计算机研究与发展 ISSN 1000-1239/CN 11-1777/TP
Journal of Computer Research and Development 47(10): 1832-1839, 2010

对象存储设备端数据管理策略研究

刘景宁 谢黎明 冯 丹 吕 满
(华中科技大学计算机科学与技术学院 武汉 430074)
(武汉光电国家实验室光电信息存储研究部 武汉 430074)
(j.n.liu@163.com)

Research on Object Storage Device End Data Management Strategy

Liu Jingning, Xie Liming, Feng Dan, and Lü Man
(*School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074)
(*Division of Data Storage System, Wuhan National Laboratory for Optoelectronics, Wuhan* 430074)

Abstract With the increasing demand of high I/O throughput, highly parallel data transfer and highly scalable storage system, especially in the supercomputing field, the traditional storage system has been difficult to meet the requirements. OBS (object-based storage) system is a new network storage architecture. In OBS system, local storage space allocation is managed by intelligent OSD (object-based storage device), and at present, the OSD is mainly managed by the common file system. However, when the common file system deals with the flat namespace, especially in the course of long-term use, the performance degenerates seriously. In this paper a new file system is proposed. XOBFS, which stands for extensible hashing object-based storage file system, uses fixed-length block allocation strategy and manages free block bitmap combination. Based on expanding hash manage object attribute, the same object attributes are stored in adjacent hash bucket. XOBFS applied in OSD has a lot of advantages. For example, the metadata has small scale, long-term performance is not degraded, and attribute of the object is managed effectively and so on. Test results show that big-object based XOBFS throughput rate is better than that of the traditional file system. XOBFS provides an effective method for storage management in OSD.

Key words object-based storage system; object-based storage device; object-based file system; object attribute; extensible hashing

摘 要 在对象存储系统中,数据由智能化的对象存储设备管理.当前,对象存储设备端主要提供对象接口,传统的设计主要由通用文件I/O来封装.但是,通用文件系统在管理平坦命名空间时,尤其在长期使用后性能退化严重.因此,提出基于扩展Hash的对象文件系统XOBFS(extensible hashing object-based storage file system),将磁盘空间划分成多个区域单元,区域中定长块分配策略与位图管理空闲块方式结合;对象属性用扩展Hash管理,在Hash桶中采用相同对象属性相邻存放策略.XOBFS应用在对象存储设备端,具有元数据规模小、长期使用性能不退化、对属性进行有效管理等特点.测试结果表明,基于大对象的XOBFS的吞吐率优于传统文件系统.

关键词 对象存储系统;对象存储设备;对象文件系统;对象属性;可扩展Hash
中图法分类号 TP311

收稿日期:2008-10-31;修回日期:2010-02-09
基金项目:国家"九七三"重点基础研究发展计划基金项目(2004CB318201);国家"八六三"高技术研究发展计划基金项目(2009AA01A401,2009AA01A402);国家自然科学基金项目(60703046);长江学者和创新团队发展计划基金项目(IRT0725)

0 引 言

随着存储需求的增加以及存储应用日益复杂, 传统的存储难以满足需求. 对象存储很好地结合了传统的文件模式和块模式, 保留了文件的逻辑性和语义性, 又具有块的传输高性能, 较好地满足安全性、跨平台数据共享、高性能及可扩展性等要求.

对象存储系统将传统存储系统中的物理存储管理下移到智能化的对象存储设备中, 建立拥有自主管理、数据共享、安全和智能化等特征的存储网络. 保存对象的存储设备称为基于对象的存储设备(object-based storage device, OSD)[1], 对象存储设备是对象存储系统中数据的载体, 能够自行管理其上的数据分布. 对象存储系统中的对象是数据的一种逻辑组织形式[2]. 对象是可变长的, 可包含任何类型的数据. 对象具有属性, 用于描述对象的特征, 由 ID 号标识. 对象存储设备端文件系统在对象文件系统中处于底层, 负责本地数据存储管理. 对象文件系统的上层, 即文件管理部分将文件请求转化为对象请求, 通过 iSCSI 协议与设备端通信. 设备端接收并处理对象请求, 向上层返回处理结果.

