# ReconFS：闪存上的可重构文件系统

——第十二届USENIX文件和存储会议技术（FAST '14）

单位：

清华大学计算机科学与技术系

清华大学信息科学与技术国家实验室

**学号： 2017286190093**

**姓名： 王中霖**

**学院：遥感测绘信息工程国家重点实验室**

**课程： 海量存储**

2017-11-29

在每一章节处，写了关于个人的看法。

简介：

文件系统中的分层命名空间（目录树）对索引文件系统数据是有效的。然而，命名空间元数据的更新模式（如密集回写和分散的小更新）会大大夸大对闪存存储的写入，这会影响存储系统的性能和持久性（即闪存的有限程序/擦除周期）。

在本文中，我们提出了一种可重构的文件系统ReconFS，以减少命名空间元数据回写大小，同时提供分层命名空间访问。 ReconFS将易失性和持久性目录树维护解耦。使用易失性目录树模拟分层命名空间访问，并在系统故障的情况下使用两种机制提供持久性目录树的一致性和持久性。首先，通过在每个页面中嵌入反向索引来确保一致性，消除指针的写入（索引目录树）。其次，通过将分散的小更新压缩并记录到元数据持久性日志来保证持久性，以减少写入大小。分别使用反向索引和日志来重构目录树的结构和内容。实验表明，与ext2相比，ReconFS提供高达46.3％的性能提升和27.1％的写入减少，这是一种具有低元数据开销的文件系统。

**个人感受：**

由于windows系统不开源，促使实验平台选择linux系统，同时也由于windows文件系统的回收机制（闪存会暂留有一些程序进程数据导致内存被占用），运行效率较低，高频度的文件读写，会促使内存最终由于回收机制，使得闪存被占据。

**1、介绍**

近年来，由于其高性能，低功耗和小尺寸，闪存在存储系统中日益普及[11，12，13，19，23，28]。 然而，闪速存储器的编程/擦除（P / E）周期有限，随着P / E周期接近极限，可靠性被削弱，这被称为耐力问题[10,14,17,23]。 最近，通过多层单元（MLC）或三层单元（TLC）技术增加存储容量的更加密集的闪存的趋势使得耐久性问题更加严重[17]。

文件系统设计在过去几十年中缓慢发展，但对存储子系统的I / O行为有明显的影响。 最近的研究提出了重新访问文件系统的命名空间结构，例如针对搜索友好文件系统的灵活索引[33]和表结构化元数据管理，以获得更好的元数据访问性能[31]。 同时，还利用固态硬盘（SSD）的闪存转换层（FTL）的内部存储管理来提高存储管理效率[19,23,25,37]。 但命名空间管理也会影响基于闪存的存储性能和耐久性，特别是在考虑元数据密集型工作负载时。 然而，这尚未得到很好的研究。

由于系统一致性或持久性保证，命名空间元数据集中写入永久存储器[18,20]。 由于闪存的无覆盖属性需要在空闲页面中进行更新，所以频繁写入引入了大的动态更新大小（即，使用的空闲页面的总写入大小）。 更糟的是，单个文件系统操作可能会将更新分散到不同的元数据页面（例如，创建操作同时写入inode和目录条目），并且每个元数据页面的平均更新大小远远小于一个页面大小（例如， ext2中的inode的大小为128字节）。 即使页面中只有一小部分更新，整个页面都需要写入。 闪存存储系统的耐久性以及性能受到名称空间元数据访问的影响，这是由于频繁和分散的小写入模式。

为了解决这些问题，我们提出了一种可重构的文件系统ReconFS，它提供了一个易失性的分层命名空间，并放宽了回写要求。 ReconFS解除了易失性和持久性目录树的维护。只有当主存储器被驱逐或检查点（即，与易失目录树相同的永久性目录树的更新操作）时，元数据页被写回到它们的归属位置。持久目录树的一致性和持久性使用两个新机制得到保证。首先，我们使用嵌入式连接机制在每个页面中嵌入一个反向索引，并跟踪无索引页面。由于命名空间是树形结构，反向索引用于目录树结构重构。其次，我们将每个元数据页面的差异更新记录到元数据持久性日志，并将其压缩成较少的页面，我们将其称为元数据持久性记录机制。这些日志用于重建的目录树内容更新。

幸运的是，可以利用闪存属性来保持两个机制的开销较低。 首先，页面元数据，每个Flash页面旁边的备用空间，用于存储反向索引。 反向索引使用其页面数据进行原子访问，无需额外的开销[10]。 其次，通过将新的分配限制在一个连续的逻辑空间中，在无索引区域中跟踪无索引的页面。 FTL中的地址映射表将写入重定向到不同的物理页面，即使更改了逻辑布局，性能也不受影响。 第三，高随机读取性能使紧凑的日志记录成为可能，因为在恢复期间相应基本页面的读取速度很快。 因此，ReconFS可以以相当低的开销高效地获得性能和耐力益处。

**个人看法如下:**

由于闪存的擦出次数有限制，同时又为了保证日常运行维护的低开销，应该尽力增大闪存的使用周期。同时，基于闪存的存储特性，最小的擦出单元是以块为单位，而最小的写入单元确实以页来写入的，这样导致了一个问题：哪怕数据仅仅只需要更新其中的一小部分：都必须擦出整个块来进行写入。但是这样导致了写入频率大幅增加，耐久性大幅降低。

文章作者提出一种可重构的文件系统设计，以避免维护一个持久目录树的高开销，并使用内存中的易失性目录树模拟分层命名空间访问。我们通过使用索引数据嵌入反向索引来提供命名空间一致性，并消除父节点（在目录树视图中）的指针更新，以减少回写频率。通过记录并将多个元数据页面中的脏部分压缩到元数据持久性日志来提供元数据持久性，紧凑形式减少元数据回写大小。

基于ext2实现ReconFS，并针对不同的文件系统进行评估，包括ext2，ext3，btrfs和f2fs。 与ext2相比，结果显示，与元数据开销较低的文件系统相比，性能提升了46.3％，耐力提高了27.1％。

本文的其余部分安排如下。 第2节给出了闪存和命名空间管理的背景。 第3节描述了ReconFS设计，包括解耦的易失性和持久性目录树维护，嵌入式连接和元数据持久性记录机制以及重构。 我们在第4节介绍实施，并在第5节评估ReconFS。第6节给出了相关工作，第7节得出结论。

