|  |  |
| --- | --- |
| 学号 | 2017202110079 |
| 密级 |  |

**海量信息存储课程论文**

|  |
| --- |
| **WAFL高效空闲空间回收算法与数据结构** |

|  |  |
| --- | --- |
| 院（系）名称： | 计算机学院 |
| 专业名称： | 信息安全 |
| 学生姓名： | 李想 |
| 指导教师： | 何水兵 教授 |

二〇一七年十二月

摘要

任意位置写入文件布局WAFL是一种事务性文件系统，它使用写入时复制机制COW来支持快速写入性能和高效的快照创建。但是，写时复制增加了文件系统对快速查找空闲块的需求，否则可能会因为任务消耗CPU和其他资源而妨碍传入写入的分配，所以在本文中效率也很重要。我们描述了为了对整个存储设备的性能影响最小，WAFL回收空间的算法和数据结构在数十年时间里的发展情况。

关键词：任意位置写入文件布局（WAFL）；写时复制（copy on write）；空闲空间；活动位图（activemap）；一致性树。

目 录

[1 介绍 1](#_Toc500320288)

[2 背景与动机 2](#_Toc500320289)

[2.1 块获得释放 2](#_Toc500320290)

[2.2 空闲空间的延迟回收 3](#_Toc500320291)

[2.3 WAFL中释放块的成本 3](#_Toc500320292)

[3 空闲空间：基础 5](#_Toc500320293)

[3.1 释放块的重用 5](#_Toc500320294)

[3.2 循环依赖 5](#_Toc500320295)

[3.3 处理文件删除 6](#_Toc500320296)

[4 批量更新元数据 7](#_Toc500320297)

[4.1 树日志 7](#_Toc500320298)

[4.2 批量空闲日志 8](#_Toc500320299)

[4.3 登录参考计数文件系统 8](#_Toc500320300)

[4.4 评估 9](#_Toc500320301)

[5 快照和可用空间 10](#_Toc500320302)

[5.1 汇总图 10](#_Toc500320303)

[5.2 快照创建 11](#_Toc500320304)

[5.3 另一个循环依赖 11](#_Toc500320305)

[5.4 快照删除 11](#_Toc500320306)

[6 FlexVol虚拟化层 13](#_Toc500320307)

[6.1 评估 13](#_Toc500320308)

[7 总结 15](#_Toc500320309)

[参考文献 16](#_Toc500320310)

# 介绍

文件系统控制数据和元数据的存储和检索，它通常将其持久存储分成可寻址的块，通过分配和释放这些块来处理客户端请求，所以高效的空闲空间管理是文件系统性能的关键要素。企业级文件系统要求读写性能始终如一的高性能。WAFL是基于事务的文件系统，采用写时复制（COW）机制来实现快速写入和高效快照创建的性能。

与其他现代文件系统（如FFS[16]，XFS[22]，ZFS[17]，ext3，ext4[15]和brtfs[19]）一样，WAFL跟踪已分配空间和空闲空间的原因有两个：使文件系统能够查找和分配空闲块，以适应新的写入；向系统管理员报告空间使用情况，以指导采购和供应决策。与ext3和ext4写入数据不同，WAFL，ZFS和btrfs不会覆盖包含活动数据或元数据的块，而是分配一个空闲块，并将修改后的数据或元数据写入其中。由于旧数据在事务处理期间被保存，所以创建快照的过程被简化。但是，每次覆盖也会废弃较旧的块（如果该块不属于任何快照）。这样的文件系统必须跟踪和回收这些不需要的块，为了确保为新分配提供足够的可用空间，其速度几乎需要与传入写入速率相同。

在本文中，我们介绍了基础技术并展示它如何满足高性能要求。我们描述了它如何适应文件系统大小的增长，同时兼容快照、克隆、重复数据删除和压缩等存储效率功能。我们也解释了删除快照后如何异步有效地回收空闲空间的情况。