1 相关工作

目前, 对象存储设备端文件系统主要有 IBM Haifa 实验室的 ObjectStone, University of California Santa Cruz(UCSC) 的 OBFS 以及华中科技大学的 LOBFS. 这 3 种文件系统在文件与对象的对应关系、磁盘块管理方法上存在较大差别.

IBM 的 ObjectStone 用 Linux 本地文件系统管理对象[3]. 在本地文件系统(如 Ext2, Ext3) 上针对树型结构进行了优化, 用户在树型文件系统中进行查询操作; 对于一维平坦的对象 ID 空间, Linux 本地文件系统没有优势, 因线性查找, 单个目录操作的时间复杂度是 $O(n)$[4]. 加州大学 Santa Cruz 分校的 OBFS 中提出了区域概念以及分块大小的设定. OBFS 中具有 RAID 功能; 大文件被划分成固定长度的对象; 设备端磁盘按对象区域进行划分, 每个域中的对象长度相同; 这种分配方式具有性能不退化、磁盘空间连续性高等优点. 但是, 该系统没有涉及如何处理对象属性、如何从对象属性中挖掘规律. 华中科技大学的 LOBFS 具有 OBFS 的优点, 还涉及到对象属性的处理, 丰富了系统的功能. 但是, 该系统用 B^+ 树管理空闲块以及对象存储索引节点对缓存的依赖性太强, 为了避免访问对象时进行多次磁盘 I/O, 需要将所有 B^+ 树结点读入内存. 随着系统管理对象数目的不断增多, B^+ 树结点规模逐渐庞大, 缓冲区必定满足不了需要. 另外, 元数据的一致性维护开销也不容忽视.

综合上述文件系统的优点, 本文提出一种对象存储设备文件系统 ——基于扩展 Hash 的对象文件系统(XOBFS), 目录操作时间复杂度降低为 $O(1)$. XOBFS 吸收了 OBFS 系统区域的思想, 并使用扩展 Hash 管理对象元数据. 采用定长块分配策略, 用位图方式管理空闲空间. 在经历了长期的对象创建删除后, 系统的仍然能够保持较高吞吐率.

2 XOBFS 的设计与实现

LLNL 科学计算数据表明, 占文件系统容量的 80% 的文件大小集中在 512 KB 和16MB之间, 61.7% 的文件大小集中在 2 MB 与8MB之间, 并且大约 83% 的对象是 512 KB 的大对象[5].

为了使得大多数对象与系统分条单元大小相同, XOBFS 将系统条块分配粒度设定为 512 KB, 在最坏情况下, 这种条块划分方式将会浪费少于 40% 的磁盘空间. 但大粒度的分配带来大粒度的空间回收, 这样可以保证文件系统空间的连续性.

文件系统 XOBFS 将磁盘空间划分成许多独立的区域, 对象不能跨区域存放. 区域内部数据块大小与对象大小保持一致, 均为 512 KB. 文件系统将重要的元数据分布到各个区域中, 对它们进行单独操作, 将错误限制在区域内部. 在系统长期运行时能够有效减少碎片, 使系统性能不退化. XOBFS 的文件与对象是一对多的关系, 即可以将一个大文件分割为多条, 按 RAID 的形式将文件分块存放到一组对象中. 设备端文件系统管理的对象大小固定, 长度为 512 KB.

2.1 XOBFS 元数据

XOBFS 中包括 5 种基本元数据: 对象存储索引节点(Onode)、数据块位图(bitmap)、区域超级块(supper block)、Hash 目录表(directory) 以及元数据 Hash 桶(metadata bucket). 对象存储索引节点中存放对象号、块地址、对象偏移和对象长度. 位图记录本区域内数据块的使用情况. 区域超级块中包含了区域大小、区域类型、位图和 Hash 目录的区域内偏移地址, Hash 目录表的深度. Hash 目录表实现对象号到对象存储索引节点的转换. 元数据桶是一个 Hash 桶, 对象存储索引节点以及对象属性都包括在内, 并且同一对象的属性相邻存放. XOBFS 文件系统的格式如图 1 所示:

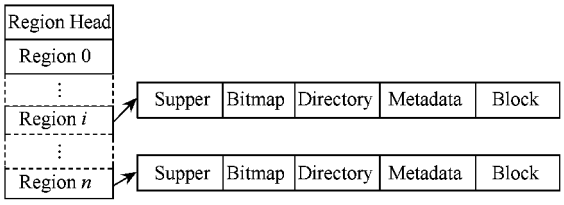


Fig. 1 XOBFS file system.