**2、背景资料：**

**2.1 闪存基础**

闪存编程在一个方向进行。 闪存单元需要在被覆盖之前被擦除。 读/写单元是闪存页（例如，4KB），并且擦除单元是闪存块（例如，64页）。 在每个闪存页面中，存在用于存储页面的元数据的备用区域，其被称为页面元数据或带外（OOB）区域[10]。 页面元数据用于存储纠错码（ECC）。 并且已经提出将页面元数据公开到NVMe标准中的软件[6]。

闪存转换层（FTL）用于基于闪存的固态驱动器（SSD），以导出块接口[10]。 FTL将软件中的逻辑页码转换为闪存中的物理页码。 地址映射隐藏系统软件的无覆盖属性。 FTL还执行垃圾收集以回收空间和磨损均衡以延长设备的使用寿命。

与硬盘驱动器（HDD）相比，基于闪存的SSD提供更高的带宽和IOPS [10]。 多个芯片通过SSD内的多个通道连接，以提供内部并行性，提供高聚合带宽。 由于机械移动部件的消除，SSD提供高IOPS。 耐力是使基于闪存的SSD与HDD不同的另一个元素[10,14,17,23]。 每个闪存单元都有限制的程序/擦除（P / E）周期。 随着市盈率周期接近极限，每个电池的可靠性急剧下降。 因此，耐用性是基于闪存存储的系统设计中的关键问题。

**注解：**

闪存的读写单位为页，而页的大小一般为4KB或8KB，但我们的操作[系统](http://www.liangchanba.com/misc.php?mod=tag&id=300)读写数据是按HDD的扇区尺寸进行的（512Byte（字节）），更麻烦的是闪存擦除以块作单位，而且未擦除就无法写入，这导致操作系统现在使用的文件系统根本无法管理SSD，需要更换更先进、复杂的文件去解决这个问题，但这样就会加重操作系统的负担。

而为了不加重操作系统的负担，SSD采用[软件](http://www.liangchanba.com/misc.php?mod=tag&id=241)的方式把闪存的操作虚拟成磁盘的独立扇区操作，这就是FTL。因FTL存在于文件系统和物理介质（闪存）之间，操作系统只需跟原来一样操作LBA即可，而LBA到PBA的所有转换工作，就全交由FTL负责

**2.2 层次命名空间**

易失目录树

Volatile

Directory

Tree

ReconFS

Storage 存储

(

Persistent

Directory

Tree, 持久目录树

Persistent

Data

Pages) 持久数据页

Metadata元数据持久记录

Persistence

Log

**Main**

**Memory**

**Persistent**

**Storage**

*Persistence持久性*

*Induced*

*Writeback*

*Consistency一致性*

*Induced*

*Writeback*

*Buffer缓冲*

*Eviction/Checkpoint逐出/检查点*

*Induced诱发*

*Writeback写回*

Figure 1: ReconFS框架工作图

目录树已经在不同的文件系统中使用，以分层次的方式管理数据。但分层命名空间引入了高开销，为目录树提供一致性和持久性。此外，静态元数据组织放大元数据写入大小。命名空间的一致性和持久性。目录和文件以树结构索引，目录树。每个页面都使用指针来索引其子目录树。为了保持目录树的一致性，应该以原子方式更新具有指针和指向页面的页面。使用不同的机制，例如日志记录[4,7,8,34,35]和写时复制（COW）[2,32]来提供原子性，但是引入大量额外的写入。此外，即使在电源故障之后，持久性也需要指针处于持久状态，这就要求这些页面的内部回写。这增加了回写频率，这也对耐力产生了负面影响。

在本文中，我们专注于目录树的一致性，即元数据一致性。

**3.设计**

ReconFS旨在减少对闪存的写入存储同时提供分层名称空间访问。在这一节中，我们先来介绍一下总体设计ReconFS包括分离的易失性和持久性目录树维护和四种类型的元数据写回。 然后我们描述两种机制，嵌入式连通性和元数据持久性日志记录提供持久性的一致性和持久性目录树减少写入，分别。 最后，  
我们讨论ReconFS重建。

**个人观点：**

由于目录树的维护（也就是说元数据与数据必须保持记录上的一致性）；这样就导致如果数据的修改对于元数据来说就必须跟着修改，又因为服务器级别的文件系统一般分为：加载机集群、查询机集群、元数据结点集群以及存储节点集群；用户查询数据要先经过查询机，然后去存储街道集群去取数据。而元数据结点集群则是保存整个系统的所需的元数据，每次更新数据也就意味着元数据结点集群也要更改，使得存储设备次数乘以2。本论文所做将两者之间进行解耦，使得数据维护次数下降，从而增大耐久性。

ReconFS利用反向查找来建立失去元数据的数据，同时把元数据与数据合并从而达到存储次数下降一半，使得耐久性大幅提高。由于两者的解耦，导致元数据会失去数据的信息，这时利用反向查找，便可以使得脏页（数据）重新建立新的链接。

**3.1 ReconFS概览**

ReconFS解耦了volatile的维护永久目录树。 ReconFS模拟一个易失性主内存中的目录树提供分层结构命名空间访问。元数据页面更新为易失目录树不被写回持久目录树。而减少回写可以使闪光灯的性能和续航能力受益存储，一致性和持久性目录树需要在意外的情况下提供系统故障。而不是写回元数据页面直接到他们的家庭地点，ReconFS嵌入倒排索引与命名空间的索引数据一致性或紧凑性，并写回散布以日志结构化的方式进行小型更新。如图1所示，ReconFS由...组成三部分：易失目录树，ReconFS存储和元数据持久性日志。挥发性目录树管理名称空间元数据页面在主内存中提供分层名称空间访问。 ReconFS存储是持久性存储用于ReconFS文件系统。它存储数据和元数据，包括永久目录树，文件系统。元数据持久性日志是一个在持续存储中连续分配空间主要用于元数据持久化。

**3.1.1  解耦易失性和持久性目录树的维护**

由于ReconFS模拟分层命名空间使用volatile目录树访问主内存，提出三个问题。 首先，命名空间元数据内存压力高时，页面需要更换。第二，命名空间的一致性不能保证一旦没有命名空间元数据写入的系统崩溃时光倒流。 第三，更新命名空间元数据意外的系统故障后可能会丢失。

因此，有三种永久存储元祖（表示为Mdisk-管理磁盘）最新的元数据被写回驱逐/退回（表示为  
作为Mup-to-date），没有改变过的元数据被读入内存（表示为untouched）和在内存中复制的过时元数据（表示为作为Mobsolete）。 注意Mobsolete包括两个页面在内存中有脏或干净的副本。 让Mvdt，Mpdt分别是volatile的命名空间元数据和持久化的目录树和Mmemory是挥发性的  
主内存中的命名空间元数据，我们有如下公式。

Mvdt = Mmemory + Mup−to−date + Muntouched,

Mpdt = Mdisk = Mobsolete + Mup−to−date + Muntouched.

