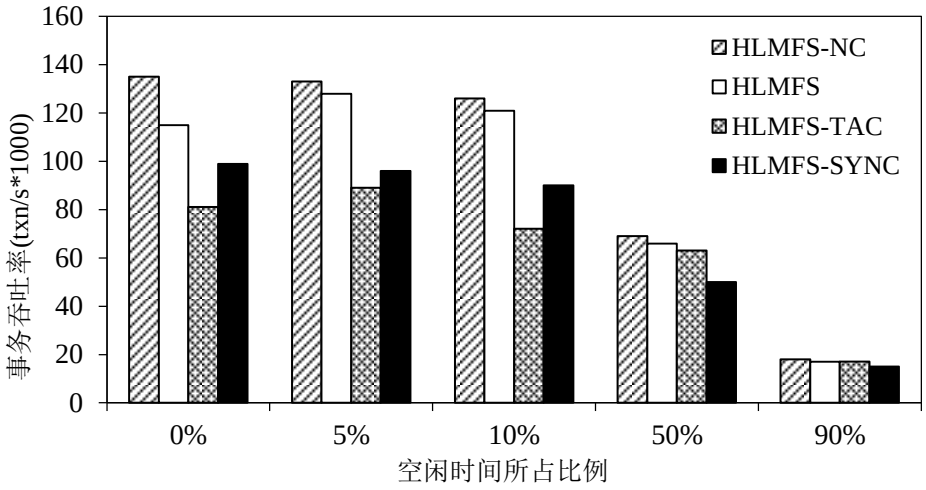


图 5.7 事务吞吐率与系统空闲时间的关系

其次，为了证明 HLMFS 能够充分利用系统空闲时间进行 checkpoint 操作使得 checkpoint 的开销达到最小，从而不会对系统的事务处理性能产生很大的影响，测试了系统空闲时间从 0%（没有空闲时间）到 90%的事务吞吐率。把 HLMFS 与 HLMFS-NC、HLMFS-TAC 和 HLMFS-SYNC 进行了对比，HLMFS-SYNC 是使用优化的 checkpoint 过程进行同步 checkpoint 的情况。测试结果如图 5.7 所示，由测试结果可以看出，系统空闲时间越少，HLMFS 相对于其他文件系统的性能提升越明显。当系统空闲时间高于 50%的时候，HLMFS 和 HLMFS-TAC 比 HLMFS-SYNC 提高了 26%到 32%，这是因为系统空闲时间足够长，后台的 checkpoint 线程的 checkpoint 操作基本上都能够在系统空闲时完成，即使是严格按照提交顺序进行也是如此，因此对主线程的事务处理操作产

生的影响很小，然而 HLMFS-SYNC 的 checkpoint 操作是在事务提交之前由主线程完成的，因此，HLMFS-SYNC 的事务处理性能比使用异步 checkpoint 过程的 HLMFS 和 HLMFS-TAC 都要低。当系统空闲时间低于 10%的时候，HLMFS 比 HLMFS-TAC 性能提高了 25%，因为当系统空闲时间比较少的时候，系统的性能就会被 checkpoint 过程中的持久化操作所限制，而 HLMFS-TAC 采取的严格按照提交顺序把所有需要 checkpoint 的数据区域都拷贝回去并且持久化，这种操作需要持久化的数据量很大，在系统空闲时不能全部完成，因此会影响到主线程的事务处理过程，对系统性能产生了很大的影响。HLMFS 通过并行 checkpoint 以及优化的 checkpoint 过程使得每次拷贝和持久化的数据量达到了最小，从而消除了系统空闲时间比较小时的性能瓶颈，能充分利用系统空闲时间完成 checkpoint 过程。因此，当系统空闲时间比较小的时候，HLMFS 的性能比 HLMFS-TAC 和 HLMFS-SYNC 都要高。当空闲时间为 0%的时候，性能分别提高了 42%和 16.2%，当空闲时间为 5%的时候，性能分别提高了 43.8%和 33.3%。