图 1 XOBFS 文件系统格式示意图

2.2 对象映射机制

对象映射机制由 OSD 负责, 分两个部分: 一是对象号到对象存储索引节点的映射; 另一个是对象属性号到对象属性的映射[6].

作为动态 Hash 中的一种方法[7], 扩展 Hash 的思想体现在, 先在已有桶中寻找空位, 若有空位则插入记录, 若映射目录表已满则溢链, 溢链不增加桶数. 当出现新 Hash 值时增加新桶. 图 2 描述了目录表深度为 2 时扩展 Hash 的映射过程. 其中, 编号位数 i(即目录表深度)与桶数 M 的关系为 $i = \lceil \mathrm{lb}\ M \rceil$, 扩展散列中的记录地址为"桶号+ 桶内偏移地址".

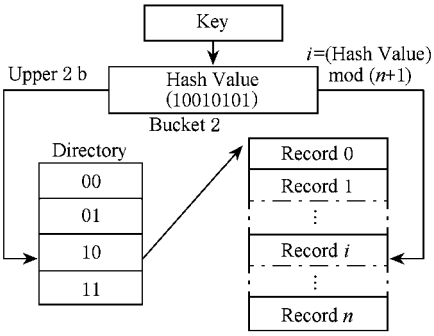


Fig. 2 Mapping with extended Hash when the directory depth is 2.

图 2 目录表深度为 2 时扩展 Hash 的映射过程

OSD 在接收到用户对象请求后, 根据对象号计算 Hash 值, 按照 Hash 目录表的深度(假设深度为 d), 取高 d 位得到 Hash 目录表索引号 i. 索引号 i 在 Hash 目录表中的位置就是对象存储索引节点的地址. 目录表深度为 2 时, XOBFS 的映射过程如图 3 所示:

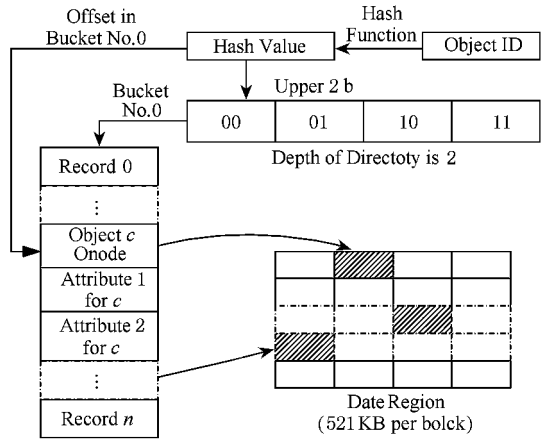


Fig. 3 Mapping between ID and Onode.

图 3 对象号到对象存储索引节点映射过程

对象属性号到对象属性的映射过程与对象号到对象存储索引节点的映射过程类似. 对象属性的 Hash 值也是由对象号生成, 这样做是为了实现相同对象属性相邻存放. 对象属性的 Hash 值在 Hash 桶中不能唯一确定对象属性, 需要使用对象属性号才能完成一次查找操作. 对象属性号到对象属性的映射过程如图 4 所示. 对象属性 Hash 表使得对象号到对象存储索引节点的映射时间复杂度变为 $O(1)$.

