# NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories

## Abstract

快速的非易失性存储器（NVM）将很快与DRAM一起出现在处理器内存总线上。由此产生的混合内存系统将为软件提供亚微秒级、高带宽的持久性数据访问，但管理、访问和维护存储在非易失性存储器中的数据的一致性会带来一系列挑战。**现有的为旋转磁盘或固态磁盘建立的文件系统引入了软件开销**，这将掩盖NVM应该提供的性能，但为NVM提出的文件系统要么有类似的开销，要么无法提供应用所需的强大的一致性保证。

我们提出了NOVA，这是一个文件系统，旨在**最大化混合内存系统的性能**，同时提供强大的一致性保证。NOVA 采用传统的日志结构文件系统技术，以利用 NVM 提供的快速随机访问。特别是，它为每个inode维护单独的日志以提高并发性，并将文件数据存储在日志外部，以最小化日志大小并降低垃圾回收成本。NOVA的日志提供元数据、数据和mmap的原子性，并专注于简单性和可靠性，将复杂的元数据结构保存在DRAM中以加速查找操作。实验结果表明，在写密集型工作负载中，与最先进的文件系统相比，NOVA提供了22%到216倍的吞吐量改进，与提供同样强大的数据一致性保障的文件系统相比，NOVA提供了3.1倍到13.5倍的改进。

## Introduction

新兴的非易失性存储器（NVM）技术，例如自旋扭矩转移、相变、电阻式存储器[2, 28, 52]以及英特尔和美光的3D XPoint[1]技术，有望彻底改变I/O性能。研究人员已经提出了几种将NVM集成到计算机系统中的方法[11, 13, 19, 31, 36, 41, 58, 67]，最令人兴奋的建议是将NVM与传统DRAM一起放在处理器的内存总线上，从而形成混合的易失性/非易失性主存储器系统[4, 51, 72, 78]。将速度较快的易失性DRAM与速度稍慢、密度较高的非易失性主存储器（NVMMs）结合起来，就有可能形成兼具两种技术最佳特性的存储系统。

混合DRAM/NVMM存储系统为系统设计者带来了大量的机会和挑战。如果这些系统要充分利用NVMM的高性能并有效地支持更灵活的访问模式，就需要尽量减少软件开销。同时，它们必须提供应用所需的强大一致性保证，并尊重新兴存储器的限制（如有限的程序周期）。

传统的文件系统不适合混合存储系统，因为它们是为磁盘（旋转式或固态）的性能特征而建立的，**并且依靠磁盘的一致性保证（例如，扇区更新是原子性的）来保证正确性[47]**。混合存储系统在这两方面都与传统存储系统不同。NVMMs比磁盘的性能有很大的提高，而DRAM的性能甚至更好，尽管没有持久性。而且内存提供了与磁盘不同的一致性保证（例如，64位原子存储）。

对于基于内存的文件系统来说，提供强大的一致性保证尤其具有挑战性，因为在NVMM中保持数据一致性的成本很高。现代CPU和内存系统可能会重新排序存储到内存以提高性能，在系统故障时破坏一致性。为了弥补这一点，文件系统需要明确地从CPU的缓存中刷新数据以执行排序，这就增加了大量的开销，浪费了NVMM所能提供的改进性能[6, 76]。

克服这些问题是至关重要的，因为许多应用程序依靠原子文件系统操作来确保其自身的 正确性。**现有的主流文件系统使用日记。影子分页或日志结构技术来提供原子性。**然而，journal技术通过将写入存储设备的次数翻倍来浪费带宽。写入存储设备的次数，而影子分页文件系统需要从受影响的叶子节点到根的级联更新。实施这两种技术都需要严格的排序要求，从而降低性能。

日志结构文件系统（LFSs）[55]将小的写请求组合成一个较大的顺序写，硬盘和基于NAND闪存的固态硬盘（SSD）可以有效地处理。然而，传统的LFSs依赖于连续的自由区域的可用性，**而维护这些区域需要昂贵的垃圾收集操作**。因此，最近的研究[59]表明，LFSs在NVMM上的表现比日记文件系统差。

为了克服所有这些限制，我们提出了NOVA（NOn-Volatile memory Accelerated）日志结构的文件系统。NOVA调整了传统的日志结构文件系统技术，**以利用混合内存系统提供的快速随机访问**。这使得NOVA支持大规模并发，减少日志大小，并最大限度地减少垃圾收集成本，同时为传统的文件操作和基于mmap的加载/存储访问提供强大的一致性保证。

NOVA的几个方面使它与以前的日志结构文件系统不同。NOVA为每个节点分配一个单独的日志，以便在正常运行和恢复期间最大限度地提高并发性。 NOVA将日志存储为链接列表，所以它们不需要在内存中连续，而且它使用原子更新日志的尾部指针来提供原子日志追加。对于跨越多个节点的操作，NOVA使用轻量级日志。

NOVA不log数据，所以恢复过程只需要扫描NVMM的一小部分。**这也使NOVA在页面变得陈旧时立即回收**，大大减少了垃圾收集的开销，使NOVA即使在文件系统接近满的情况下也能保持良好的性能。

在描述NOVA时，本文提出了以下结论贡献：

- 它扩展了现有的日志结构文件系统技术以利用混合内存系统的特性。

- 它描述了atomic mmap，这是一个简化的接口，用于将NVMM直接暴露给具有强大一致性保证的应用。

- 它证明了NOVA优于在混合内存系统上运行的现有日记、影子分页（CoW）和日志结构的文件系统。

- 它表明NOVA在一系列建议的NVMM技术中提供了这些优势。

我们在一个基于硬件的NVMM模拟器上使用一系列微观和宏观的测试来评估NOVA。我们发现，在广泛的应用中，NOVA比现有的文件系统快得多，在写密集型工作负载中，比提供相同数据一致性保证的文件系统快3.1倍到13.5倍。我们还测量了垃圾收集和恢复的开销，我们发现NOVA在NVMM的高利用率下提供了稳定的性能，并在系统故障时快速恢复。

本文的其余部分组织如下。第2节描述了NVMMs，它们带来的挑战，以及NVMM文件系统设计的相关工作。第3节给出了NOVA架构的概述，第4节详细描述了其实现。第5节评估了NOVA，第6节是结论。

## Background

NOVA的目标是包括新兴非易失性存储器技术和DRAM在内的存储器系统。本节首先简要介绍了非易失性存储器技术以及它们给系统设计者带来的机遇和挑战。然后，我们讨论其他文件系统如何提供原子操作和一致性保证。最后，我们讨论了以前关于NVMM文件系统的工作。