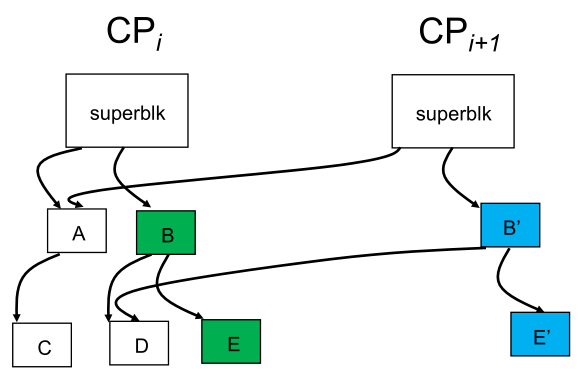
# 背景与动机

本节介绍了跟踪和回收可用空间到企业级质量文件系统的挑战，这些文件系统具有多个TiB的大小，并且可以处理每秒千兆字节（GiB / s）的操作。

## 块获得释放

写时复制（COW）文件系统不会在原处修改数据或元数据，它们将新数据写出到一个空闲的位置，从而保留以前的版本，如图2.1所示。

图2.1 写时复制文件系统



与事务模型一起写出一系列修改时，永久性介质上的文件系统即使在断电后也会保持一致。COW模型意味着覆盖几乎总是使块的不属于任何快照先前版本没有用。因此，1 GiB / s覆盖工作负载会生成1 GiB / s的潜在空闲块。这意味着一个COW文件系统必须：

（1）为新的写操作找到并分配1个GiB / s的空闲空间；

（2）修改元数据以记录1个GiB / s的空闲块，以保持文件系统的实时性。元数据的更新也会产生新的分配和可能的释放。

当文件被删除时，文件系统最终需要回收所有的块。由于某些应用程序突发性的创建和删除文件，以可预见的方式回收空间是一个重大的挑战。

创建和保留文件系统快照的能力对数据管理的生命周期至关重要，因为快照可用于数据保护、复制、恢复等。当快照被删除时，唯一属于它们的所有块都需要由文件系统回收。压缩和重复数据删除等提高存储效率的功能通常在后台运行，以减少空间和空闲块的浪费。文件系统必须选择回收上述活动生成的空闲块的比率，以满足客户对准确报告的需求，同时最大限度地降低对客户操作的性能影响。

## 空闲空间的延迟回收

传统上，文件系统保持其空闲空间元数据的实时性。例如，如果文件系统中的对象引用块b，并且所有这些引用都在事务中被丢弃到文件系统中，那么由该事务写出的可用空间元数据将指示b是空闲的。换句话说，事务持久化的元数据准确地跟踪所有被删除的块的引用，因此更新成本直接随着删除次数而增加。这为空闲空间回收工作提供了现收现付的模式。

文件系统还可以选择一个一致性模型，其中空闲空间元数据可能会变得陈旧，但保守。在这个模型中，删除一个参考变得非常简单：在前一个例子的事务结束时，即使没有对象引用b，文件系统元数据也指示仍然分配了b。在后台，定期扫描将遍历文件系统中的每个对象，以重建由其所有对象引用的块映射。这个想法已经在研究环境[9]和商业[24]上得到了探索。使用具有典型用户工作负载的标准硬盘，扫描这些元数据块可能需要2~4GiB/s的用户数据[9]。根据这些假设，在任何可用空间可以被回收之前，将花费超过一天的时间来扫描50TiB的用户数据集合。诸如压缩、重复数据删除、克隆等现代存储效率技术将更多用户数据打包到可用存储中，并允许用户数据块被不同的元数据块或文件指向，这会导致使用相同数量存储空间的元数据更多。因此，即使硬盘被像SSD这样速度更快的介质取代，多TiB文件系统的回收扫描仍然需要几个小时才能完成。虽然这对于相对静态的数据集来说可能是可以接受的，但是在繁重的写入工作负载情形下变得不可行。文件系统需要搁置大量的可用空间来对付不可预测的回收时间。

文件系统需要保留大约34％的可用空间，以确保覆盖成功，即使后台扫描程序运行得尽可能快。因此，除了快照以外，WAFL选择支付所有情况下保持其元数据的持续成本。

## WAFL中释放块的成本

WAFL是一种UNIX风格的文件系统，包含代表其文件的inode集合[11,8]。文件系统被写成一个以超级块为根的块树。WAFL中的每个文件系统对象，包括元数据都是一个文件。WAFL中的文件是4 KiB块的对称n元树，其中树的级别是。叶节点（L0）保存文件数据，下一级节点（L1）是指L0的间接块，依此类推。WAFL是一个COW文件系统，文件的每个修改块都被写入到一个新的存储位置，只有文件系统超级块被写入到位。这种方法的一个优点是可以将新的分配收集在一起并高效地写出来。