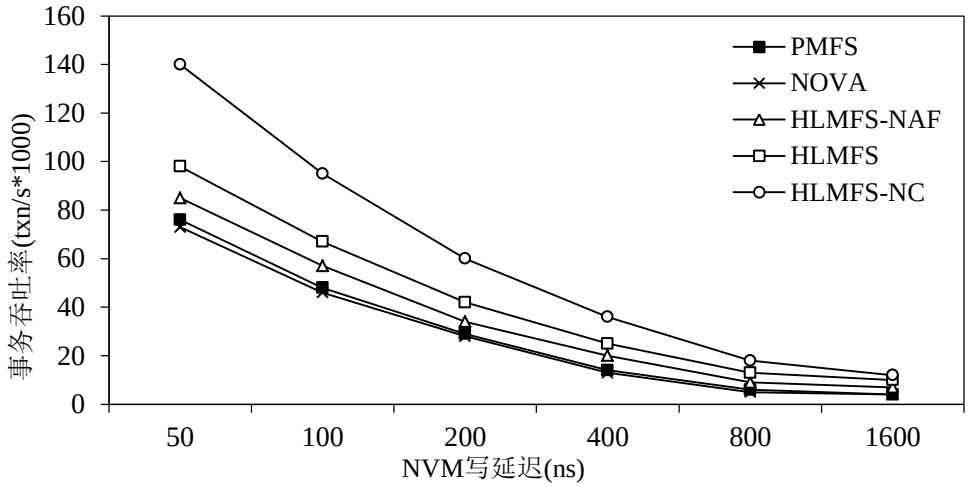


图 5.8 事务吞吐率与 NVM 写延迟的关系

最后，测试了 NVM 写延迟从 50ns 到 1600ns 变化时的事务吞吐率。测试结果如图 5.8 所示，随着 NVM 写延迟的增大，所有文件系统的事务吞吐率都呈下降趋势。测试还发现，当 NVM 写延迟增大时，HLMFS 的性能与 HLMFS-NC 的性能越来越接近，这是因为当写延迟比较小的时候，软件开销占比比较大，软件开销对系统性能影响较大。当写延迟较大的时候，数据的持久化开销成为主要开销，此时，软件开销的影响可以忽略，因此当写延迟较大的时候，HLMFS 的性能提升比较明显。当写延迟为 50ns

时，相对于 PMFS、NOVA、HLMFS-NAF 分别提高 28.9%、34.2%、15.3%，当写延迟为 1600ns 时，HLMFS 的事务吞吐率是 PMFS 和 NOVA 的 1.7 倍，相对于 HLMFS-NAF 提高了 42.9%。

(3) 系统崩溃后恢复时间测试

为了研究日志区域（包括日志项区域和混合的日志和数据区域）的大小对系统恢复时间的影响，把 NVM 区域的大小从 1GB 到 5GB 进行变换，通过测试系统崩溃后的恢复时间来进行。把 HLMFS 与 HLMFS-TR 进行对比，HLMFS-TR 是在系统恢复阶段对未完成事务进行回滚，对已经完成的事务进行 checkpoint 操作，HLMFS 是在系统恢复阶段只对未完成事务进行回滚，对已经完成的事务只构建索引，不进行 checkpoint 操作，checkpoint 操作在系统空闲时进行。