### Non-volatile memory technologies

新兴的非易失性存储器技术，如自旋扭矩传输RAM（STT-RAM）[28, 42]、相变存储器（PCM）[10, 18, 29, 52]、电阻式RAM（ReRAM）[22, 62]和3D XPoint存储器技术[1]，有望提供快速、非易失性、可字节编址的存储器。Suzuki等人[63]对这些技术及其随时间的演变进行了调查。

这些存储器有不同的优点和缺点，使它们在存储器层次结构的不同部分都很有用。STT-RAM可以满足或超过DRAM的延迟，它最终可能出现在片上的最后一级缓存中[77]，但是它的大单元尺寸限制了容量和作为DRAM替代品的可行性。PCM和ReRAM比DRAM密度大，可以实现非常大的非易失性主存储器。然而，它们相对较长的延迟使得它们不太可能完全取代DRAM作为主存储器。据传，英特尔和美光最近宣布的3D XPoint内存技术就是其中之一，其性能比NAND闪存快1000倍[1]。它将出现在固态硬盘和处理器内存总线上。因此，我们期望看到混合易失性/非易失性存储器的层次结构在大型系统中变得普遍。

### Challenges for NVMM software

NVMM技术给文件系统设计者带来了若干挑战。其中最关键的是平衡存储器的性能和软件的开销，强制更新的顺序以确保一致性，以及 提供原子式更新。

Performance。NVMMs的低延迟改变了硬件和软件延迟之间的权衡。在传统的存储系统中，慢速存储设备（如磁盘）的延迟主导着访问延迟，所以软件的有效性并不重要。以前的工作表明，在快速的NVMM中，软件成本可以迅速支配内存延迟，浪费了NVMM可以提供的性能[7, 12, 68, 74]。

由于NVMM存储器具有低延迟，并将在处理器的存储器总线上，软件应该能够通过加载和存储直接访问它们。最近基于NVMM的文件系统[21, 71, 73]绕过了DRAM页面缓存，**使用一种叫做直接访问（DAX）或就地执行（XIP）的技术直接访问NVMM，避免了存储栈中NVMM和DRAM之间的额外拷贝。NOVA是一个DAX文件系统，我们希望所有的NVMM文件系统都能提供这些（或类似）功能**。我们在第2.4节描述了目前可用的DAX文件系统。

Write reordering（写重排）。现代处理器和它们的缓存层次可以重新安排存储操作以提高性能。CPU的内存一致性协议对内存更新的顺序做出了保证，但现有的模型（除了研究建议[20，46]）并没有对更新何时到达NVMM提供保证。因此，电源故障可能使数据处于不一致的状态。

具有NVMM意识的软件可以通过明确地冲刷缓存并发布内存障碍以执行写入顺序。x86架构提供了clflush结构来刷新CPU缓存线，但clflush是严格有序的，并且不必要地使缓存线失效，从而产生显著的性能损失[6, 76]。而且，clflush只向内存控制器发送数据；它并不保证数据会到达内存。内存屏障，如英特尔的mfence指令，在屏障前后强制执行内存操作的顺序，但mfence只保证所有CPU对内存有相同的看法。它没有对数据回写到NVMM的顺序施加任何约束。

Atomicity（原子性）。POSIX风格的文件系统语义要求许多操作是原子性的（即以 "全部或无 "的方式执行）。例如，POSIX重命名要求，如果操作失败，旧名称的文件和新名称的文件都不应该被改变或创建[53]。重命名文件是一个只针对元数据的操作，但是一些原子更新同时适用于文件系统元数据和数据。例如，对一个文件的追加是对文件数据的原子更新，并改变文件的长度和修改时间。许多应用程序依靠原子文件系统操作来保证自己的正确性。

存储设备通常只提供关于原子性的基本保证。磁盘提供了原子化的扇区写入，而处理器只保证8字节（或更小）、对齐的存储是原子化的。为了建立文件系统所需要的更复杂的原子性更新，程序员必须使用更复杂的技术。

### Building complex atomic operations

现有的文件系统使用各种技术，如日记、影子分页或日志结构来提供原子性的 保证。这些技术以不同的方式工作，产生不同的开销类型。

Journaling。日志（或写前日志）被广泛用于日志文件系统[24, 27, 32, 71]和数据库[39, 43]，以确保原子性。日志系统在应用之前记录所有的更新，并且在断电的情况下，复制日志以恢复系统的一致性。日志要求写两次数据：一次到日志，一次到目标位置，为了提高性能，日志文件系统通常只写日志的元数据。最近的工作提出了反向指针[17]和从耐久性中解耦排序[16]，以减少日记的开销。

Shadow paging。一些文件系统使用一种称为影子分页的写时复制机制[20, 8, 25, 54]。影子分页文件系统在很大程度上依靠其树状结构来提供原子性。在写的过程中，影子分页不是在原地修改数据，而是将受影响页面的新副本写到存储设备的一个空部分。**然后，它通过更新页面和根之间的节点，将新页面拼接到文件系统树上。由此产生的级联更新有可能是昂贵的**。

Log-structuring。日志结构文件系统（LFSs）[55, 60]最初是为了利用硬盘驱动器在顺序访问方面的高性能。LFSs在内存中缓冲随机写入，并将其转换为较大的、连续的写入磁盘，充分利用了硬盘的优势。

尽管LFS是一个优雅的想法，但有效地实现它是很复杂的，因为LFS依赖于按顺序向磁盘的连续空闲区域写入。为了确保这种区域的持续供应，LFSs不断清理和压缩日志，以回收被陈旧数据占据的空间。

日志清理增加了开销，降低了LFS的性能[3, 61]。为了减少清理开销，一些LFS设计将热数据和冷数据分开，并对每个数据应用不同的清理策略[69, 70]。固态硬盘在顺序工作负载下也表现最好[9, 14]，所以LFS技术也被应用于固态硬盘文件系统。SFS[38]根据更新的可能性对文件块进行分类，并将具有类似 "热度 "的块写入同一个日志段，以减少清理开销。F2FS[30]使用多头日志，将元数据和数据写入独立的日志中，并在磁盘利用率高的时候直接将新的数据写入脏段的空闲空间，避免频繁的垃圾收集。

RAMCloud[44]是一个基于DRAM的存储系统，它将所有数据保存在DRAM中以服务于读取，并在硬盘上保持一个持久的版本。RAMCloud对DRAM和磁盘都采用了日志结构。它以日志结构的方式分配DRAM，实现了比其他内存分配器更高的DRAM利用率[56]，并将备份数据存储在磁盘上的日志中。

### File systems for NVMM

有几个小组已经设计了基于NVMM的文件系统，通过应用第2.3节中讨论的一种或多种技术来解决第2.2节中描述的一些问题，但没有一个能满足现代应用对文件系统的所有要求。