由于缓冲区和inode被客户端操作修改，所以它们被组合在一起以获得性能和一致性。每个可变的客户端操作在被确认之前也被记录到非易失性存储器中的日志中，日志中的操作会重播以在发生崩溃时恢复数据。WAFL从成千上万的已记录操作中收集结果脏缓冲区和inode，并使用称为一致点（CP）的检查点机制，将它们作为一个单独且非常大的事务在永久性介质中清除。每个CP都是一个原子事务，只有当它的所有状态都成功写入永久存储时，它才能完成。对内存数据结构的更新是隔离的并针对特定的CP，以确保每个CP代表文件系统的一致和完整的状态。为CP分配和释放的块也被捕获为对在相同CP中写出的分配元数据文件的修改。一旦属于CP的整个新块被保存到存储器中，新的文件系统超级块被就地原子性写入，指向这个新的文件系统树[8,11]。

文件系统使用不同的数据结构，如链接列表，位图，B（+）树等来跟踪空闲空间信息，并在设计中加入维护这些结构的成本。他们的块分配器要么直接使用这些结构，要么建立二级结构来有效地寻找空间。WAFL使用位图文件activemap作为主数据结构来跟踪块的分配状态或空闲状态。例如，如果activemap文件中第i位为0，则文件系统的第i个块是空闲的；如果该位为1，则代表该块已经被分配。WAFL使用4 KiB块大小，因此，1 GiB WAFL文件系统需要32 KiB的activemap文件，1 TiB WAFL文件系统需要32 MiB一个，依此类推。显然，多TiB大小的文件系统的元数据太大，不适合内存（WAFL缓冲区高速缓存[7]），需要通过持久性介质按需读取。

WAFL分配块号空间中的块，从而最小化对活动映射的更新。然而，释放可以在数字空间上随机分布，并且所有这种对于activemap的更新都必须在相同的事务中写出。更新越随机，该CP要处理的脏activemap缓冲区的数量越大。这会延长事务处理时间，降低文件系统的写入吞吐量。

# 空闲空间：基础

让我们首先定义activemap的一致性模型。如前面部分所述，超级块superblock指向定义WAFL文件系统的块的一致性树。这意味着树中的元数据准确地描述了树的分配块。

## 释放块的重用

**引理1.** 在COW文件系统中，事务CPn期间释放的块只能被分配用于后续事务，不能被分配给由事务CPn要写的新块。

因此，当CPn中的位ai被清除时，WAFL块分配器在CPn的超级块被写出之前不能使用bi。通过创建对应的activemap 4 KiB缓冲区的第二个副本可以实现。这两个副本被称为安全和最新的，块分配器查询前者，只在后者中清零。分配器在记录分配时设置两个副本中的位。当前的副本最终由CP写出，并在CP完成时删除安全副本。因此任何由于块释放而变脏的activemap缓冲区都需要两倍的内存，并且如果在一个CP中释放大量的随机块，则这变得昂贵。

## 循环依赖

activemap文件是一个普通的自我参照文件。像所有文件一样，它由块组成，这些块根据定义也由activemap进行跟踪。如果ai驻留在bj中，则写入bj的activemap块覆盖activemap块bi。

**定义2.** 以特定的activemap块开始的activemap链定义为列表中第i + 1个块覆盖第i个块的activemap块的最长列表。

根据定义，每个activemap块只能属于一个唯一的activemap链。因此，假定没有快照存在，当activemap缓冲区变脏，并且随后的CP分配一个新的块并释放对应于该activemap块的旧块时，它必须修改链中的下一个元素。如果该缓冲区尚未被修改，则重复此步骤。这一直持续到遇到先前被修改过的activemap缓冲区。因此，当任何块被释放并且其activemap位被清除时，该activemap块的整个链都将被更新。理论上，activemap的所有块都可以属于单个链，所以在最坏的情况下，CP可能会修改整个activemap文件。所以多TiB文件系统的activemap可能不适合主内存。