由于Mup-to-date和Mmemory是最新的，Mvdt是最新。 相比之下，Mpdt不是最新的，因为ReconFS不会回写仍有副本的元数据在主记忆 挥发性元数据被回写到他们的家庭地点有三种情况：（1）文件系统卸载，（2）无索引区域开关（3.2节）和

（3）对数截断（3.3节）。 我们称之为操作这使得Mpdt = Mvdt成为检查点操作。 当在卸载时检查点的volatile目录树时，它可以通过直接读取持久性来重构  
目录树，供以后的系统启动。

**3.1.2元数据回写元数据回写到持久存储**，包括文件系统存储和元数据持久性日志，可以分为以下四种类型：  
•缓存驱逐引发写回：元数据页由于记忆压力而被驱逐出来回到他们的家庭地点，以便这些页面可以直接读出以备以后访问查找日志。  
•检查点引发回写：元数据页面  
回到他们的家庭位置检查点

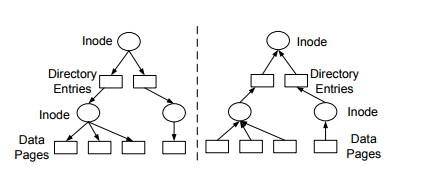


图2：正常索引（左）和反向索引（右）在目录树中  
操作，以减少重建高架。  
•一致性引发的回写：回写指针（用作索引）被消除用索引的数据嵌入一个反向索引的灰分储存，以减少回写  
频率。  
•持续性引发回写：元数据页面由于持久性要求写回来压缩并记录到元数据持久性登录一个紧凑的形式来减少元数据回写大小。

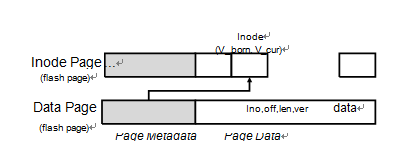
**3.2  嵌入式连接**

命名空间一致性是命名空间元数据需要经常回写到持久性存储的原因之一。在图2左侧所示的目录树的正常索引中，每个链接的指针和指向的页面应该被原子地写回来，以使每个元数据中的命名空间一致性  
操作。这不仅需要更新两个页面，而且还要求日志记录或排序更新以保持一致性。相反，ReconFS使用反向索引提供命名空间一致性，它将反向索引与索引数据嵌入，如图2的右半部分所示。由于指针嵌入了指向的页面，因此可以轻松实现一致性。和日志写的一样，指针更新是一种废弃。以这种方式，嵌入式连接降低元数据回写的频率，并确保元数据的一致性。嵌入式反向索引：在目录树中，有两种链接：从目录条目到索引节点（直接索引节点链接）的链接以及从inode到数据页面（索引节点数据链接）的链接。由于目录条目作为目录的数据页存储在Unix / Linux中，所以从inode到目录条目的链接被分类为inode数据链接。对于一个inode数据链路，反向索引是文件或目录中的索引节点号和数据的位置（即偏移量和长度）。由于反向索引为几个字节，因此存储在页面中每个页面的元数据。对于一个直方图链接，反向索引是文件或目录名称及其inode号。由于名称长度可变，难以进入页面元数据，因此生成由反向索引，索引节目内容和操作类型组成的操作记录，存储在元数据持久性日志中。操作记录中的操作类型设置为用于创建操作的“创建”和用于硬链接操作的“链接”。在重建期间，“链接”类型不会使以前的创建记录无效，而“创建”也不会失效。反转索引还与用于在inode号或目录条目重新使用的情况下识别正确版本的版本号相关联。当inode号码或目录条目被删除后重新使用时，属于已删除的文件或目录的页面可能仍然驻留在具有倒置索引的持久存储器中。在重建期间，这些页面可能被错误地视为有效。为了避免这种歧义，每个目录条目都使用版本号进行扩展，每个inode都使用版本对<Vborn，Vcur>进行扩展，这表示了inode的活跃性。当生成或重用inode时，Vborn是版本号。为一个删除操作，Vborn被设置为增加一个到Vcur。因为当时的所有页面的版本号不大于Vcur，所以删除的inode的所有数据页都被设置为无效。与创建和硬链接操作一样，删除操作将生成删除记录，并将其附加到元数据持久性日志，该日志用于将inode与目录树和所有子页面都无效

未索引区域：索引未写入的页面在系统故障后无法在目录树中访问。 这些页面被称为无索引页面，需要被跟踪进行重建。 ReconFS将逻辑空间划分为多个区域，并将写入限制在每个阶段中的一个区域。 该区域称为无索引区域，并跟踪所有未绑定的页面，一个阶段是无索引区用于分配的时间段。 当区域用尽时，未索引区域切换到另一区域。 在区域切换之前，执行检查点操作将脏页写回其本地位置。 对无索引区的写入限制导致很少的性能损失。 这是因为SSD内的FTL将逻辑地址重新映射到物理地址和数据布局在逻辑空间视图中对物理空间视图的数据布局至关重要，对系统性能影响不大。

命名空间连接外，位图回写是频繁元数据持久化的另一个来源。位图更新经常被写回来保持空间分配一致。 ReconFS仅将易失性位图保存在主内存中，该内存用于逻辑空间分配，并且不保持持久位图更新。 一旦系统崩溃，重建位图。 由于仅在无索引区域中执行新分配，因此使用页面的有效和无效状态重建无索引区域中的位图。 其他区域中的位图仅在页面被删除时更新，并且可以使用元数据持久性日志中的删除记录来重构这些更新。