BPFS[20]是一个影子分页文件系统，提供元数据和数据一致性。BPFS提出了一种**硬件机制**来执行存储的耐久性和排序。BPFS使用短路影子分页来减少普通情况下的影子分页开销，但是某些跨越文件系统树的大部分的操作（例如，目录之间的移动）仍然会产生很大的开销。

PMFS[21, 49]是一个轻量级的DAX文件系统，它通过绕过块层和文件系统页面缓存来提高性能。PMFS使用日志进行元数据更新。它执行就地写入，所以它们不是原子性的。（错误的，PMFS提供了强一致性）

Ext4-DAX [71]用DAX功能扩展了Ext4，以直接访问NVMM，并使用日记来保证元数据更新的原子性。普通的（非DAX）Ext4文件系统有一个数据日志模式来提供数据的原子性。Ext4-DAX不支持这种模式，所以数据更新不是原子性的。

SCMFS[73]利用操作系统的虚拟内存管理模块，将文件映射到大的连续的虚拟地址区域，使文件访问变得简单和轻便。SCMFS不提供任何元数据或数据的一致性保证。

Aerie[66]在用户空间实现了文件系统接口和功能，以提供对NVMM中数据的低延迟访问。它有一个优化，通过放松POSIX语义来提高性能。Aerie对元数据进行journal，但不支持数据原子性或mmap操作。

## NOVA Design Overview

NOVA是一个日志结构的POSIX文件系统，它建立在LFS的优势之上，并将其调整为利用混合内存系统。由于它的目标是不同的存储技术，**NOVA看起来与传统的日志结构文件系统非常不同**，后者是为了最大限度地提高磁盘带宽而建立的。

我们根据三个观察结果设计了NOVA。首先，支持原子更新的日志很容易在NVMM中正确实现，但它们对搜索操作（如目录查询和文件内的随机访问）并不高效。相反，支持快速搜索的数据结构（如树形结构）在NVMM中更难正确和有效地实现[15, 40, 65, 75]。第二，清理日志的复杂性主要源于需要提供连续的自由存储区域，但这在NVMM中是不必要的，因为随机访问很便宜（AEP的随机访问还是有性能差距）。第三，使用单一日志对磁盘来说是有意义的（那里有一个磁盘头，提高空间定位性是最重要的），但它限制了并发性。由于NVMM支持快速、高并发的随机访问，使用多个日志并不会对性能产生负面影响。

基于这些观察，我们在NOVA中做出了以下设计决定。

**Keep logs in NVMM and indexes in DRAM**. NOVA将日志和文件数据保存在NVMM中，并在DRAM中建立radix树[35]以快速执行搜索操作，使NVMM中的数据结构简单而高效。我们使用radix树是因为在Linux内核中有一个成熟的、经过测试的、广泛使用的实现。弧形树的叶子指向日志中的条目，而这些条目又指向文件数据。

**Give each inode its own log** NOVA中的每个节点都有自己的日志，允许跨文件的并发更新而不需要同步。这种结构允许在文件访问和恢复过程中实现高并发，因为NOVA可以同时重放多个日志。NOVA还保证有效的日志条目数量很小（与文件中的extents数量相当），这确保了扫描日志的速度。

**Use logging and lightweight journaling for complex atomic updates**. NOVA是日志结构的，因为这提供了比journaling和影子分页更便宜的原子更新。**为了原子地将数据写入日志，NOVA首先将数据存入日志，然后原子地更新日志tail以提交更新**，从而避免了日记文件系统的重复写入开销和影子分页系统的级联更新成本。

一些目录操作，如目录之间的移动，跨越了多个节点，NOVA使用journaling来原子化地更新多个日志。NOVA首先在每个节点的日志末尾写入数据，然后对日志尾部的更新进行journaling，以原子化地更新它们。NOVA的日志是轻量级的（相对于文件数据或元数据），因为它只涉及到日志尾部，并且没有POSIX文件操作在四个以上的节点上进行。

**Implement the log as a singly linked list**. 在基于NVMM的存储中，顺序日志的局部性优势不太重要（在AEP中并不是这样，需要测试最佳的log大小），所以NOVA使用4KB的NVMM页面的链接列表来保存日志，并在每个日志页面的末尾存储下一个页面指针。

允许非连续的日志存储有三个优点。首先，分配日志空间很容易，因为NOVA不需要为日志分配大的、连续的区域。第二，NOVA可以在细粒度、页大小的粒度上进行日志清理。第三，回收只包含陈旧条目的日志页只需要几个指针的赋值。

**Do not log file data.** NOVA的inode日志不包含文件数据。相反，NOVA对修改过的页面使用写时拷贝，并将关于写入的元数据附加到日志中。**元数据描述更新并指向数据页**。第4.4节更详细地描述了文件写入操作。

**对文件数据使用写时拷贝是很有用的，有几个原因。首先，它导致了更短的日志，加速了恢复过程。**第二，它使垃圾收集更简单、更有效，因为NOVA从来不需要把文件数据从日志中复制出来以恢复一个日志页。第三，回收陈旧的页面和分配新的数据页面都很容易，因为它们只需要从DRAM内的空闲列表中添加和删除页面。第四，由于它可以立即回收陈旧的数据页，NOVA甚至在沉重的写入负载和高NVMM利用率水平下也能保持性能。