扩展 Hash 算法空间花销较小. 如在一个大小为 512 MB 的区域中, 数据块按照 512 KB 划分, 位图大小仅为 128 B. 最差情况一个 Hash 桶只能存放一个对象存储索引节点, 则映射 1024 个数据块需要 1 024 个桶, 故目录表为 1 024 项, 大小为 1024 × 4 B= 4 KB. 超级块大小小于 50B. 而需要常驻内存的只有

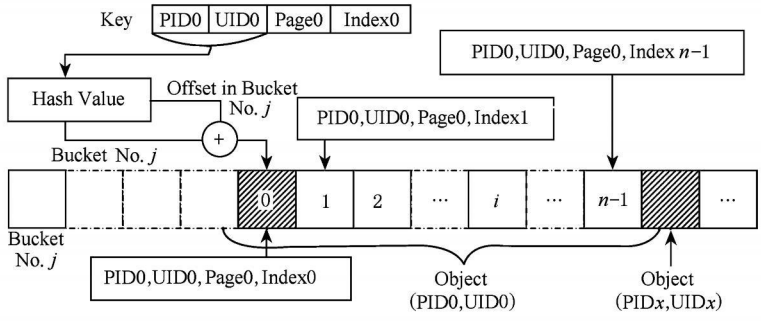


Fig. 4 Mapping between ID and attributes.

图 4 对象号到对象属性的映射过程

超级块、目录表和当前桶(桶大小为4 KB),总计50 B + 4 KB+ 4 KB= 8 242 B. 如果是256 GB的硬盘,按照512 MB一个区域划分,共有512个区域. 需要缓冲区的大小为512× 8 242 B,小于5 MB. 然而,对于传统文件系统来说,若采用4 KB的分块大小,则需要至少8 MB的位图,以及几百KB的inode位图.

扩展Hash算法不足之处在于Hash桶分裂时,桶内数据要在桶间迁移. 并且,随着桶数增多目录表深度增加,目录表项也将会成倍增长. 针对扩展Hash存在的问题,XOBFS将设备划分成若干区域,并且每个区域大小确定. 根据上面讲述的区域中数据块大小固定的特性,将位图、Hash目录表以及元数据桶大小设置为固定值. 这样,在某个区域内,扩展Hash目录表表项不会增加,元数据桶也不需要分裂,避免了扩展Hash算法的不足之处.

2.3 对象数据区空间分配

磁盘是一维线性块空间,这个空间划分为两部分:空闲空间和已分配空间. 文件系统的功能是管理这一维线性空间. 空闲空间连续度和已分配空间连续度直接影响到文件系统性能. 连续存放的对象可以减少磁头寻道、扇区定位时间,有利于以大块I/O的形式提高数据吞吐率. 在I/O密集型应用中,频繁的对象创建、写入及删除会导致空间连续度不断降低,空闲空间会变得越来越破碎,使得文件系统的性能越来越差.

在XOBFS中,对象数据块分配粒度采用大粒度定长块分配. 分配粒度指的是指一次性的从空闲空间分配数据块的长度. 大粒度有利于保持较好的已分配空间连续度. 另外,大粒度分配会意味着大块数据回收,有利于保证空闲空间的连续性.

OSD根据对象内部的块与块之间的逻辑关系,尽可能为对象分配连续的存储空间. 当OSD收到写命令,请求把对象内某个偏移地址处的块写入,此时OSD采用定长块分配为该块分配物理空间. OSD采用位图方式管理空闲块,位图方式实质上是按地址排序的方式管理空闲块. 按地址排序可以尽可能地为文件分配地址连续或邻近的数据块,从而减少寻道时间,提高预取效率. 位图方式不考虑空闲块大小,顺序分配空闲块,增加产生外部碎片的可能. 但是定长块分配可以弥补这个缺陷. 两者在OSD中结合使用,使得设备端文件系统性能长期不退化.

2.4 对象属性的管理

对象属性用于描述数据特征以及安全策略. 对象属性包含数据的公用信息,有静态属性(如建立时间等)、动态属性(如最后访问时间、访问频度)以及文件元数据(文件名、组、用户ID)等,不同种类对象的属性都不相同. OSD可利用这些属性对数据的组织进行优化,可通过对象属性了解外部环境,合理地对资源进行预留和分配,提高Cache的命中率和预取的效率. 可见,对象属性在对象存储系统中具有重要的地位. 对象属性的管理有效程度影响了存储系统的性能.