**3.3元数据持久记录元数据持久性导致频繁的元数据回写。**写回的分散的小更新模式放大了以页为单位写回的元数据写入。而不是使用静态压缩（如第2节所述），ReconFS会动态地压缩元数据更新并将其写入元数据持久性日志。虽然静态压缩需要将元数据更新写回其本地位置，但动态压缩能够以紧凑的形式集中小型更新。动态压缩仅写入脏部分而不是整个页面，以减少写入大小。在元数据持久性日志记录中，需要持久性时触发写回，例如显式同步或pd fl ush守护程序的唤醒。元数据持久性记录机制跟踪主内存中每个元数据页面的脏部分，并将这些部分压缩到日志中：  
•内存脏标签：对于每个元数据操作，元数据页面都会在主内存中进行更新。 ReconFS记录每个更新的元数据页面中脏部分的位置元数据（即，偏移量和长度）。位置元数据附加到元数据页面的缓冲区头以跟踪每个页面的脏部分。 •回写压缩：在回写期间，ReconFS会传送多个元数据页面，并将其脏部分附加到日志页面。每个脏部分具有附加在每个日志页的头部中的位置元数据（即，基页地址，页面中的偏移量和长度）。当元数据持久性日志运行空间不足时，需要进行日志截断。 ReconFS将日志中的小型更新与基础元数据页面进行合并，而不是执行检查点操作，将所有脏元数据页面回写到其本地位置。为了减轻回写成本，使用回写守护程序以异步方式执行检查点操作，并且当日志空间低于预定义阈值时，守护程序启动。因此，日志被截断，没有昂贵的合并操作。多页更新原子性。操作记录需要多页更新的原子性大于一页（例如，具有4KB文件名称的文件创建操作）。为了提供元数据操作的一致性，这些页面需要原子更新。在灰存储中保证单页更新原子性，因为闪存存储器的无覆盖属性需要在新的页面中更新页面  
 在FTL映射表中紧随原子映射条目更新。多页更新原子性简单地通过每个页面中的flag位来实现。由于元数据操作记录被写入连续分配的日志页中，原子性是通过标记这些页面的开始和结束来实现的。最后一页用fl ag'1标记，其他标签为'0'。该位存储在每个日志页面的头部。当日志页面被写回时，它被设置，并且不需要额外的写入。在恢复期间，flag位'1'用于确定原子性。两个1之间的页面属于完整操作，而没有结尾'1'的日志尾部页面属于不完整的操作。以这种方式，实现了多页面更新的原子性。

3.4 **ReconFS重建**：在正常关机期间，易失性目录树将检查点写入永久存储器中的持久性目录树，只需将其读入主存储器中即可重构易失目录树，以便下次启动系统。但是一旦系统崩溃，ReconFS需要使用嵌入式连接和元数据持久性记录机制记录的元数据重构易失性目录树。由于持久化目录树是检查点  
当无索引区域被切换或日志被截断时，在非索引区域中执行所有页面分配，并且所有元数据更改已记录到​​持久性元数据日志中。因此，ReconFS只需要通过扫描未索引的区域和元数据持久性日志来更新目录树。 ReconFS重建包括：  
**1.文件/目录重建**：无索引区域中的每个页面都使用其反向索引连接到其索引节点。然后，每个页面检查其反向索引中的版本号  
在其索引节点中的<Vborn，Vcur>如果这匹配，则页面被索引到文件或目录。否则，页面被废弃，因为页面已被无效。此后，所有页面（包括文件数据页面和目录条目页面）将被索引到其索引节点。  
**2.目录树连接重建**：元数据持久性日志被扫描以搜索dirent-inode链接。这些链接用于将这些inode连接到目录树，以便

**图3：Inode数据链路的反向索引更新目录树结构**。

**目录树内容更新**：元数据持久性日志中的日志记录用于更新目录树中的元数据页面，因此目录树的内容更新到最新。

**4.位图重建**：通过检查每个页面的有效状态可以重新设置无索引区域中的位图，可以使用版本号识别位图。 其他区域中的位图不会被更改，除了已删除的页面。 使用删除或截断日志记录，更新位图。  
在重建之后，持久性目录树中的那些过时的元数据页被更新为最新，并且最近分配的页被索引到目录树中。 重建易失性目录树以提供分层命名空间访问。

个人拙见：

位图：个人按照图形位图理解：由称作像素（图片元素）的单个点组成的。这里理解为矢量图，也就是说每一个元数据以矢量图的形式指向数据

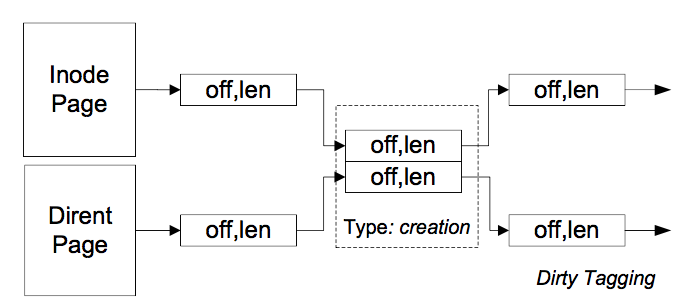
**4实施**

**个人看法：**

文件重构的过程，便是在文件完全更新完一个block块的时候，或者更大。重新将元数据通过inode号来找到他原来的指向页。版本号则是又来后查看是否为脏页，如果是的话，则更新当前的元数据（到元数据存储块）。这与海量结构化数据存储检索系统中并发查找子任务并发执行调度相似。

ReconFS是基于Linux内核3.10.11中的ext2文件系统实现的。 ReconFS共享ext2的磁盘和内存数据结构，但修改命名空间元数据回写流。在易失性目录目录树中，ReconFS为每个元数据缓冲区使用两个脏流：持久性脏和检查点脏。持久性脏标签用于回写元数据持久性日志。 Checkpoint dirty被标记为持久性目录树的回写。当缓冲区更新时，它们都被设置。只有当元数据页被写入元数据持久性日志以进行元数据持久性时，持久性清除才被清除。只有当元数据被写回其原始位置时，检查点脏污才被清除。 ReconFS使用双重脏器从元数据组织（持久目录树）中分离元数据持久性（元数据持久性日志）。在嵌入式连接性中，inodeata和dirent-inode链接的反向索引以不同的方式存储。 inode数据链接存储在  
每个页面的页面元数据。它的形式（ino，o f f，len，ver），其中ino是索引节点号，o f f和len是偏移量和有效数据长度文件或目录，ver是版本inode的数量。