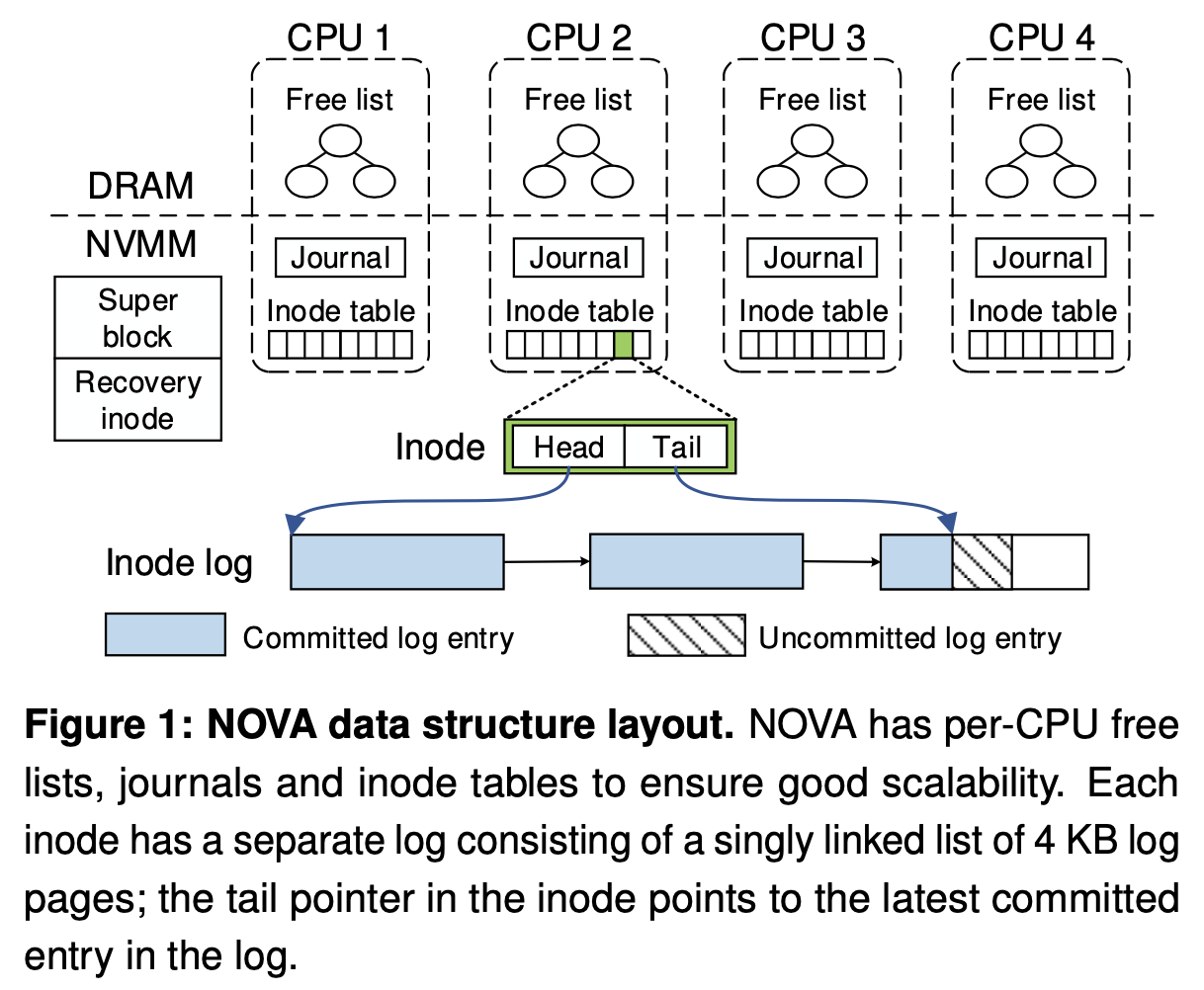
下一节将更详细地描述NOVA的实现。

## Implementing NOVA

我们已经在Linux内核4.0版本中实现了NOVA。NOVA使用内核中现有的NVMM钩子（hooks），并通过了Linux POSIX文件系统测试套件[50]。源代码可以在GitHub上找到：<https://github.com/NVSL/NOVA> 。在这一节中，我们首先描述了整个文件系统的布局及其原子性和写入顺序机制。然后，我们描述NOVA如何执行原子目录、文件和mmap操作。最后我们讨论NOVA的垃圾收集、恢复和内存保护。

### NVMM数据结构和空间管理

图1显示了NOVA数据结构在它所管理的NVMM区域的高级布局。NOVA将NVMM分为四个部分：超级块和恢复节点inode、节点inode表、journals和日志/数据页。超级区块包含全局文件系统信息，恢复节点存储恢复信息，加速NOVA在清洁关闭后的重挂（见第4.7节），节点表包含节点inode，journals为目录操作提供原子性，其余区域包含NVMM日志和数据页。我们在设计NOVA时考虑到了可扩展性。NOVA在每个CPU维护一个节点表、journal和NVMM空闲页列表，以避免全局锁定和扩展性瓶颈。



**Inode table** NOVA将每个inode表初始化为一个2MB的inode块阵列。每个NOVA的**节点在128字节的边界上对齐**，所以给定的节点号NOVA可以很容易地找到目标节点。NOVA以轮流的顺序向每个节点表分配新的节点，这样节点就会在节点表之间均匀地分布。如果节点表已满，NOVA通过建立一个2MB的子表的链接列表来扩展它。为了减少节点表的大小，每个NOVA的节点都包含一个有效位，NOVA为新的文件和目录重新使用无效的节点。每CPU节点表避免了节点分配的争论，并允许在故障恢复时进行并行扫描。

一个NOVA的节点包含指向其日志头部和尾部的指针。日志是一个4KB页面的链接列表，尾部总是指向最新提交的日志条目。当系统第一次访问inode时，**NOVA从头到尾扫描日志以重建DRAM的数据结构**。

**Journal** NOVA的journal是一个4KB的循环缓冲区，NOVA用<enqueue, dequeue>指针对管理每个日志。为了协调跨越多个inodes的更新，NOVA首先将日志条目追加到每个日志，然后通过将所有受影响的日志尾部追加到当前CPU的日志enqueue开始一个事务，并更新enqueue指针。在将更新传播到目标日志尾部后，NOVA更新dequeue等于enqueue以提交事务。对于一个创建操作，NOVA对目录的日志尾部指针和新节点的有效位进行journal。在断电恢复期间，NOVA检查每个journal并回滚journal的dequeue和enqueue之间的任何更新。NOVA在每个内核上一次只允许一个开放的事务，每个CPU的日志允许并发的事务。对于每个目录操作，内核的虚拟文件系统（VFS）层会锁定所有受影响的节点，所以并发的事务不会修改同一个节点。

**NVMM space management** 为了使NVMM分配和取消分配的速度更快，NOVA将NVMM分成池子，每个CPU一个，并在DRAM中保持空闲的NVMM页面列表。如果当前CPU的池中没有可用的页，NOVA从最大的池（其他cpu池子）中分配页，并使用每个池的锁来提供保护。这种分配方案类似于Hoard[5]这样的可扩展内存分配器。为了减少分配器的大小，NOVA使用红黑树来保持空闲列表按地址排序，允许有效的合并并提供O(logn)的去分配。为了提高性能，NOVA在运行期间不在NVMM中存储分配器状态。**在正常关机时，它将分配器状态记录到恢复节点的日志中**，并在断电时通过扫描所有节点的日志恢复分配器状态。

NOVA积极地分配日志log空间，以避免经常调整日志的大小。最初，**一个节点的日志包含一个页面。当日志耗尽可用空间时，NOVA分配足够的新页，使日志空间增加一倍，并将它们附加到日志中。如果日志长度超过一个给定的阈值，NOVA每次都会追加一个固定数量的页面。（和vector类似）**