由于WAFL经常分配activemap中的块，所以不容易形成长链。它们只是由非常不可能的分配和释放序列组成的。WAFL已经建立了三个解决方案来解决这个问题。

**（1）预取整个链。**当activemap缓冲区被修改时，WAFL预取整个链。理想情况下，当CP开始为文件系统的activemap文件分配新块时，预取完成。

**（2）抢先打破链。**一个任务抢先发现每一个中等长度的链并修改第一个块，随后的CP打破了这条链。

**（3）延迟释放旧区块。**当一个CP处理脏的activemap缓冲区时，它只是将引用移动到一个新的文件，称为溢出文件。溢出文件的L1作为空闲activemap块的附加日志，而CP避免修改链中的下一个元素。一旦CP完成，溢出文件中的块才被真正释放；其相应的activemap位被清除。因此，n个元素的activemap链最多在n个连续的CP中完全分解。这确保了一致的CP长度而不违反定义1。

## 处理文件删除

要处理文件删除操作，WAFL将文件从目录的命名空间移动到隐藏的命名空间。但是，隐藏文件使用的空间不被回收，然后可能跨越多个CP，随着WAFL逐渐释放该文件的块，文件逐渐缩小，最终整个块的树被释放，隐藏文件的索引节点被回收。如果文件被随机覆盖，对activemap的更新更随机，因为它的树将引用随机块，我们称这些为aged files。

# 批量更新元数据

如前文所述，当在CP中释放大量的随机块时，可以会产生许多元数据更新。如果可以累积这些更新，则可以将它们分类为更高效的元数据更新。而且如果这项工作可以在多个CP的过程中完成，那么成本可以以可预见的方式随时间推移。

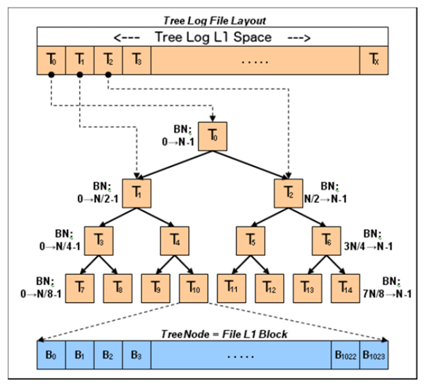
就像溢出文件一样，本节描述的两种结构通过取得所有权来延迟块的释放，这些块被这些文件的L1引用，我们称之为删除日志记录。文件系统一致性检查将像普通文件一样对待这些结构。排序部分地通过这些文件的L1上的块编号对参考进行排序而发生，一旦它们被充分排序，块的引用就从L1释放出来，相应的activemap以批处理方式清除。

## 树日志

WAFL中的删除日志机制的第一个方式是树日志（TLog）。如图4.1所示，TLog的L1引用TLog所拥有的块形成二叉树的节点，其中每个节点包含位于块编号空间的固定范围内的块引用。

将删除记录的块插入到根节点中；如果根节点填满，则根据其覆盖的范围将其内容推送到子节点；如果两个孩子都填满了，那么这个过程就会在树从上至下推送。如果树的叶子节点被填满，则释放其中所有的快，并更新activemap。TLog的大小决定了叶子节点覆盖的范围。在典型的部署中，大小约为文件系统大小1.5％的TLog提供了显着的性能改进，每个叶节点覆盖4到6个activemap块。

图4.1 TLog结构



但是TLog结构存在一个严重的问题：没有简单的方法来告诉有多少节点因为插入而被更新。这是二叉树的哪个部分受插入和树的该部分中的节点的充满度影响的函数。在极端的情况下，少量的插入会突然导致大量的节点被更新，这将导致CP需要处理更多的脏TLog缓冲区。为了缓解这些最坏情况的事件，基础设施将预取activemap缓冲区，当它检测到这些不好的情况时，它会停止删除日志记录，直接释放。我们决定逐渐从TLog结构转移到新的批量空闲日志BFLog解决方案。