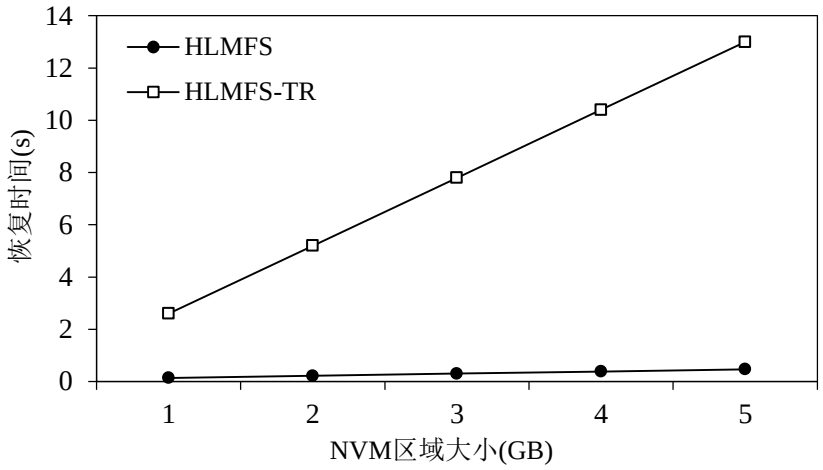


图 5.9 系统崩溃后的恢复时间测试

测试结果如图 5.9 所示。可以观察到，HLMFS 和 HLMFS-TR 的系统恢复时间都是随着 NVM 区域的增大呈线性增长，这是因为当日志区域增大时，需要进行 checkpoint 的数据量也会增大，无论是立即进行 checkpoint 操作的 HLMFS-TR 还是只对需要 checkpoint 的日志项构建索引的 HLMFS 的恢复时间都会随着日志区域的增大而增大。但是 HLMFS-TR 的恢复时间远高于 HLMFS 的恢复时间。例如，当 NVM 区域的大小是 5GB 的时候，HLMFS 的恢复时间是 470ms，而 HLMFS-TR 的恢复时间是 13s，因为 HLMFS 把对已提交事务的 checkpoint 操作延迟进行了，checkpoint 操作过程中大量的持久化操作对系统性能有很大的影响，因此，HLMFS-TR 的恢复时间远高于 HLMFS。

对系统崩溃后的恢复时间的测试结果说明，采取混合日志机制的文件系统能够实现系统崩溃后数据的快速恢复。

5.3.2 Filebench 测试结果和分析

使用测试工具 filebench1.5 对系统的吞吐率进行测试，选择负载 fileserver，创建 10000 个文件，每个文件大小 128KB，读写块大小为 1MB，追加写块大小为 64KB，负载运行 60s。使用吞吐率作为性能衡量指标。

单线程测试结果如图 5.10 所示，HLMFS 的性能比 PMFS、NOVA、HLMFS-NAF 分别提高了 1.8 倍、1.7 倍和 1.4 倍。多线程测试结果如图 5.11 所示，随着线程数的增加，系统的吞吐率也在增加，当测试线程数为 10 个线程时，HLMFS 的性能相对 PMFS、NOVA、HLMFS-NAF 分别提高了 1.6 倍、1.4 倍和 1.3 倍。由测试结果图可以看出，当测试 2 个线程时，HLMFS 相对于 HLMFS-NAF 的性能提升比比单线程时的性能提升比有明显的增加，但是当线程数增大到 6 个线程之后，HLMFS 相对于 HLMFS-NAF 的性能提升比就不再有明显的增加了。这是因为机器的 CPU 资源有限，用户线程和并行刷新线程并不能都调度服务器的物理资源，机器的物理核个数为 12 个，因此测试线程数小于 6 个时，用户线程和并行刷新线程可以利用相同个物理核资源，性能提升比也一直在增加，而当测试线程数超过 6 个之后，会出现 CPU 资源竞争的情况，因此，性能提升比不再有明显的增加。