### 原子性和强制写入顺序

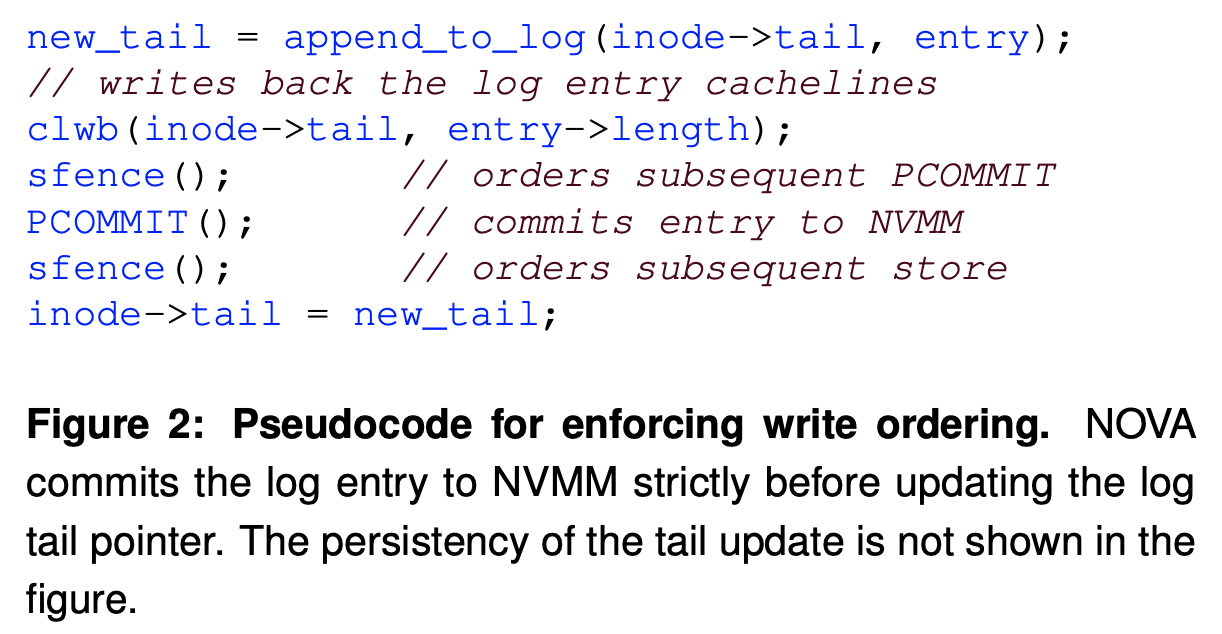
NOVA 使用结合了日志结构和journal的技术为元数据、数据和 mmap 更新提供快速原子性。 该技术使用三种机制。

**64-bit atomic updates** 现代处理器支持易失性内存的 64 位原子写入，NOVA 假设对 NVMM 的 64 位写入也将是原子的。 NOVA 使用 64 位就地写入来直接修改某些操作的元数据（例如，文件的读取时间），并使用它们通过更新 inode 的日志尾指针来提交对日志的更新。

**Logging** NOVA 使用 inode 的日志来记录修改单个 inode 的操作。 其中包括诸如 write、msync 和 chmod 之类的操作。logs彼此独立。

**Lightweight journaling** 对于需要更改多个 inode 的目录操作（例如，创建、取消链接和重命名），NOVA 使用轻量级日志来提供原子性。 在任何时候，任何 NOVA 日志中的数据都很小——不超过 64 个字节：最复杂的 POSIX 重命名操作涉及多达四个 inode，而 NOVA 只需要 16 个字节来记录每个 inode：8 个字节用于日志尾指针的地址和8 个字节的值。

**Enforcing write ordering**。NOVA 依靠三个写入顺序规则来确保一致性。 首先，它在更新日志尾部之前将数据和日志条目提交到 NVMM。 其次，它在传播更新之前将journal数据提交给 NVMM。 第三，它在回收旧版本之前将新版本的数据页提交给 NVMM。 如果 NOVA 在支持 clflushopt、clwb 和 PCOMMIT 指令的系统上运行，它会使用图 2 中的代码来强制执行写入顺序。



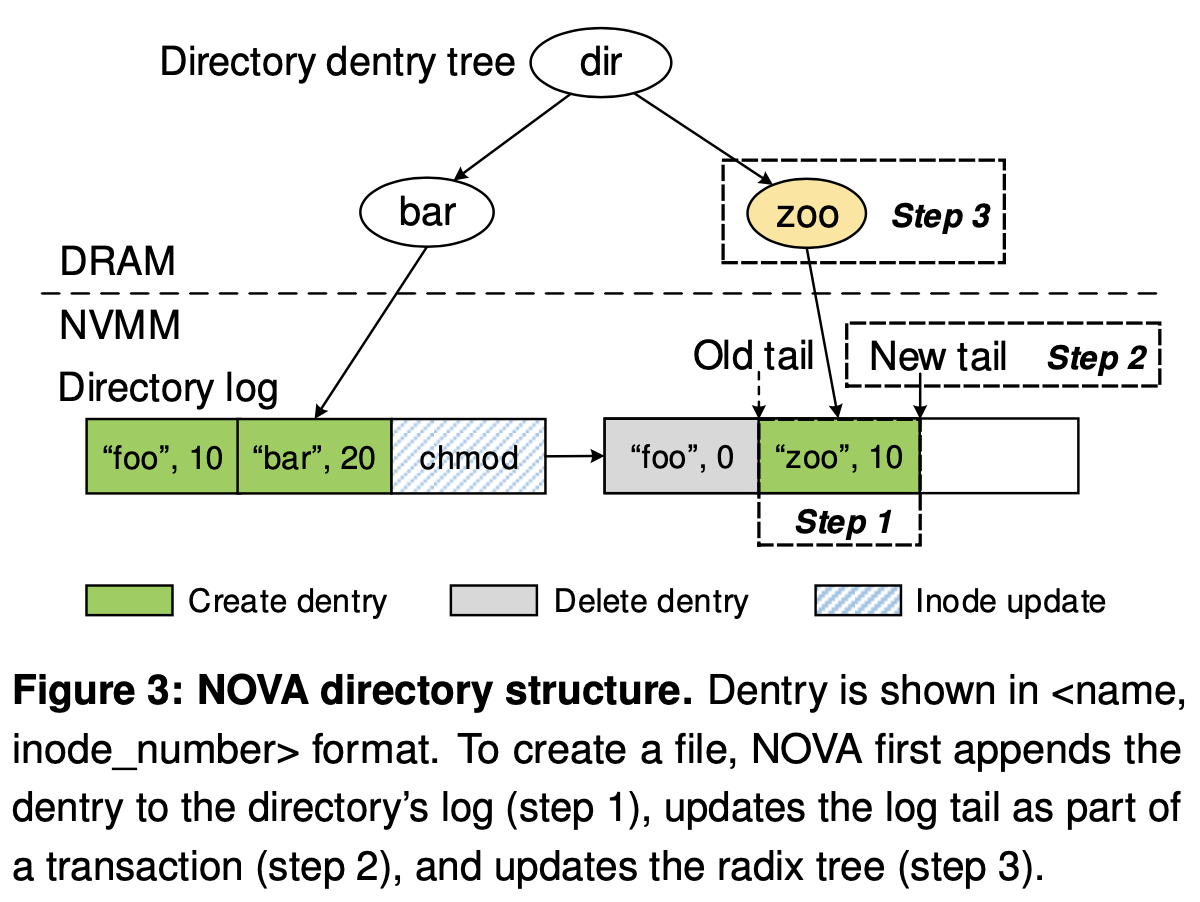
首先，code将条目附加到log中。 然后它用 clwb 刷新受影响的缓存行。 接下来，它发出一个 sfence 和一个 PCOMMIT 指令来强制所有以前的更新到 NVMM 控制器。 第二个 sfence 防止尾部更新在 PCOMMIT 之前发生。 尾部更新的回写和提交，图中没有显示。（**就是一个log的更新过程**）