## 批量空闲日志

BFLog采用了不同的方法，不是在其L1中嵌入排序数据结构。它累积了删除记录的块，并在一定的阈值之后，对它们进行排序和释放。

这可以通过几种方式来实现。在WAFL中，通过使用三个文件来完成：活动日志、非活动日志、排序日志。通过附加到活动日志的L1，块被删除记录。一旦活动日志填充到某个阈值，它就成为非活动日志，并被一个空的活动日志所取代。非活动日志中的删除记录块将在所有L1中排序。这是分两步完成的，一次对固定数目的L1内的块进行排序，然后进行合并排序，将块移动到排序日志。一旦排序日志准备好了，它的块就会被释放，并且activemap会以一种方式被更新。BFLog数据结构的所有处理本质上是本地化的或顺序的。

一旦BFLog达到稳定状态，就有可能调整其操作，使得平均而言新块的每个附加结果导致一个块在排序日志之外被排序、或被合并排序、或被打孔。这与TLog的不可预测性形成鲜明对比，因为TLog的分类严格受二叉树内容的控制。实验表明，将BFLog所有三个文件一起的大小调整到文件系统的0.5％可以提供足够的性能提升。

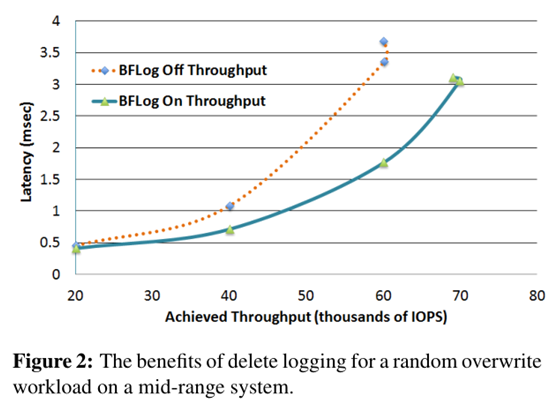
## 登录参考计数文件系统

WAFL的一些存储效率特性取决于一个块可以被多个文件共享。对块的额外引用由refcount文件进行跟踪，该文件是一个整数平面文件，每个块一个。当一个重复引用的块被释放时，refcount文件被更新。由于该文件比activemap密度少了几倍，由块的随机释放引起的问题增加了多样性。现在无处不在的重复数据消除功能导致调用refcount文件次数增加，这使得删除日志记录更加关键。重复引用块的删除记录与常规块的删除记录没有区别。

## 评估

我们首先研究了随机覆盖工作负载的好处。配置了一组LUN[23]，并使用多个客户端来生成对LUN的随机8 KiB重写流量，模拟数据库OLTP写入。虽然BFLog的好处在高端平台上稍微有些弱点（我们观察到吞吐量大约有5％到10％的提高），但是在中端平台上它们更高。图4.2显示了具有12个内核（Intel Westmere）和98 GiB DRAM的中档系统的结果。

图4.2 删除日志记录在中档系统随机覆盖工作负载上的性能



没有删除日志记录，我们的吞吐量稳定在约60k IOPs，而使用删除日志记录，我们能够继续约70k IOP，这代表吞吐量大约提高了17％。除了吞吐量的提高之外，我们还观察到大多数负载点的延迟都有34％到48％的提高。这些改进是通过将元数据开销减少65％来实现的，因为BFLog能够批量合并元数据块的大量更新。

# 快照和可用空间

快照是许多数据管理功能的重要组成部分，文件系统快照是整个文件系统的时间点版本。在WAFL中，通过存储CP的超级块的额外副本，创建快照非常简单。在快照被删除之前，快照中的块都不能被释放。本节介绍可用空间回收如何保护和回收快照中保留的块。

我们引入一个新的术语volume（卷），其中包括当前文件系统及其所有快照。当前的文件系统是卷的活动部分，为CP机制正在写入修改的所有客户机操作提供服务。快照用Si表示，当前的文件系统用S0表示。

S0中的activemap不能告诉我们哪个块被困在给定的快照中。最初，S0的activemap与新快照S1的activemap的上的信息完全相同，但是由于S0的改变，它迅速发散。S1的块已经被记录在它的activemap（称为snapmap）中，由于文件系统的COW性质，它们保持不变。因此，S1使用的块可以在S0中释放，然后在activemap中标记为空闲。