但是,对象属性的扩展性使得对象元数据管理面临重大挑战. UCSC的OBFS的对象属性借用传统文件系统的做法,将属性信息存放在对象存储索引节点中,并且对象存储索引节点与对象数据一起存放在固定大小的数据块中. 随着应用环境的变化,如果对象属性需要进行扩展,UCSC的OBFS将不能满足对象属性的可扩展性[8].

XOBFS的一个重要特点是对对象属性进行了有效管理,并能满足对象属性的扩展性. 属性管理方法的设计思想是以扩展散列表为基本数据结构,将对象属性进行集中存放并采用基于效益值的缓冲替换策略[9]. 在XOBFS中,相同对象属性相邻存放,在表中查找某一记录属性时,最少需要查找1次才能命中,最差情况则需要遍历整个桶,即需要查找S次. 当访问某一属性记录时,首先在目录表中查找一次,再到Hash表中查找. 假设该记录所在桶已在缓冲区中,查找次数最小为2次,最多为1+ S次. 平均查找时间为1+ $S/2$次. Hash表中删除某一属性记录时,因相邻存放,这些属性记录存放在同一桶中. 假设该桶中待移动记录数为Q,对象属性记录有M条,且$M \leqslant S$, $Q \leqslant S$. 则删除成本为 $\sum_{i=1}^{M}(Q- i)=(2\times Q- 1- M)\times M/2$,由Hash值映射到该桶时成本为1,则总成本为1+ $(2\times Q- 1- M)\times M/2$.

如果OSD上存在N个对象,每个对象平均具有M个对象属性. 假定OSD上存在一个对象O_1,并且O_1具有m个属性,满足$m \leqslant M$. 下面按对象管理的3种方法进行比较.

方法1: 该M属性对应M个对象属性文件. OSD文件系统(OSD file system, OSDFS)需要管理$(M+ 1)\times N$个文件(N个数据文件, $N\times M$个属性文件). 对象属性文件由OSDFS管理,增加了文件系统的负担. 每访问一次对象属性都必须经过OSDFS访问对象属性文件. 访问对象O_1的m个属性时,共需要访问m个文件.

方法2: M个对象属性可以用Onode进行存

放,即 UCSC 的 OBFS 适用于假设 2. 当访问对象 O_1 的 m 个属性时,首先通过 1 次 I/O 操作读出对象 O_1 的数据以及该对象 O_1 的 Onode 节点,然后,在内存中,从 Onode 中检索出 m 个目标属性.

方法 3: M 个对象属性使用 XOBFS 进行属性管理,即该 M 个属性采用扩展 Hash 方法进行属性号到属性的映射,并且相同对象属性相邻存放. 当访问对象 O_1 的 m 个属性时,首先通过 1 次 I/O 操作读出对象 O_1 的属性所在的 Hash 桶,再通过 Hash 映射检索出 m 个目标属性.

方法 1 虽然满足对象属性的扩展性需求,但是性能明显低于方法 2 和方法 3,但是方法 1 需要的 I/O 次数高于方法 2 和方法 3,频繁的 I/O 操作对性能影响不能忽视. 方法 2 与方法 3 的空间复杂度为 $O(1)$,时间复杂度为 $O(n)$. 但是方法 2 不能满足对象属性的可扩展性. 因此,XOBFS 更能有效管理对象属性.