在嵌入式连接中，inode数据的反向索引和dirent -inode链接以不同的方式存储。inode- data链接的反向索引存储在每个flash页面的页面元数据中。它的形式是(ino, off, len, ver)，其中in是inode号，off和len分别是文件或目录中的偏移量和有效数据长度，而ver则是inode的版本号。一个dirent-inode链接的反向索引存储为日志记录，记录类型在元数据持久性日志中设置为“创建”。



**图4 主要内存中的脏标签**

日志记录包含目录条目和inode内容，并为每个内容保留一个(off、len、lba、ver)元数据项。lba是基本元数据页的逻辑块地址。日志记录充当inode的反向索引，用于将其重新连接到目录树。在ext2中，通过集群多个块组来设置ReconFS中的未索引区域。ReconFS限制了这些块组的新分配，从而使这些块组成为未索引区域。这些块组的地址保存在文件系统超级块中，并在每个区域交换机上进行持久化。

在元数据持久性日志中，ReconFS使用链表标记每个元数据页面的脏部分，如图4所示。链表中的每个节点都是一对(f,len)来表示哪个部分是脏的。在每次插入之前，都要检查列表来合并重叠的脏部分。持久日志记录还将每个元数据页面的类型的类型、版本号ver和逻辑块地址lba与链接列表对关联起来，然后是脏内容。在当前的实现中，ReconFS将元数据持久性日志写为根文件系统中的一个文件。为文件系统卸载、未索引区域开关或日志截断执行检查点。当发出卸载命令时，将执行文件系统卸载检查点，而另外两个的检查点则是在未索引区域的空闲空间或元数据持久性日志下降低于5%时触发的。

ReconFS的重建分三个阶段进行:

1 .扫描阶段:将未索引区域中的所有flash页面的页面元数据和元数据持久性日志的日志记录读入内存。在此之后，将收集出现在其中任何一个的元数据页的所有地址。然后，所有这些元数据页都被读入内存。

2 .区域处理阶段:在未索引区域中，每个flash页面都使用其页面元数据中的反向索引连接到它的inode。文件和目录的结构被重建，但它们可能有过时的页面。

**表1**

|  |  |
| --- | --- |
| ext2 | 传统的文件系统，没有日志记录 |
| ext3 | 传统的日志文件系统(jour - naled版本的ext2) |
| btrfs[2] | 最近的一个即写即拷(COW)文件系统 |
| f2fs[12] | 最近的一个日志结构的优化了flash的文件系统 |

3 .项目日志处理阶段:每个日志记录用于将一个文件或目录连接到目录树，或者更新元数据页内容。对于创建或硬链接日志记录，为inode更新目录条目。对于删除或截断日志记录，要读取和更新相应的位图。其他日志记录用于更新页面内容。最后，检查页面和inode的版本，以丢弃过时的页面、文件和目录。

**5 评价**

我们评估了ReconFS相对以前文件系统的性能和持久性，包括ext2、ext3、btrfs和F2FS，并打算回答以下四个问题:

1．工作在性能和持久性方面，ReconFS如何与以前的文件系统进行比较?

2．什么样的操作能从ReconFS获得更多受益? 从嵌入式连接和元数据持久化日志能获得什么益处?

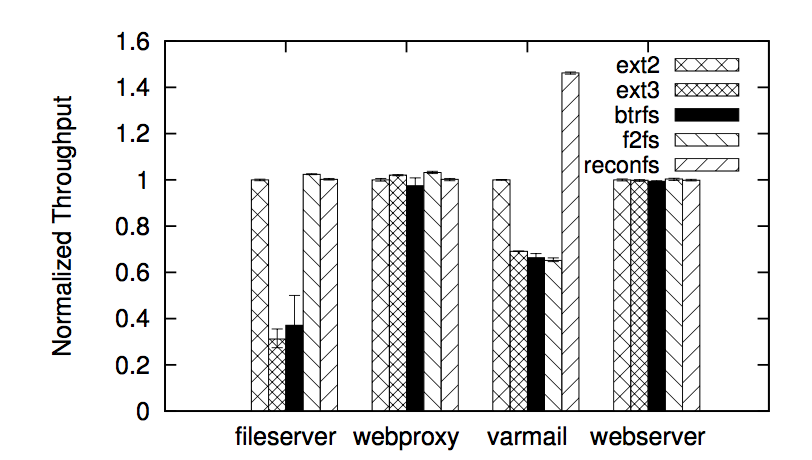
3 . 项目内存大小变化的影响是什么?

4．在ReconFS中，检查点和重建的开销是多少?

在这一节中，在回答上述问题之前我们首先描述实验的设置。

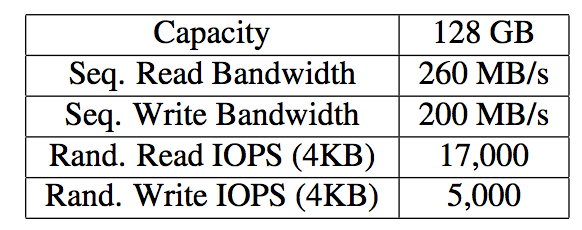
**5.1 实验设置**

我们在Linux内核3.10.11中实现了ReconFS，并评估了ReconFS与表1中列出的文件系统的性能和持久性。我们使用来自filebench基准的4个工作负载。它们模拟不同类型的服务器。每个工作负载的操作和读写比率说明如下: fileserver仿真一个文件服务器，它执行一系列的创建、删除、附加、读、写和属性操作。读写比率为1:2。webproxy模拟了一个web代理服务器，它执行混合了create -write- close、open -read- close和delete操作，以及日志追加。读写比率是5:1。



**图5 系统性能对比**

**表2 SSD规格说明**



varmail模拟一个邮件服务器，它执行一组create-append-sync，read-append-sync，read和delete操作。 读写比为1：1。webserver模拟一个Web服务器，它执行开放读取操作，以及日志追加。 读写比为10：1。

实验使用Linux内核3.10.11在Fedora 10上进行，计算机配备4核2.50GHz处理器和12GB内存。 我们评估128GB SSD上的所有文件系统，其规格如表2所示。所有文件系统都装有默认选项。