如果平台不支持新指令，NOVA 使用 movntq，这是一种非临时移动指令，它绕过 CPU 缓存层次结构直接写入 NVMM，并使用 clflush 和 sfence 的组合来强制执行写入顺序。

### 目录操作

NOVA 密切关注目录操作，因为它们对应用程序性能有很大影响 [37,33,64]。 NOVA 包括对所有主要目录操作的优化，包括链接、符号链接和重命名。

NOVA 目录由两部分组成：NVMM 中目录的 inode 日志和 DRAM 中的基数树。 图 3 显示了这些组件之间的关系。 目录的日志包含两种条目：目录条目（dentry）和 inode 更新条目。条目包括子文件/目录的名称、其 inode 编号和时间戳。NOVA 使用时间戳自动更新目录 inode 的 mtime 和 ctime 操作。 NOVA 在创建、删除或重命名该目录下的文件或子目录时会在日志中附加一个 dentry。 删除操作的 dentry 将其 inode 编号设置为零，以将其与 create dentry 区分开来。



NOVA 将 inode 更新条目添加到目录的日志中，以记录目录 inode 的更新（例如，对于 chmod 和 chown）。 这些操作修改了inode的多个字段，inode更新条目提供了原子性。

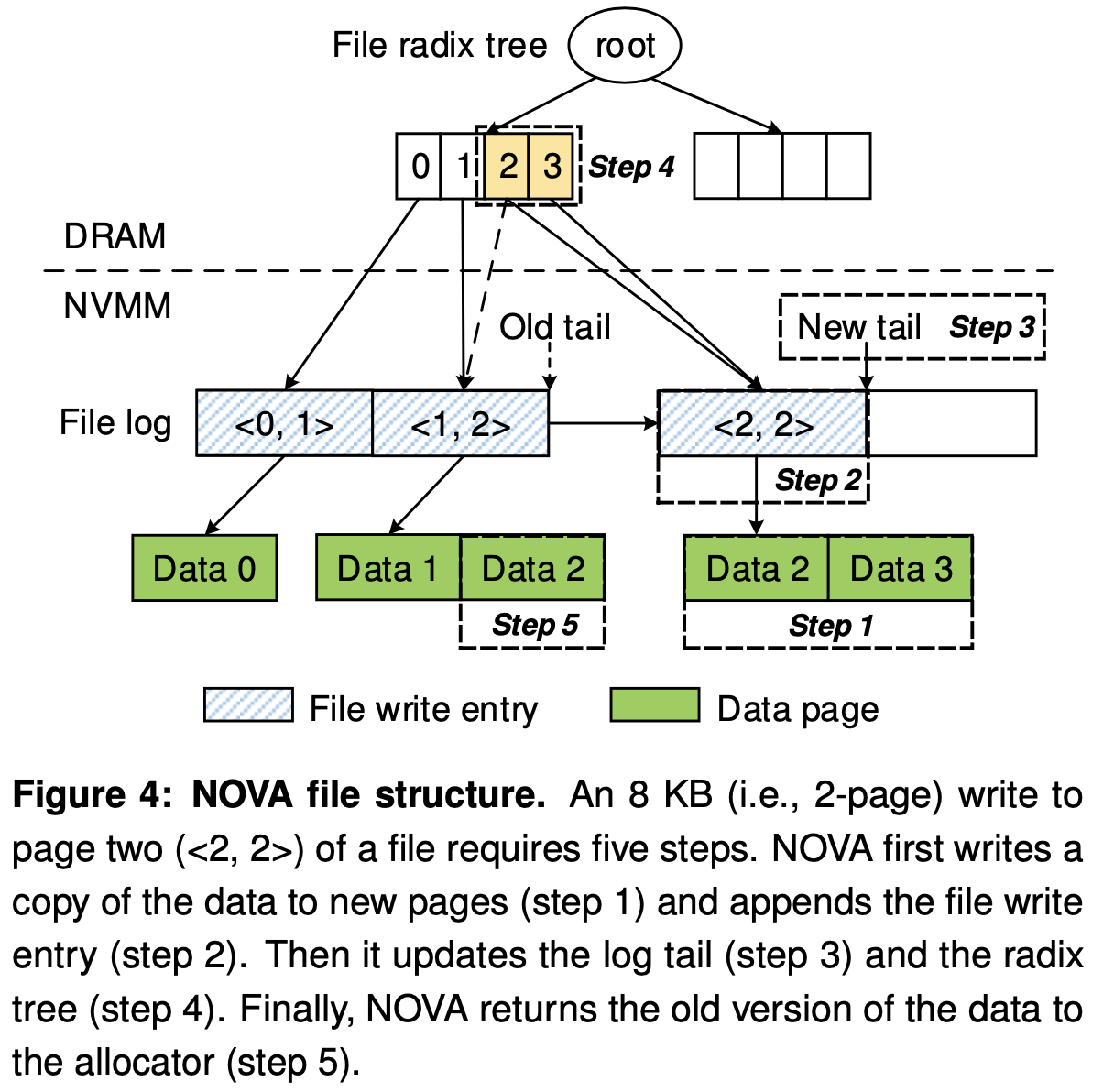
为了加快 dentry 查找速度，NOVA 在 DRAM 中为每个目录 inode 保留一个基数树。 key是dentry名称的hash值，每个叶子节点都指向日志中对应的dentry。 即使对于大型目录，基数树也可以使搜索变得高效。 下面，我们用文件的创建和删除来说明这些原理。