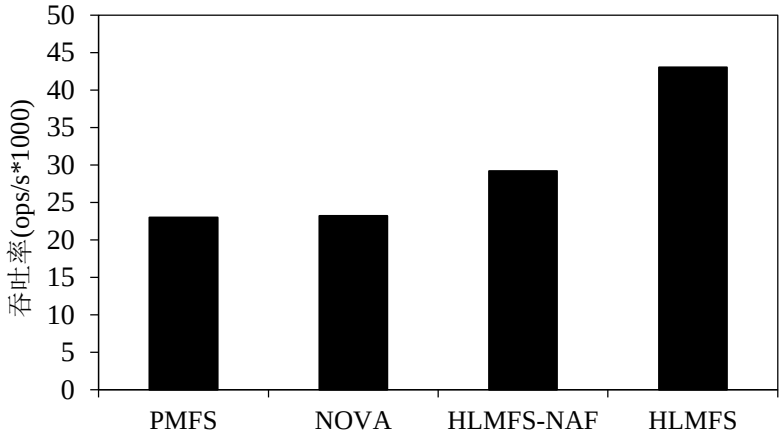


图 5.10 单线程负载 fileserver 测试结果

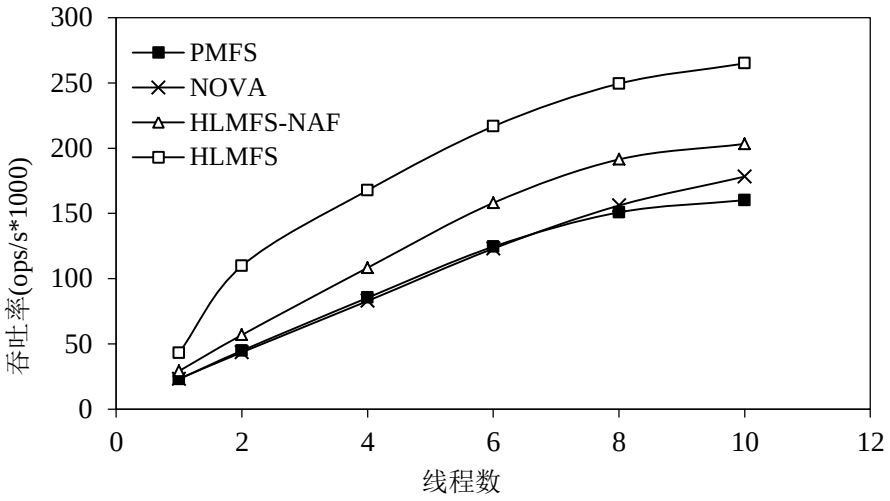


图 5.11 多线程负载 fileserver 测试结果

5.3.3 实际应用的测试结果和分析

在本节中，对实际应用 SQLite 数据库和 Tokyo Cabinet 数据的性能进行了测试，把基于 HLMFS、HLMFS-NAF 和 HLMFS-NC 的数据库以及传统的 SQLite 和 Tokyo Cabinet 数据库的性能进行了对比。传统的数据库是指没有对数据库进行过任何修改，数据库自带有各自的一致性保障机制，运行于 PMFS 之上。基于 HLMFS-NC 的数据库是去除了数据库和底层文件系统中的一致性保障机制，不对数据库中的数据进行一致性保证。基于 HLMFS 的数据库是去除了数据库中的一致性保障机制，用底层文件系统的混合日志机制来保证数据库数据的一致性。

SQLite 是嵌入式的 SQL 数据库，被广泛用于手机等移动计算系统，SQLite 自带两种一致性保障机制供用户灵活选择：回滚式日志（RollBack Journaling，RBJ）、预写式日志（Write-Ahead Logging，WAL）。在 RBJ 模式下，SQLite 会把更新前的数据副本拷贝到一个单独的日志文件中，然后再把新数据写入数据库文件，当发生系统崩溃时使用数据副本进行回滚操作。在 WAL 模式下，SQLite 直接把新数据写入日志文件，旧数据仍保留在原来的数据库文件中，发生系统崩溃时直接把日志文件中的新数据丢弃即可。对于数据库 SQLite，使用测试工具 TPC-C 来测试其性能。TPC-C 执行 10000 次数据库更新操作，每次更新大小是从 0 到 16KB 中随机选择。SQLite 默认使用 4KB 的

页面。测试结果如图 5.12 所示，发现基于 HLMFS 的 SQLite 性能比传统的 RBJ 和 WAL 的性能分别提高了 80%和 48.7%。因为使用 WAL 的 SQLite 默认当日志区域达到 1000 页时进行 checkpoint 操作，此时的 checkpoint 操作由主线程完成，因此对系统的事务处理性能造成了影响，而 HLMFS 的 checkpoint 操作是异步并行的进行的，因此基于 HLMFS 的 SQLite 的性能会比使用 WAL 和 RBJ 的 SQLite 的性能要好。

数据库 Tokyo Cabinet 是一种 DBM 开发库，它的数据文件只有一个，里面存放多个数据记录，每个数据记录由键值对组成，键值可以是任意长度的字节序列，Tokyo Cabinet 中没有数据类型和数据表的概念，Tokyo Cabinet 自带的一致性保障机制是 undo 日志，把旧数据写入日志区域后，再把新数据写入数据库中。对于 Tokyo Cabinet，使用它自带的测试工具 tctreetest 来测试其性能，向数据库中随机插入 10000 个键值对，每个键值对包含 0 到 16KB 的数据。测试结果如图 5.12 所示。基于 HLMFS 的 Tokyo Cabinet 的性能比传统的 Tokyo Cabinet 提高了 69%。