5.2 系统对比

5.2.1全部对比

我们评估所有文件系统的性能通过测量基准报告的吞吐量，以及通过测量存储器的写入大小来获得耐久性。 使用blktrace工具从块级别跟踪中收集对存储的写入大小。图5显示了归一化为ext2的吞吐量以评估性能。如图所示，对于所有评估的工作负载，ReconFS是所有文件系统中最好的系统之一，并且对于varmail（元数据密集型工作负载），性能提升高达ext2的46.3％。对于阅读密集的工作负载，如webproxy和webserver，所有评估的文件系统都没有显示出很大的差异。但是对于写入密集型工作负载，如文件服务器和varmail，它们显示了不同的性能。Ext2显示的性能比其他文件系统(不包括ReconFS)要高。ext3和btrfs都提供了不同机制的名称空间一致性。例如，等待数据在写入元数据之前到达持久性存储，但与ext2相比性能较差。F2FS，具有针对闪存优化的数据布局的文件系统显示与ext2相当的性能，但在varmail工作负载中性能较差，它是元数据密集型且有频繁的fsync。相比之下，ReconFS在所有评估的工作负载中实现了ext2的性能，几乎是所有以前的文件系统的最佳性能，甚至比varmail工作负载中的ext2更好。此外，ReconFS提供命名空间与嵌入式连接的一致性，而ext2则不具备。

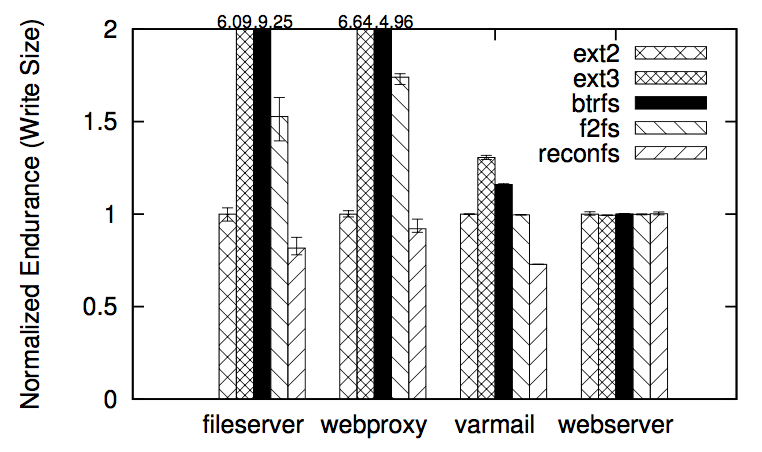


图6:系统持久性的比较

图6显示了标准化到ext2的写入大小，以评估耐久性。 从图中可以看出，ReconFS有效地减少了元数据的写入大小，并将写入大小与ext2相比减少了高达27.1％。和性能一样，ext2的持久性是除ReconFS之外的所有文件系统中最好的。在此期间，ext3、btrfs和F2FS使用日志或复制-写入来提供一致性，这引入了额外的写入。例如，btrfs的写大小为fileserver工作负载中ext2的9倍。ReconFS提供了使用嵌入式连接的名称空间一致性，而无需添加额外的写入，并通过压缩元数据回写来进一步降低写的大小。如图所示，ReconFS显示了与ext2相比，fileserver、webproxy和varmail工作负载的写入大小降低了18.4%、7.9%和27.1%。

**5.2.2 性能**

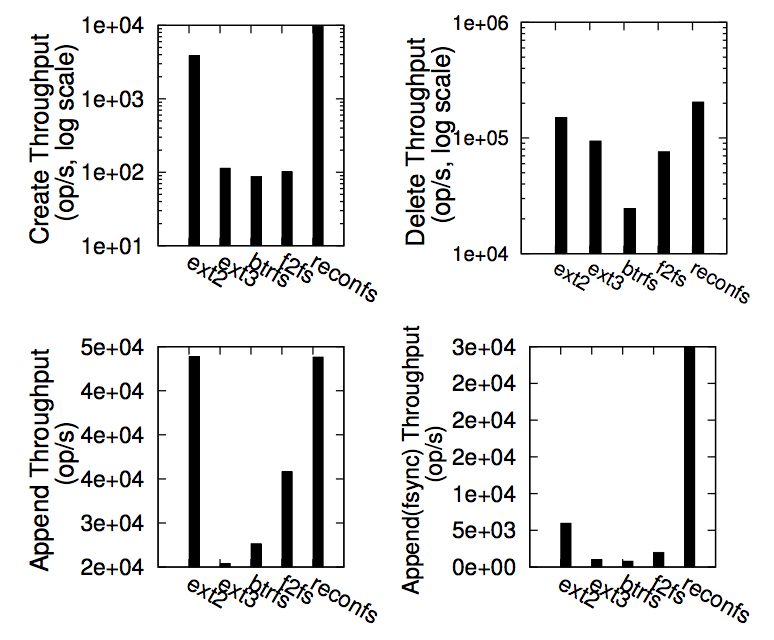


图7:操作的性能评估(文件创建、删除、附加和附加fsync)

为了了解ReconFS的性能影响，我们评估了四个不同的操作，这些操作必须更新索引节点页面和/或目录条目页面。四个操作是文件创建、删除、附加和附加fsyncs。他们评估使用微基准测试。文件创建和删除基准将创建或删除100个目录中的100K文件。在每次创建后执行fsync。append基准将4KB的页面追加到一个文件中，并为每1000个（每4 mb一个fsync）和10个（one fsync per 40KB ）append操作分别插入一个fsync用于评估append并附加fsyncs。图7显示了这四个操作的吞吐量。ReconFS显示了在文件创建中增加了显著的吞吐量，并附加了fsyncs。在ReconFS中文件创建吞吐量使ext2的吞吐量增加一倍。图7显示了这四个操作的吞吐量。ReconFS显示了在文件创建中增加了显著的吞吐量，并附加了fsyncs。在ReconFS中文件创建吞吐量使ext2的吞吐量增加一倍。这是因为只有一个日志页面附加到元数据存储日志,而在ext2多个页面需要写回。其他文件系统的文件创建性能甚至更差，这是由于一致性的原因。在ReconFS中的文件删除操作也比其他操作显示出更好的性能。在ReconFS中，文件附加吞吐量几乎等于在ext2中对每1000个append 0.8操作中的一个fsync的append操作。但是，随着fsync频率从1 / 1000增加到1 / 10，文件追加(与fsyncs)的吞吐量显著下降，其他日志或日志结构文件系统也是如此。相比之下，在ReconFS中，file append(与fsyncs)吞吐量只下降到以前吞吐量的一半。当fsync频率为1 / 10时，ReconFS的文件吞吐量比ext2好5倍，比其他文件系统好几个数量级。