**引理2.** 一个卷所指向的块集合可以用其所有快照的snapmap的位或来表示。

## 汇总图

如果在activemap和所有快照中是空闲的，则该卷中的块是空闲的。后一种情况需要更多的IO和CPU来确定。有几个解决这个问题的选择：一个文件，每个块存储一个位，每个快照一个位，一个B +树，以一些可搜索但精简的方式捕获每个快照中的空闲和分配的块等。WAFL使用的位图文件称为汇总图，这是卷中所有快照的snapmap的位或。这个简单的方式实现了低消耗的空间跟踪。

当快照被删除时，汇总图中的一些位需要被清除，并且以原子方式执行（作为单个CP的一部分）是不切实际的。还有其他一些原因使得这个会中途瞬间变得陈旧，但是保守。WAFL支持一种名为卷snaprestore的功能，其中一个卷可以及时向后跳转到一个较旧的快照。这是通过将Si的超级块写入下一个CP的超级块来完成的。Si创建时的概要图被冻结了，在snaprestore之后，成为了S0的概要。但是，此摘要可以包含属于早于Si的快照的块，这些快照已被删除。

## 快照创建

创建快照时，汇总图尚未使用新快照的块进行更新，后台快照创建扫描将快照映射到摘要中。应该指出的是，这个扫描是幂等的，所以它可以安全地重新启动后从头开始。

**定理1.** 假设没有删除的快照，一个卷所指向的块集合总是保证等于activemap和摘要的“或”，即使在快照创建扫描正在进行中。

在S1的扫描完成之前，可以创建新的快照S2。在这种情况下，扫描将重新启动，但会切换到将S2快照映射到摘要中。这种情况不需要再处理S1的快照，因为所有属于S1的块都属于S2，或者在创建S2之前被释放。

## 另一个循环依赖

快照创建扫描正在进行中时，另一个循环依赖关系将产生。假设快照S1被创建，所以activemap和S1的snapmap是一样的。如果bi被释放，则ai被清除，并且汇总图中的第i位由快照创建按需工作设置。这会导致要修改汇总图缓冲区，这意味着该汇总块bj的旧版本必须被释放。所以aj得到清除，依次设置第j个汇总位，依此类推。虽然可能性较小，但长的activemap-summary链可以被创建，这是activemap链问题的两倍。一个客户系统在大小为80TiB的老化卷上遇到这个问题。CP使用溢出文件推迟释放旧的汇总块，从而避免修改链中的下一个activemap。

## 快照删除

从命名空间中删除快照时，其块可能会反映在汇总图中。背景快照删除扫描遍历摘要以删除专属于该快照的块。此扫描也是幂等的，可以随时安全地重新启动。

**定理2.** 独立于快照创建和删除的扫描状态，卷所指向的块集始终等于activemap和汇总图位“或”的子集。

**引理3.** 让卷中的所有快照（包括S0）按创建的时间顺序排序。对于卷中的任何块bi，通过连接所有snapmap的第i位创建的位串（在S0的情况下，以该时间顺序的activemap）将产生0 \* 1 \* 0 \*模式。这被称为010快照属性。

删除扫描有两种方法可以更新摘要，其中一种使用010属性。

**（1）通过减法删除（或dbys）。**此扫描使用010属性来处理Si的删除。如果一个块被Si使用，而不是由它的临时邻居快照中的任一个使用，则它独占属于Si。换句话说，如果Si的snapmap中的位被设置，但是邻居快照的快照中的位不是，则扫描清除汇总位。在这个方案中，最新的快照没有更新的邻居，最旧的快照没有旧的邻居。假设dbys扫描正在进行时删除第二个快照Sj，一个单独的扫描可以处理该Sj，并且因为结果是幂等的，所以两个扫描可以独立工作，只要它们不同时更新相同的位。如果Sj碰巧是Si的邻居，那么相同的扫描会用下一个邻居代替Sj并继续进行。现在扫描正在代表两个删除更新摘要。当扫描到达最后一位时，它完成了Si的删除，但是它会环绕并继续处理Sj的删除，直到它回到Sj被删除的状态。

**（2）通过添加删除（或dbya）。**此扫描只需简单地对所有剩余快照的快照进行“或”操作即可重建汇总图。如果在扫描过程中删除了第二个快照，则可以重新启动扫描以获取其余快照。在实践中，扫描继续向下位图空间以利用已经发布的snapmaps和汇总图的预提取是更好的。当扫描到达摘要中的最后一位时，它将环绕并继续，直到回到发生第二次删除的状态。