2.5 缓冲策略

为了减少磁盘 I/O 次数,Hash 桶常驻内存. 为了降低空间复杂度,只对最经常用的 Hash 桶进行缓冲. 按照目录表的映射子区间对 Hash 值的映射次数进行统计,所谓映射子区间是指具有相同映射概率的目录表项所组成的集合[9]. 如图 5(a) 所示,该目录表的映射子区间为 A_1 = {000, 001, 010, 011}, A_2 = {100, 101}, A_3 = {110, 111},其映射关系的二叉树描述如图 5(b) 所示. 映射子区间的元素在二叉树的同一层,具有相同的映射概率. 映射子区间的权值为子区间的元素个数,如 A_1 的权 $A_1.w$ 为 4, A_2 的权 $A_2.w$ 为 2, A_3 的权 $A_3.w$ 为 2. 在向后 T 次映射中,映射到子区间的次数,称之为映射价值. 当计数器计数满 T 次后,清零重新计数. 当计数器满 40 次时,共访问区间 A_1 14 次,区间 A_2 20 次,区间 A_3 6 次,则 A_1 的价值 $A_1.v$ 为 14, A_2 的价值 $A_2.v$ 为 20, A_3 的价值 $A_3.v$ 为 6. 再根据各子区间的效益值决定缓冲桶数. 效益值 P 计算公式为 $P = A.v/A.w$,即子区间权为 1 时对应的价值,则 $A_1.P$ = 3.5, $A_2.P$ = 10, $A_3.P$ = 3. 取具有最大 P 值的子区间对应的桶数为缓冲桶数. 这里由于 $A_2.P$ 最大,故缓冲桶数为 A_2 对应的桶数 1.

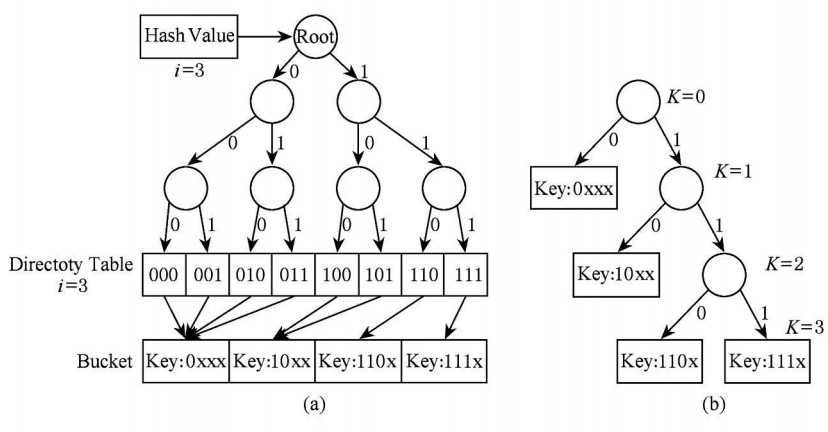


Fig. 5 Mapping between ID and attributes when the directory depth is 3.

图 5 目录深度为 3 时目录索引与桶号映射关系

若访问序列 $J = \{J_1, J_2, \cdots, J_{n-1}\}$,缓冲区中桶集合 $T = \{T_1, T_2, \cdots, T_m\}$. 第 J_i 次访问缓冲区中桶 T_j,这时 $J_i \in J$. 当 $T_j \in T$ 时,不需磁盘 I/O. 当 $T_j \notin T$ 时,需要调入新桶到缓冲区中,设调入新桶数量为 P,则磁盘 I/O 数为 P 次. 调入新桶时,可按 LRU-K 替换算法以及 LCBK 替换算法淘汰效益值最小的旧桶[9-10].

3 性能测试

目前 Linux 典型的文件系统有 Ext2, Ext3, 根据面向对象思想设计的 ReiserFS[11-12] 以及由 IBM AIX 系统移植过来的 JFS[13]. 利用合成 Trace 对 XOBFS 以及上述几种文件系统进行模拟使用过程仿真测试. 通过吞吐率来比较文件系统管理的性能.

测试 Trace 由美国 Lawrence Livermore 国家实验室的高性能科学计算集群收集[5] 的 Trace 根据负载特征合成得到[14]. 针对两种典型的应用: 读密集型业务 ($read\_rate$ = 70%) 和读非密集型业务 ($read\_rate$ = 30%),该合成 Trace 模拟了长期使用文件系统的过程,插入了大量的对象创建和删除操作,并根据磁盘剩余容量不断调整对象创建和删除

操作的比例.