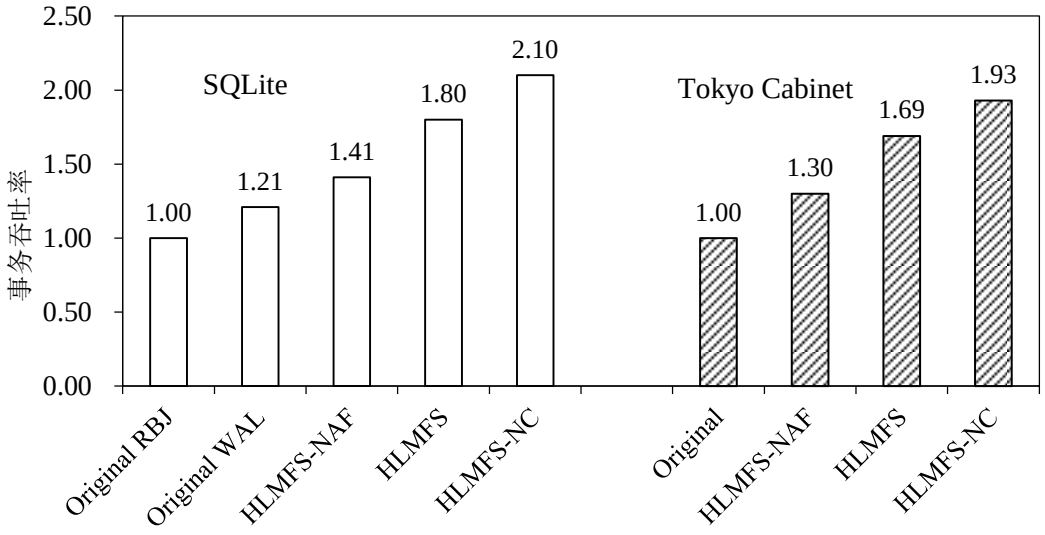


图 5.12 SQLite 和 Tokyo Cabinet 测试结果

数据库 SQLite 和 Tokyo Cabinet 的测试结果表明混合日志机制能够提供应用程序级别的数据一致性保护，运行于 HLMFS 之上的 SQLite 和 Tokyo Cabinet 只需要调用 HLMFS 提供的事务编程接口就可以保证数据的一致性，不再需要自己设计复杂的一致性保障机制，在消除了数据库中冗余的一致性保障机制之后，数据库的性能也得到了

很大的提升。

5.4 本章小结

本章主要对使用混合日志机制的系统 HLMFS 进行了测试，功能测试证明了混合日志机制确实能够保证文件系统的一致性。在性能测试中，选择 PMFS、NOVA、HLMFS-NAF、HLMFS-NC 等系统作为对照，分别测试了单线程和多线程情况下的事务处理性能，测试结果表明 HLMFS 的事务处理性能都高于 PMFS、NOVA 和 HLMFS-NAF。之后又测试了系统的事务吞吐率与写大小、系统空闲时间以及 NVM 写延迟的关系，这三种情况下的测试结果都比较理想。然后又测试了系统崩溃后的恢复时间，证明了系统所采取的快速恢复操作的高效性。除此之外，还用标准测试工具 Filebench 对系统的吞吐率进行了测试，测试结果表明 HLMFS 的吞吐率比 PMFS、NOVA 和 HLMFS-NAF 都要高。最后，测试了实际应用 SQLite 和 Tokyo Cabinet 数据库在运行于 HLMFS 之上时的性能，发现与原始的数据库相比，性能也有明显的提升。本章的测试结果表明混合日志机制在保证文件系统的一致性的基础上，在很大程度上提升了文件系统的事务处理性能，实现了系统崩溃后数据的快速恢复，而且运行于 HLMFS 之上的应用程序的性能也得到了提升。

6 总结与展望

6.1 论文总结

传统的基于磁盘的文件系统的一致性保障机制不适用于 NVM 文件系统，而当前具有代表性的 NVM 文件系统的一致性保障机制都存在很多不足之处，对文件系统以及上层应用的性能造成了一定的影响。针对上述问题，本文研究实现了高性能的混合日志机制，在保证文件系统的元数据和数据的一致性的基础上，能够实现系统崩溃后数据的快速恢复，同时在很大程度上提高了文件系统的性能，此外，还提供了应用程序级别的数据一致性保护，提升了上层应用的性能。主要工作包括以下几点：

(1) 深入研究了现有的 NVM 文件系统 BPFS、SCMFS、Aerie、PMFS 和 NOVA 的一致性保障机制，并对它们各自的优缺点进行了分析和总结。

(2) 深入细致的讨论了 NVM 存储架构以及 NVM 文件系统的数据一致性的含义，详细介绍了日志机制包括 undo 和 redo 日志机制的执行流程，并对 undo 和 redo 日志的主要性能开销进行了分析和对比。