如前所述，删除扫描的两种模式都可以在重新启动后重新启动，而不会影响结果的正确性。

# FlexVol虚拟化层

WAFL使用虚拟化技术，允许将数百个或更多卷托管在称为聚合的同一物理存储介质集合中。这是提供了FlexVol克隆、复制、自动精简配置等几项新功能的关键。文献[8]的第3节解释了这种分层：每个聚合都是一个WAFL卷，其中包含称为物理卷块编号（pvbns）的块，可直接映射到块在永久性存储介质上，并且每个FlexVol是一个带有称为虚拟卷块号（vvbns）的WAFL卷，它们实际上只是集合WAFL卷中的一个文件中的块；该文件被称为FlexVol的容器文件。换句话说，FlexVol中的块的vvbn与其对应的容器文件中的块偏移相同。

每个图层维护位图来跟踪其空闲块，每层支持快照。换句话说，FlexVol使用其activemap、汇总图、BFLog等来跟踪分配的和空闲的vvbns，并且聚合使用另一组元数据来跟踪pvbns。在FlexVol中完全释放vvbn后，需要在容器文件的适当L1偏移处有效地打出物理块，然后才能在聚合的activemap中标记相应的pvbn。在vvbn被释放之后不久就要做到这一点非常重要，这样所有的FlexVol都可以不受限制地共享可用的聚合空间。然而，这意味着释放一个随机的vvbn需要加载和更新一个随机的容器文件L1，由于因为L1比位图密度低几倍，这个操作的消耗很大。即使在第四部分的结构实施之前，这也激发了WAFL的一个重要的性能优化。WAFL延迟从容器的空闲批次。WAFL维护一个容器文件中每32K个连续vvbns的延迟空闲块的数量，这个vvbns被写入到每个FlexVol文件中。

延迟释放的总数有软限制，这样文件系统就可以吸收大量的延迟释放空间。删除专门拥有多个块的快照，卷快照或其他事件可能会产生这样的延迟释放空间突发。当超过这些限制时，系统会尝试紧急地处理那些积压。由于BFLog批量释放了activemap，所以如果在该范围内有足够的延迟释放空间，它可以选择执行“容器打卡”工作。

## 评估

我们首先展示了无延迟优化的重要性。基准是内置的读/写混合模式，用于对OLTP /基准测试应用程序的查询和更新操作进行建模。它的构建与存储性能委员会-1（SPC-1）基准[5]非常相似。图6.1显示了实现的IOPS（左侧y轴）和延迟（右侧y轴）。延迟释放相应的标准化潜伏期降低了60％。这表明，即使在使用SSD的高端系统上，随机容器L1读取和更新也会降低性能。

为了衡量BFLog的优化，我们运行了测试，随机覆盖32 KiB大小的I / O的数据集，而专门拥有大量块的FlexVol快照被删除。这是在具有4个Intel Wolfdale内核和20个GiB DRAM的低端系统上完成的。图6.2显示了结果，客户端的写入负载增加了26％的吞吐量，因为右边的条形图显示BFLog能够更高效地执行“容器打卡”工作；其中约84％是在优化模式下完成的。

图6.1延迟释放容器的性能

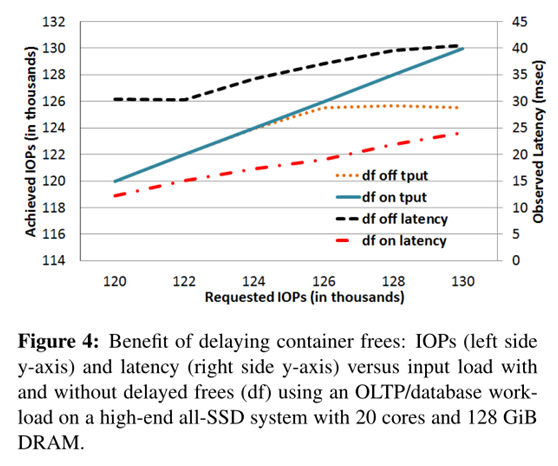
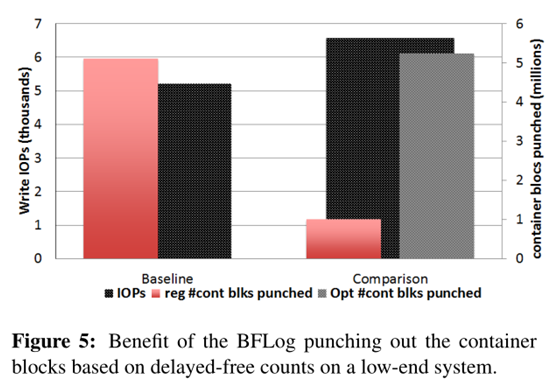