Creating a file

……

### Atomic file operations

NOVA文件结构使用logging来提供元数据和数据的原子性，开销很低，它对文件数据使用写时复制，以减少日志的大小，使垃圾收集简单而有效。图4显示了一个NOVA文件的结构。文件节点的日志记录了元数据的变化，**每个文件在DRAM中都有一个radix tree，通过文件偏移量定位文件中的数据**。



文件 inode 的日志包含两种日志条目：inode 更新条目和文件写入条目，它们描述文件写入操作并指向写入修改的数据页。 文件写入条目还包括时间戳和文件大小，以便写入操作自动更新文件的元数据。 DRAM 基数树将文件偏移映射到文件写入条目。

如果写入量很大，NOVA 可能无法用单个写入条目来描述它。 如果 NOVA 找不到足够大的连续页面集，它会将写入分成多个写入条目并将它们全部附加到日志中以满足请求。 为了保持原子性，NOVA 通过对日志尾指针的一次更新来提交所有条目。

对于读取操作，NOVA 使用 64 位原子写入更新文件 inode 的访问时间，使用文件的基数树定位所需的页面，并将数据从 NVMM 复制到用户缓冲区。

图 4 说明了一个写操作。 符号 <file pgoff, num pages> 表示页偏移和写入影响的页数。 日志中的前两个条目分别描述了 4 KB 和 8 KB（即 1 和 2 页）的两次写入，<0, 1> 和 <1, 2>。 第三个 8 KB 写入 <2, 2> 正在进行中。

为了执行 <2, 2> 写入，NOVA 填充数据页，然后将 <2, 2> 条目附加到文件的 inode 日志中。 然后 NOVA 原子更新日志尾部以提交写入，并更新 DRAM 中的基数树，使偏移量“2”指向新条目。 **保存第 2 页旧内容的 NVMM 页面立即返回到空闲列表。** 在操作期间，每个 inode 锁保护日志和基数树免受并发更新。 当 write 系统调用返回时，所有更新都将持久保存在 NVMM 中。

### Atomic mmap

DAX 文件系统允许应用程序通过加载和存储指令直接访问 NVMM，方法是将物理 NVMM 文件数据页映射到应用程序的地址空间。 这个 DAX-mmap 将 NVMM 的原始性能暴露给应用程序，并且很可能成为未来的关键接口。

虽然 DAX-mmap 绕过了文件系统页面缓存并避免了分页开销，但它给程序员带来了挑战。 DAX-mmap 提供原始 NVMM，因此程序员可用的唯一原子性机制是处理器提供的 64 位写入、围栏和缓存刷新指令。 使用这些原语来构建健壮的非易失性数据结构非常困难 [19,67,34]，并且期望程序员这样做可能会限制直接映射 NVMM 的有用性。

为了解决这个问题，NOVA 提出了一种一致性更强的直接 NVMM 访问模型，称为 atomic-mmap。 当应用程序使用 atomic-mmap 将文件映射到其地址空间时，NOVA 从 NVMM 分配副本页面，将文件数据复制到副本页面，然后将副本映射到地址空间。 当应用程序在副本页面上调用 msync 时，NOVA 将其作为上一节中描述的写入请求进行处理，使用 movntq 操作将数据从副本页面直接复制到数据页面，并以原子方式提交更改。

由于 NOVA 对文件数据使用写时复制并立即回收过时的数据页，因此它不支持 DAX-mmap。 Atomic-mmap 的开销比 DAX-mmap 更高，但提供了更强的一致性保证。 正常的 DRAM mmap 不是原子的，因为操作系统可能会急切地将脏页的子集写回文件系统，从而在系统故障时使文件数据不一致 [45]。 NOVA 可以通过阻止操作系统刷新脏页来支持 DRAM 中的原子 mmap，但我们将此功能留作未来的工作。

### Garbage collection

NOVA 的日志是链表，只包含元数据，使得垃圾回收变得简单高效。 这种结构还使 NOVA 无需不断移动数据以维持连续自由区域的供应。

NOVA 分别处理陈旧数据页和陈旧日志条目的垃圾收集。 NOVA 在写操作期间立即收集陈旧的数据页（参见第 4.4 节）。

清理 inode 日志更加复杂。 如果一条日志条目不是日志中的最后一个条目（因为最后一个条目记录了 inode 的最新 ctime）并且满足以下任一条件，则该日志条目在 NOVA 中是死的：

• 如果文件写入条目未引用有效数据页，则该条目已失效。

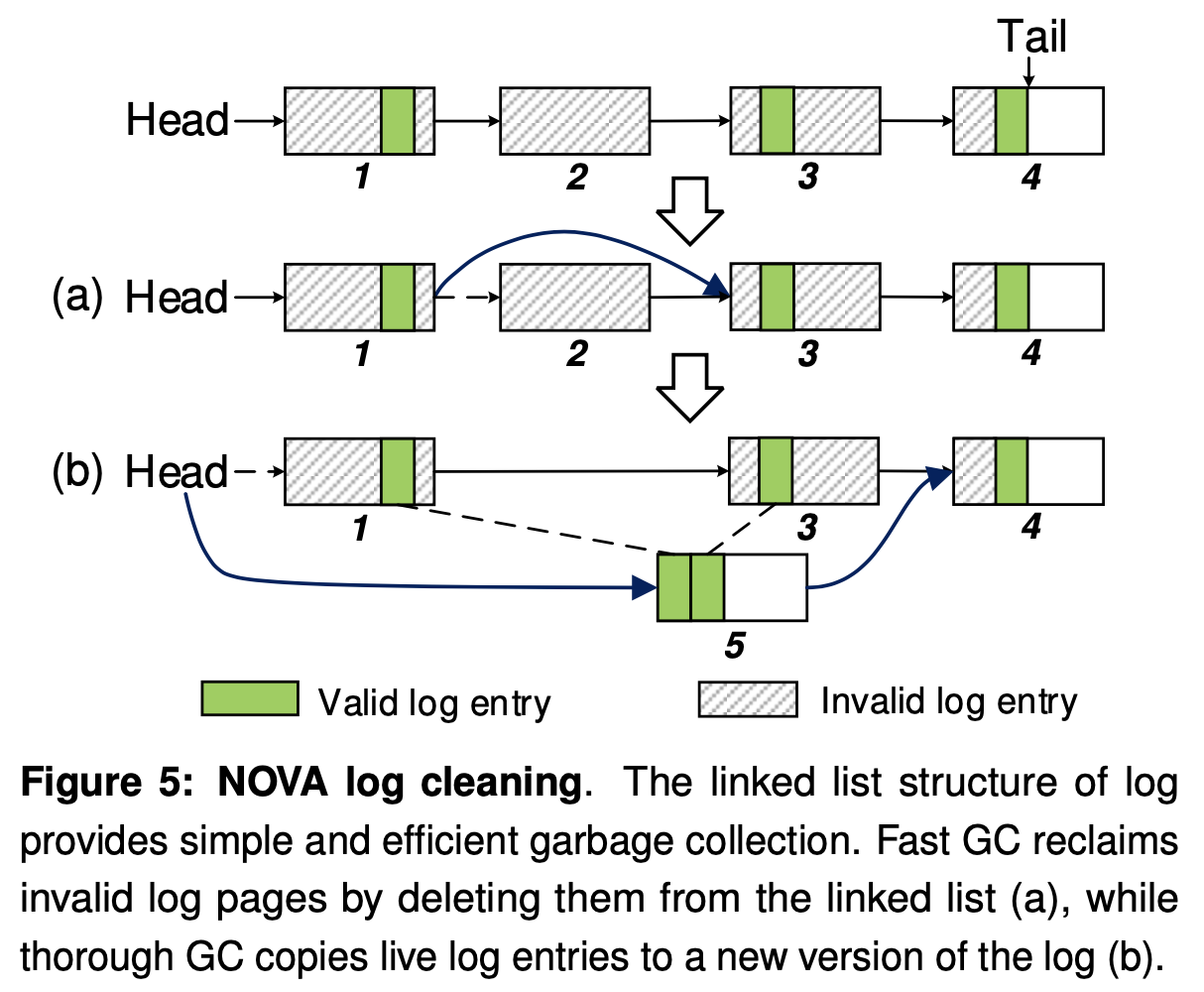
• 如果稍后的 inode 更新修改了同一条元数据，则修改元数据（例如，mode 或 mtime）的 inode 更新已失效。