这两种请求分布如表 1 所示. 这两个 Benchmark 创建的对象大小的分布特点符合负载分布情况. *Read_Trace* 是读密集型 Trace, 读请求的总字节数约占 70.0%, 创建/写及删除分别占 15.8% 和14.2%, 而 *Write_Trace* 则读、创建/写及删除的总字节数分别占 30.0%, 35.8% 和 34.2%. 利用两种 Trace 在磁盘上进行随机读写删除操作, 测试 XOBFS 以及其他文件系统的吞吐率. 测试平台使用武汉光电国家实验室公用实验平台, 硬件环境如表 2 所示.

Table1 Request Distribution of Synthesized Trace
表 1 合成 **Trace** 的对象请求分布

Operations	*Read_Trace* Object Number (Total Bytes/GB)	*Write_Trace* Object Number (Total Bytes/GB)
Read	70008(22.6)	29877(9.6)
Create/Write	15752(5.1)	35826(11.5)
Delete	14240(4.6)	34297(11)

Table 2 Testing Platform
表 2 测试平台硬件环境

Hardware	Types
CPU	Intel Xeon 3.0 GHz
Mainboard	Supermicro X6DHE-XB
Memory	DDR ECC RG 512 MB
Hard Disk	Maxtor DiamondMax 10/SATA150 200GB/7200 r/8MB Cache

测试所用的磁盘为 Maxtor DiamondMax 10 SATA150. 数据传输率为 1.5 Gbps, 平均时延 4.17 ms. 测试时主机运行的是 Linux 操作系统, 内核版本为2.4.21-4.EL. 将 200 GB 的磁盘进行分区, 每个分区 10 GB. 在每个分区上安装上述几种文件系统.

在通用文件系统中, 缓冲策略很完善, 小文件读写可直接在内存页面中完成. 因此, 小文件读写吞吐率不能直接反映磁盘空间管理性能. 这里, 每次对象读写请求都小于 512 KB, 属于小文件读写, 因此不适合异步读写方法测试.

用读、写密集型 Trace 进行同步随机读写时, 各系统总吞吐率如图 6 所示. 图 6 中日志型文件系统 Ext3, JFS 在同步读写时, "*Write_Trace* 的吞吐率只有 *Read_Trace* 吞吐率的 30%, 而 ReiserFS 的这一数据则为 10% 左右, 日志文件系统由于日志开销, 平均吞吐率没有 XOBFS 高".

图 7、图 8 分别反映上述几种系统在长期使用

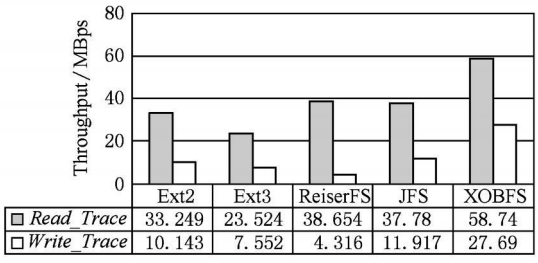


Fig. 6 Average throughput of synchronous access for 100 thousands operations.
图 6 10 万次操作时同步读写平均总吞吐率

过程中平均性能随时间的变化趋势. XOBFS 和 Ext2/3 均使用定长块分配, XOBFS 采用大粒度激进式分配. 通用文件系统在长期使用中, 经大量创建、删除操作后, 磁盘碎片逐渐增多, 文件和空闲空间的连续度均受到影响. 从而导致顺序读写减少, 随机读写增多, 性能不断退化. 因此, XOBFS 吞吐率显著优于其他几种文件系统.