图6.2 BFLog在低端系统基于延迟计数的基础上释放容器块的性能



# 总结

随着技术发展的趋势，更快的硬件和更大的文件系统将会产生，WAFL文件系统已经发展了二十多年。它也从处理用户目录风格的NFS / CIFS工作负载转变为服务于广泛的SAN和NAS应用程序。它已被部署为以不同的配置运行：专门构建的平台，包括硬盘和固态磁盘的所有组合，软件定义的解决方案，甚至AWS上的NetApp CloudOntap®实例。空闲空间回收的基础架构在所有这些排列中发展良好，提供了丰富的数据管理功能和始终如一的高性能。即使释放的模式和速率可以是非常随机的，我们认为提出的技术可以实现空闲空间回收继续和行为确定性，而不会影响系统的其余部分。我们描述了有效的方法来跟踪分配给快照的空间。最后，我们展示基础架构如何在FlexVol虚拟化的额外层中工作。

参考文献

[1] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In Proceeding of ACM Symposium on Parallel algorithms and architectures, pages 81–92, 2007.

[2] Jeff Bonwick. Space maps. https://blogs.oracle.com/bonwick/en/entry/space\_maps, 2007.

[3] Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In Linux Symposium, page 263, 2008.

[4] Standard Performance Evaluation Corporation. Spec sfs 2014. https://www.spec.org/sfs2014/.

[5] Storage Performance Council. Storage performance council-1 benchmark. www.storageperformance.org/results/#spc1\_overview.

[6] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In Proceeding of Symposium on Operating Systems Design and Implementation (OSDI), 2016.

[7] Peter R. Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system. In Proceeding of 45th International Conference on Parallel Processing (ICPP), 2016.

[8] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A.Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In Proceedings of the 2008 USENIX Annual Technical Conference, pages 129–142, Jun 2008.

[9] Travis R. Grusecki. Improving block sharing in the write anywhere file layout file system. http://hdl.handle.net/1721.1/76818, 2012.

[10] Christopher Hertel. Implementing CIFS: The Common Internet File System. Prentice Hall Professional Technical Reference, 2003.

[11] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In Proceedings of USENIX Winter 1994 Technical Conference, pages 235–246, Jan 1994.

[12] NetApp Inc. Data ONTAP 8. http://www.netapp.com/us/products/platform-os/ontap/, 2010. [13] Ram Kesavan, Sriram Venketaraman, Mohit Gupta, andSubramaniamPeriyagaram. Systemsandmeth-ods for instantaneous cloning. Patent US8812450, 2014.

[14] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In Proceedings of Conference on File and Storage Technologies (FAST), 2017.

[15] Avantika Mathur, Mingming Cao, and Andreas Dilger. ext4: the next generation of the ext3 file system. Usenix Association, 2007.

[16] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. ACM Trans. Comput. Syst., 2(3):181–197, August 1984.

[17] Sun Microsystems. ZFS at OpenSolaris community.http://opensolaris.org/os/community/zfs/.

[18] Peter M. Ridge and David Deming. The Book of SCSI. No Starch Press, San Francisco, CA, USA, 1995.

[19] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. Trans. Storage, 9(3):9:1–9:32, August 2013.

[20] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems, 10:1–15, 1992.

[21] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem, 1985.

[22] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In USENIX Annual Technical Conference, 1996.

[23] Ralph H. Thornburgh and Barry Schoenborn. Storage Area Networks. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[24] David D. Wright. Data deletion in a distributed data storage system. Patent US8819208, 2014.