• 如果dentry 更新被标记为无效，则它已失效。

在某些情况下，NOVA 标记dentries无效。 例如，文件创建会在日志中添加一个 create dentry。 删除文件会添加一个delete dentry，同时也会将创建的目录标记为无效。 （如果 NOVA 垃圾收集器回收了删除目录但留下了创建目录，该文件似乎会重新出现。）

这些规则决定了哪些日志条目是活动的和死的，NOVA 使用两种不同的垃圾回收 (GC) 技术来回收死条目。

Fast GC 快速 GC 强调速度而不是彻底性，它不需要任何复制。 NOVA 在扩展 inode 的日志时使用它来快速回收空间。 如果日志页面中的所有条目都已失效，则快速 GC 通过从日志的链表中删除该页面来回收它。 图 5(a) 显示了快速日志垃圾收集的示例。 最初日志有四页，第 2 页只包含死日志条目。 NOVA 原子地更新第 1 页的下一页指针以指向第 3 页并释放第 2 页。



Thorough GC 在**快速 GC** 日志扫描期间，NOVA 会记录活动日志条目占用的空间。 如果活动条目占日志空间的比例低于 50%，NOVA 在快速 GC 完成后应用**彻底 GC**，将活动条目复制到新的压缩版本的日志中，更新 DRAM 数据结构以指向新日志，然后 原子地用新日志替换旧日志，最后回收旧日志。

图 5(b) 说明了快速 GC 完成后的彻底 GC。 NOVA 分配一个新的日志页面 5，并将页面 1 和 3 中的有效日志条目复制到其中。 然后，NOVA 将第 5 页链接到第 4 页以创建新日志并替换旧日志。 NOVA 不会复制第 4 页中的 live 条目以避免更新日志尾部，以便 NOVA 可以通过更新日志头指针来原子替换旧日志。

### Shutdown and Recovery

当 NOVA 挂载文件系统时，它会重建它需要的 in-DRAM 数据结构。 由于应用程序在文件系统运行时可能只访问部分 inode，NOVA 采用称为延迟重建的策略来减少恢复时间：它推迟重建基数树和 inode，直到系统第一次访问 inode 时间。 该政策加速了恢复过程并减少了 DRAM 消耗。 因此，在重新挂载期间，NOVA 只需要重建 NVMM 空闲页列表。 NOVA 用于恢复空闲列表的算法对于“干净”关闭与系统故障不同。

Recovery after a normal shutdown 在干净卸载时，NOVA 将 NVMM 页面分配器状态存储在恢复 inode 的日志中，并在随后的重新安装期间恢复分配器。 由于在这种情况下 NOVA 不会扫描任何 inode 日志，因此恢复过程非常快：我们的测量表明，NOVA 可以在 1.2 毫秒内重新挂载 50 GB 的文件系统。

Recovery after a failure 如果出现不干净的卸载（例如系统崩溃），NOVA 必须通过扫描 inode 日志来重建 NVMM 分配器信息。 由于两个设计决策，NOVA 日志扫描速度很快。 首先，per-CPU inode 表和 per-inode 日志允许日志恢复中的大量并行性。 其次，由于日志不包含数据页，它们往往很短。 inode 日志中的实时日志条目数大致是文件中的范围数。 因此，NOVA 在恢复过程中只需要扫描一小部分 NVMM。 NOVA 故障恢复包括两个步骤：

首先，NOVA 检查每个日志并回滚任何未提交的事务以将文件系统恢复到一致状态。

其次，NOVA 在每个 CPU 上启动一个恢复线程，并行扫描 inode 表，对 inode 表中的每个有效 inode 执行日志扫描。 NOVA 对目录 inode 和文件 inode 使用不同的恢复机制：对于目录 inode，NOVA 会扫描日志的链表以枚举其占用的页面，但不会检查日志的内容。 对于文件 inode，NOVA 读取日志中的写入条目以枚举数据页。就是看哪些页面是有效的。

在恢复扫描期间，NOVA 构建一个占用页面的位图，并根据结果重建分配器。 此过程完成后，文件系统已准备好接受新请求。

### NVMM Protection

由于内核在 NOVA 挂载期间将 NVMM 映射到其地址空间，因此 NVMM 容易受到内核错误存储的破坏。 为了保护文件系统并防止 NVMM 因杂散写入而永久损坏，NOVA 必须确保它是唯一可以访问 NVMM 的系统软件。

NOVA 使用与 PMFS 相同的保护机制。 挂载时，整个 NVMM 区域被映射为只读。 每当 NOVA 需要写入 NVMM 页面时，它都会通过禁用处理器的写保护控制 (CR0.WP) 打开一个写窗口。 当 CR0.WP 清零时，在 ring 0 上运行的内核软件可以写入内核地址空间中标记为只读的页面。 NVMM 写入完成后，NOVA 重置 CR0.WP 以关闭写入窗口。 CR0.WP 不会跨中断保存，因此 NOVA 在写入窗口期间禁用本地中断。 打开和关闭写窗口不需要修改页表或TLB，因此成本低廉。

## Evaluation