(3) 通过分析元数据和数据各自的更新特性以及 undo 日志和 redo 日志的主要开销，设计实现了混合日志机制，解耦了元数据和数据，元数据采取字节粒度的 undo 日志，数据采取混合粒度的 redo 日志。并对 redo 日志构建二级日志索引结构，提高日志查询性能。针对 redo 日志设计实现了最优并行 checkpoint 过程，将 checkpoint 过程中大量的数据持久化操作并行异步的进行，并且使得持久化操作的数据量最小化。针对事务中大量的刷新操作，设计实现了并行刷新过程，使得一个事务中的刷新操作与数据处理操作并行进行，提高了系统写性能。此外，把事务创建与事务提交功能模块抽象成编程接口以系统调用的形式供上层应用程序调用，提供了应用程序级别的数据一致性保护，消除了上层应用中冗余的一致性保障机制，从而提高了上层应用的性能。

(4) 根据混合日志机制的设计给出了系统中几个重要的功能模块的详细实现方案，其中包括初始化模块、写操作处理模块、数据写回模块、系统恢复模块、事务交互模块的实现。

(5) 对使用混合日志机制的文件系统 HLMFS 进行了功能测试和性能测试。功能测试结果表明混合日志机制能够保证文件系统的一致性。性能测试使用自己编写的事务吞吐率测试程序和标准测试工具 Filebench1.5 对系统的读写性能、事务处理性能以及系统崩溃后的恢复时间进行了测试，并且与 HLMFS-NAF、HLMFS-NC、PMFS 和 NOVA 进行了对比，测试结果表明 HLMFS 的性能较 HLMFS-NAF、PMFS 和 NOVA 都有大幅度的提高。此外，还分别使用测试工具 TPC-C 和 tctreetest 对运行在文件系统之上的实际应用 SQLite 和 Tokyo Cabinet 进行了测试，发现运行于 HLMFS 之上的应用程序的性能比运行于 PMFS 之上的性能更高，说明混合日志机制在提供应用程序级别的数据一致性保护的基础之上，能够提高上层应用的整体性能。

6.2 展望

混合日志机制能够同时保证元数据和数据的一致性，实现系统崩溃后数据的快速恢复，还通过一些策略在很大程度上提高了文件系统的读写性能，此外，混合日志机制还提供了应用程序级别的数据一致性保护，使得运行于 HLMFS 之上的应用程序的性能也得到了大幅度提升。但是，混合日志机制目前还存在以下两点可以改进的地方：

(1) HLMFS 只提供了一种一致性保障机制，能够同时保证元数据和数据的一致性。但是，对于有些要求不高的应用程序来说，并不需要对元数据和数据都提供一致性保障，严格的一致性保障机制反而会对应用程序的性能造成影响。因此，可以在文件系统中设计实现两种一致性保障机制：同时保证元数据和数据的一致性的机制和只保证元数据的一致性的机制。由上层应用选择使用何种程度的一致性保障机制，在上层应用调用创建事务的接口函数时，把所选择的一致性保障机制通过参数传递给下层的文件系统，文件系统根据参数选择提供不同程度的一致性保护。

(2) HLMFS 提供了事务编程接口给应用程序使用，运行在 HLMFS 之上的应用程序必须修改自身的代码，这使得上层应用的编程复杂性变高。为解决这个问题，可以将创建事务和提交事务函数封装在文件系统的写函数 write()中，这样当上层应用每次调用文件系统写函数，都会在文件系统层对应的创建一个事务，应用程序不需要显示

的调用创建事务和提交事务的函数，因此，应用程序也就不需要再修改自身的代码了。但是，这种方法存在一个问题，使用这种方法应用程序每调用一次写函数都会创建一个事务，即一个事务中只能包含一次写操作，不能把多次写操作包含到一个事务中，也就不能进行多次写操作的合并，会对应用程序的性能造成一定的影响。

致　谢

时光如白驹过隙，转眼间，三年的研究生生活即将结束。回首往昔，刚入学时的兴奋感、选取研究方向时的迷茫、遇到问题时的挫败感、取得研究成果时的惊喜，都历历在目。这三年来，自己的不断成长除了自己的努力之外，更离不开师长、同学和父母的帮助。在这里，感谢所有帮助过我的人。