5.2.3 持久性

为了进一步研究ReconFS的持久性优势，我们测量了ext2、没有日志压缩的ReconFS(表示为ReconFS - ec)和ReconFS的写入大小。

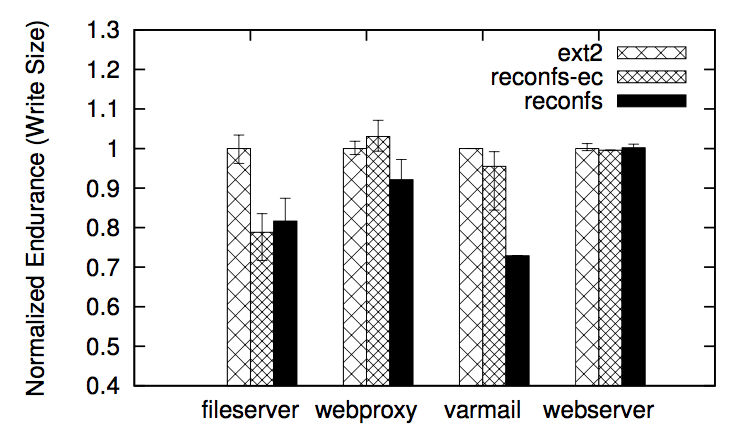


图8:嵌入式连接和元数据持久性日志的持久性评估

图8显示了三个文件系统的写大小。我们比较了ext2和reconfs - ec的写大小，以评估嵌入式连接的好处，因为reconfs - ec实现了嵌入式连接，但是没有日志压缩。从图中可以看出，文件服务器工作负载显示了从ext2到reconfs - ec的写入大小的显著下降。好处主要来自于在文件服务器工作负载中创建和添加的密集文件，否则需要为名称空间连接更新索引指针。ReconFS中的嵌入式连接消除了这些索引指针的更新。我们还比较了ReconFS - ec和ReconFS的写大小，以评估日志压缩在元数据持久性日志中的好处。如图中所示，ReconFS显示了varmail工作负载的大幅减少。这是因为频繁的fsyncs减少了缓冲的影响，换句话说，在写回数据时，对元数据页的更新是很小的。因此，日志压缩比其他工作负载获得更多的改进。

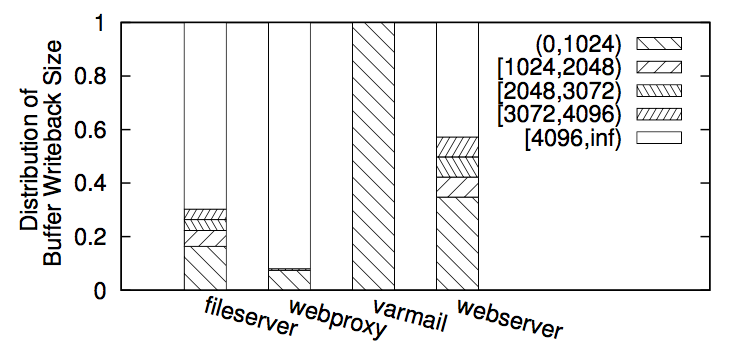
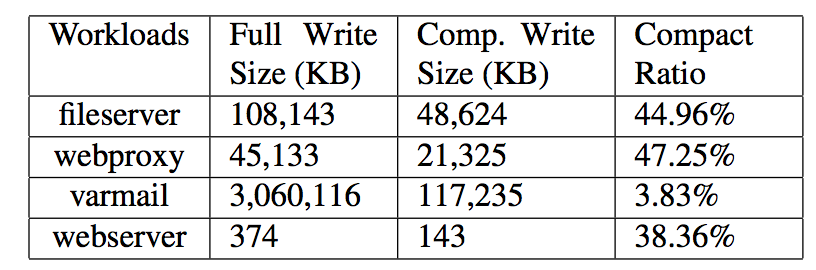


图9:缓冲页面回写大小的分布

图9还显示了缓冲页的回写大小的分布，这是每个页面中脏部分的大小。如图所示，由于频繁的fsyncs，varmail工作负载的元数据回写中每个页面的99.9％以上的脏数据都小于1KB，而其他数据的脏数据的分数从7.3％到34.7％不等于1KB。此外，我们通过将整个页面更新大小除以紧凑的写入大小来计算压缩比。如表3所示。varmail工作负载的压缩比达到3.83%。

表3 全写入和压缩写入的比较

**5.3内存大小的影响**

为了研究内存大小的影响，我们将内存大小设置为1，2，3，7和12 GB，并测量所有评估文件系统的性能和耐久性。 我们以每秒操作为单位（ops / s）测量性能，以每个操作字节（字节/ op）为单位，通过将总写入大小与操作次数相除来测量耐久性。 Webproxy和Webserver工作负载的结果由于空间限制而不显示，因为它们是读密集型工作负载，并且各文件系统之间几乎没有区别。

图10（a）显示了不同内存大小下所有文件系统的文件服务器工作负载的吞吐量。如图所示，当内存大小变大时，ReconFS会获得更多的收益，在这种情况下，数据页回写频率低，并且元数据页面的回写具有更大的影响。当内存小，内存压力高时，数据写入的影响占主导地位。 ReconFS具有比F2FS更差的性能，它具有优化的数据布局。当内存大小增加时，元数据写入的影响会增加。当内存大小从7GB增加到12GB时，ext3和btrfs几乎没有改进。相比之下，ReconFS和ext2对于其较低的元数据开销而言获得了显着的改进，并且接近了F2FS的性能。

图10（c）显示了由文件服务器每个运算的字节数度量的耐久性。在图中，ReconFS具有与其他文件系统相当或较小的写入大小。

图10（b）显示了varmail工作负载的吞吐量。性能在不同的内存规模下是稳定的，ReconFS实现了最佳性能。这是因为varmail工作负载是元数据密集型工作负载，并且频繁进行fsync操作。

图10（d）显示了varmail工作负载的耐久性。 ReconFS在所有文件系统中达到最佳。

**5.4 重建开销**

我们测量卸载时间来评估检查点的开销，该检查点写回所有脏元数据，使持久化目录树等同于易失性目录树，包括重建时间。

卸载时间：我们使用time命令来测量卸载操作的时间，并使用time命令报告的已用时间。

图11显示了当每个基准测试完成时立即执行卸载的卸载时间。读密集型工作负载，webproxy和webserver对于所有文件系统的卸载时间少于1秒。 但是写密集型工作负载，不同的文件系统有着各种不同的时间。ext2中的卸载时间为46秒，而ReconFS的卸载时间为58.所有卸载时间值都小于一分钟，它们包括用于数据和元数据回写的时间。 图12显示了在每个基准测试完成90秒后执行卸载的卸载时间。 所有都不到一秒钟，ReconFS与其他系统没有显著的区别。

重建时间：重建时间有两个主要部分：扫描时间和处理时间。 扫描时间包括无索引区扫描和日志扫描的时间。扫描是顺序读取，其性能受设备带宽限制。 处理时间是除了恢复逻辑处理时间之外，用于读取要更新的目录树中的基本元数据页面的时间。

如图13所示，SSD上的8GB区域的扫描时间为48秒，处理时间约为1秒。 预计使用PCIe SSD可以减少扫描时间。 例如，具有3GB / s的PCIe SSD的32GB区域的扫描时间约为十秒。 因此，对于高读取带宽和IOPS，ReconFS的重建可以在几十秒内完成。

**6相关工作**

文件系统命名空间：文件系统命名空间方面的研究一直集中于有效性和有效的命名空间元数据管理。 关系数据库或基于表的技术已被用于管理命名空间元数据，为了一致性或性能。 反向文件系统[26]使用PostGRES数据库系统管理命名空间元数据，为元数据提供事务保护和崩溃恢复。 TableFS [31]在LevelDB [5]中存储命名空间元数据，通过利用在LevelDB中实现的日志结构合并树（LSM-tree）[27]来提高元数据访问性能。

命名空间的层次结构还曾有过以灵活的方式实现的讨论，以便提供语义访问。语义文件系统[16]删除树结构的命名空间，并使用属性访问文件和目录。hFAD [33]提出了一种类似的方法，它将搜索友好的文件系统优先于分层文件系统。

Pilot[30]提出了一种甚至是侵略性的方法，并消除了文件系统中的所有索引，其中文件通过64位通用代理（UID）访问。 Pilot不提供树状结构的文件访问。相比之下，ReconFS仅删除永久存储的索引以降低元数据成本，并且使用volatile目录树模拟树结构化文件访问。

反向指针和倒置指数：各种用途的存储系统中使用了Backpointers。 BackLog [24]在数据块中使用backpointer来减少数据块由于高级文件系统功能（如快照，克隆）的移动而导致的指针更新。 NoFS [15]使用backpointer进行每次读取的一致性检查，以提供一致性。 他们都使用backpointer作为辅助来增强新功能，但是ReconFS使用反向指针（反向索引）作为唯一的索引（没有前向指针）。

在基于闪存的SSD中，反向指针（例如，逻辑页面地址）存储在每个闪存页面的页面元数据中，是和页面数据一起被自动获取的，以恢复FTL映射表[10]。 在每个设备启动时，扫描所有页面，并使用backpointer恢复FTL映射表。 OFSS [23]以类似的方式使用在页面元数据中的backpointer。 OFSS使用基于对象的FTL，以及每个页面中的backpointer记录对象的信息，用于延迟对象索引的持久性。 ReconFS将闪存中的backpointer的使用扩展到文件系统命名空间管理。代替维护索引（前向指针），ReconFS只将索引数据的反向索引（反向指针）嵌入，而一旦系统出现意外，反向索引就被用于重构。

文件系统日志。 文件系统以两种不同的方式使用日志。 一个是日志记录，它在更新元数据和/或数据到其本地位置之前，先将他们更新到日志区域，这被广泛地用于现代文件系统中以提供文件系统一致性[4,7,8,34,35]。 结构化日志文件系统以其他方式使用日志[32]。 日志结构化文件系统以记录方式编写所有数据和元数据，且用随机写入顺序，以获得更好的性能。

ReconFS采用日志记录机制提供元数据持久性。与需要跟踪检查点的有效和无效页面以及垃圾清理的日志文件系统或日志结构化文件系统不同，在对所有易失性元数据进行回写后， ReconFS中的元数据永久性日志会被简单丢弃 ReconFS还可以实现紧凑的日志记录，因为在重建期间可以快速读取基础元数据页面，这得益于闪存存储的高超随机读取性能

基于闪存存储的文件系统。除了嵌入式闪存文件系统[9,36]，研究人员还提出了一种新的闪存储存通用系统。 DFS [19]是一种通过利用FusionIO ioDrive提供的功能（例如块分配，原子更新）来直接管理闪存内存的文件系统。匿名[37]也在文件系统中消除了空间分配功能，利用FTL空间管理进行空间分配。 OFSS [23]提出使用基于对象的FTL直接管理闪存内存，在这个方法中还可以利用闪存内存特效来对对象索引，自由空间管理和数据布局进行优化。 F2FS [12]是一个有前途的结构日志文件系统，专为闪存存储而设计。它优化了闪存内存中的数据布局，例如热/冷数据分组。但是这些文件系统几乎没有注意到命名空间元数据的高开销，这些元数据经常被写回并以散落的小写入量模式写入。 ReconFS第一个解决闪存存储的命名空间元数据问题。

**7结论**

命名空间元数据的性质，例如密集回写和分散的小型更新，使得在性能和耐久性方面，命名空间管理的开销高于闪存存储。 ReconFS删除持久性目录树的维护，并使用易失性目录树模拟分层访问。使用嵌入式连接和元数据永久性性记录机制，系统意外故障后，ReconFS可重构。嵌入式连接利用索引数据嵌入还原索引，来实现目录树结构重建。在指针更新时删除父页面（目录树中）中的更新，简化了一致性维护，并减少了回写频率。元数据永久性日志记录为元数据页面提供持久性，并且记录的元数据用于目录树内容重建。由于只有元数据页面的脏部分在日志中被记录和压缩，所以写回大小减少。由于带宽和闪存存储的IOPS很高，重建速度很快。通过新的命名空间管理，ReconFS可以改善基于闪存的存储系统的性能和持久性，而不会影响一致性或持久性。