首先，要感谢我的导师陈俭喜老师。陈老师在学习和科研上都给了我很多的建议，在刚接触到我们组的研究方向，感到非常迷茫时，是陈老师的悉心指导让我一步一步的确定了自己的研究方向。在每周一次的例会上，陈老师也会对我在科研中遇到的阻碍和下一步的研究计划给出具体的建议。在做毕业设计和撰写毕业论文的过程中，陈老师也花费了大量的时间对我进行了详细的指导。在科研方面，陈老师积极的科研态度也是我学习的榜样，对例会、考勤、打扫卫生等实验室各项事务的安排也为我们提供了良好的科研工作环境。

还要感谢冯丹老师，冯丹老师作为国光存储实验室的领导者，不断的为实验室辛勤付出，我们才能有来之不易的科研工作环境。

其次，要感谢余亚博士。余亚博士是我们组科研工作的领头羊。在三年的科研生活中，余亚博士对我的帮助也非常大。从研究方向的确定到毕业设计的实现，每一个阶段都离不开余亚博士对我的指导。每次在科研上遇到阻碍，小到服务器故障，大到代码错误，余亚博士都不厌其烦的帮我解决问题。余亚博士的学习方式和方法给了我很大的启发，他对待科研的严谨努力的态度和坚持不懈的毅力也激励着我不断的努力。

还要感谢跟我同组的张娟同学，在科研中遇到阻碍时，与张娟同学的交流经常能让我打开新的思路，在找工作遇到挫折时，张娟同学也给予了我很大的鼓励和陪伴。感谢跟我同宿舍的雷霞同学，在生活中对我的关照和帮助。还要感谢实验室的张铮学长，张晓祎学长，胡永恒学长，张争宏学长，廖雪琴学姐，刘云同学，刘亚春同学，方才华同学等，因为你们的帮助，我才能变得更优秀。

最后要感谢我的父母，正是因为他们对我的鼓励和关心，我才能更专注于学习，更顺利的完成学业。

参考文献

[1] Pinheiro E, Weber W D, Barroso L A. Failure Trends in a Large Disk Drive Population. in: Proceedings of the 5th USENIX Conference on File and Storage Technologies, San Jose, CA, 2007. 17~28

[2] 侯斌兵. 基于存储级内存的块设备日志机制 SJBD 的设计与实现: [硕士学位论文]. 武汉: 华中科技大学, 2014

[3] Zhang Y, Swanson S. A study of application performance with non-volatile main memory. in: Mass Storage Systems and Technologies (MSST). 2015 31st Symposium on: IEEE, 2015. 1~10

[4] 张进涛. 基于多阵列、多通道的存储服务器关键技术研究: [硕士学位论文]. 武汉: 华中科技大学, 2004

[5] Lam, C. H. Storage class memory. in: Proceedings of the 10th IEEE International Conference on Solid-State and Integrated Circuit Technology. Seoul, Korea, 2010. 1080~1083

[6] Burr G W, Kurdi B N, Scott J C, et al. Overview of candidate device technologies for storage-class memory. IBM Journal of Research and Development, 2008, 52(4.5):449~464

[7] Freitas R F, Wilcke W W. Storage-class memory: The next storage system technology. IBM Journal of Research and Development, 2008, 52(4.5): 439~447

[8] Zhang Y, Yang J, Memaripour A, et al. Mojim: A reliable and highly-available non-volatile memory system. ACM SIGPLAN Notices, 2015, 50(4): 3~18

[9] Arulraj J, Pavlo A, Dulloor S R. Let's talk about storage & recovery methods for non-volatile memory database systems. in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015. 707~722

[10] Lu Y, Shu J, Sun L, et al. Loose-ordering consistency for persistent memory. in: Computer Design (ICCD). 2014 32nd IEEE International Conference on. IEEE, 2014.

216~223

[11] Ou J, Shu J. Fast and failure-consistent updates of application data in non-volatile main memory file system. in: Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on. IEEE, 2016. 1~15

[12] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. in: Proceedings of the 11th ACM Symposium on Operating Systems Principles, Austin, Texas, 1987. 155~162

[13] Tweedie, S. C. Journaling the Linux ext2fs filesystem. in: Proceeding of the 4th Annual Linux Expo, Southern California,1998. 1~8

[14] Wan H, Lu Y, Xu Y, et al. Empirical study of redo and undo logging in persistent memory. in: Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016 5th. IEEE, 2016. 1~6

[15] Coburn J, Caulfield A M, Akel A, et al. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ACM Sigplan Notices, 2011, 46(3): 105~118

[16] Mohan C, Haderle D, Lindsay B, et al. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems (TODS), 1992, 17(1): 94~162

[17] Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory. in: Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014. 15

[18] Xu J, Swanson S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. in: FAST. 2016. 323~338

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

[20] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory. in: Proceedings of the ACM SIGOPS 22nd symposium on

Operating systems principles. ACM, 2009. 133~146

[21] Pillai T S, Chidambaram V, Alagappan R, et al. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. in: OSDI. 2014, 433~448

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

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

[24] Pillai T S, Chidambaram V, Alagappan R, et al. Crash consistency. Queue, 2015, 13(7): 20

[25] Verma R, Mendez A A, Park S, et al. Failure-Atomic Updates of Application Data in a Linux File System. in: FAST. 2015. 203~211

[26] Pelley S, Chen P M, Wenisch T F. Memory persistency. Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on. IEEE, 2014: 265~276

[27] Kolli A, Pelley S, Saidi A, et al. High-performance transactions for persistent memories. ACM SIGPLAN Notices, 2016, 51(4): 399~411

[28] Kim W H, Kim J, Baek W, et al. Nvwal: exploiting nvram in write-ahead logging. ACM SIGPLAN Notices. ACM, 2016, 51(4): 385~398

[29] Wang Z, Yi H, Liu R, et al. Persistent transactional memory. IEEE Computer Architecture Letters, 2015, 14(1): 58~61

[30] Kannan S, Qureshi M, Gavrilovska A, et al. Energy aware persistence: Reducing the energy overheads of persistent memory. IEEE Computer Architecture Letters, 2016, 15(2): 89~92

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

[32] NARAYANAN D, HODSON O. Whole-system Persistence with Non-volatile Memories. Proc. ACM ASPLOS'12

[33] Ousterhout J, Gopalan A, Gupta A, et al. The RAMCloud storage system. ACM Transactions on Computer Systems (TOCS), 2015, 33(3): 7

[34] Park S, Kelly T, Shen K. Failure-atomic msync (): A simple and efficient mechanism for preserving the integrity of durable data. in: Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013. 225~238

[35] Pelley S, Chen P M, Wenisch T F. Memory persistency. in: Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on. IEEE, 2014. 265~276

[36] Santana R, Rangaswami R, Tarasov V, et al. A fast and slippery slope for file systems. ACM SIGOPS Operating Systems Review, 2016, 49(2): 27~34

[37] Venkataraman S, Tolia N, Ranganathan P, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In: FAST. 2011. 61~75

[38] Yang J, Wei Q, Chen C, et al. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. in: FAST. 2015. 167~181

[39] Volos H, Tack A J, Swift M M. Mnemosyne: Lightweight persistent memory. ACM SIGARCH Computer Architecture News. ACM, 2011, 39(1): 91~104

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

[41] Volos H, Nalli S, Panneerselvam S, et al. Aerie: Flexible file-system interfaces to storage-class memory. in: Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014. 14

[42] Xiao Y L F L N, Zhu J Z L. SNFS: Small Writes Optimization for Log-Structured File System Based-on Non-Volatile Main Memory. High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS),

2017 IEEE 19th International Conference on. IEEE, 2017. 89~97

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

[44] Sengupta D, Wang Q, Volos H, et al. A framework for emulating non-volatile memory systemswith different performance characteristics. in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. ACM, 2015. 317~320

[45] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. in: Proceedings of the 11th ACM Symposium on Operating Systems Principles, Austin, Texas, 1987. 155~162

[46] Tweedie, S. C. Journaling the Linux ext2fs filesystem. in: Proceeding of the 4th Annual Linux Expo, Southern California,1998. 1~8

[47] Kannan S, Qureshi M, Gavrilovska A, et al. Energy aware persistence: Reducing the energy overheads of persistent memory. IEEE Computer Architecture Letters, 2016, 15(2): 89~92

[48] Xu J, Swanson S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. in: FAST. 2016. 323~338