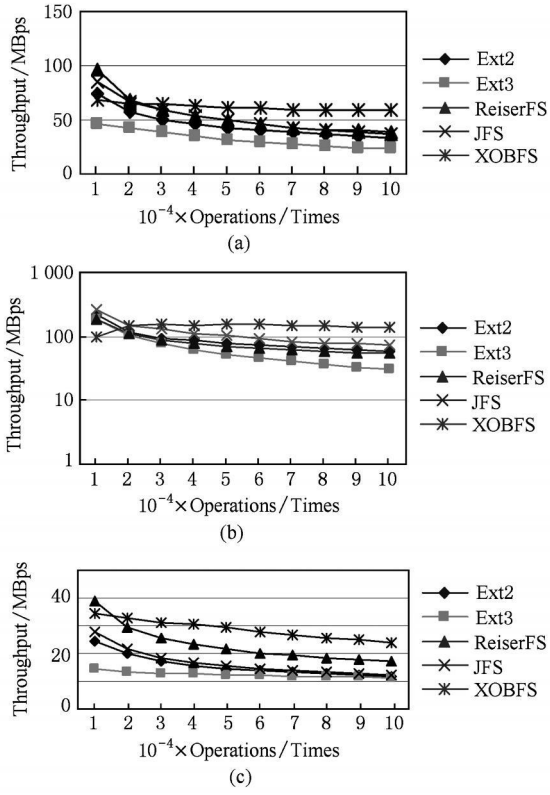


Fig. 7 Throughput related to operations as read-intensive accessing. (a) Trend analysis of the total throughput; (b) Trend analysis of throughput on reading; and (c) Trend analysis of throughput on writing.
图 7 读密集同步读写吞吐率与操作数关系. (a) 总吞吐率变化趋势; (b) 读吞吐率变化趋势; (c) 写吞吐率变化趋势

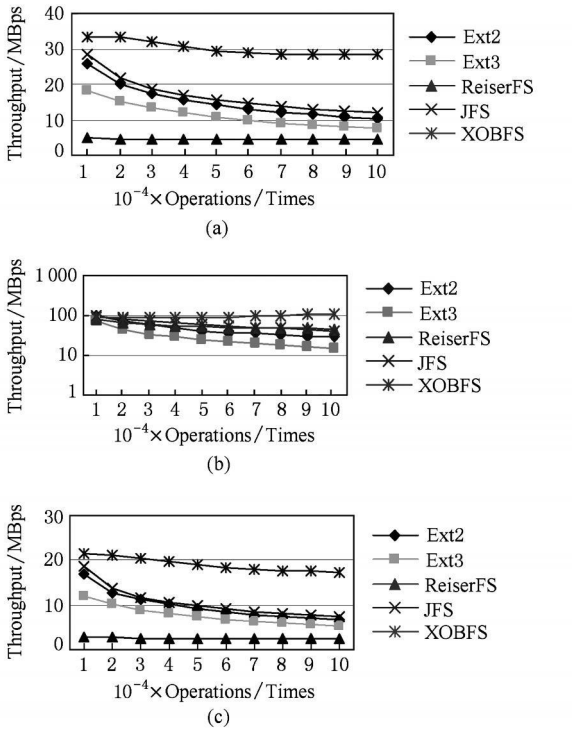


Fig. 8 Throughput related to operations as write-intensive accessing. (a) Trend analysis of the total throughput; (b) Trend analysis of throughput on reading; and (c) Trend analysis of throughput on writing.

图 8 写密集同步读写吞吐率与操作数关系. (a) 总吞吐率变化趋势; (b) 读吞吐率变化趋势; (c) 写吞吐率变化趋势

Ext2/3 由于分配粒度较小, 在长期读、写、删除操作后, 系统的空间连续度变差, 碎片明显增多. 读密集 Trace 测试结果中, Ext2 平均总吞吐率从 1 万次时的 73.693 MBps 下降到 10 万次时的 33.249 MBps; Ext3 平均总吞吐率从 1 万次时的 47.049 MBps 下降到 10 万次时的 23.524 MBps. 在写密集 Trace 测试结果中, Ext2 平均总吞吐率从 1 万次时的 25.929 MBps 下降到 10 万次时的 10.143 MBps; Ext3 平均总吞吐率从 1 万次时的 18.199 MBps 下降到 10 万次时的 7.552 MBps.

JFS 是一种提供日志的字节级文件系统. 该文件系统主要是为满足服务器的高吞吐量和可靠性需求而设计开发的. 当 JFS 日志文件系统保存一个日志时系统需要写大量数据. 因此, JFS 写数据速度慢. 在读密集 Trace 测试结果中, JFS 的 10 万次平均读吞吐率为 72.871 MBps, 写吞吐率为 12.049 MBps. 在写密集 Trace 测试结果中, JFS 的 10 万次平均读吞吐率为 42.386 MBps, 写吞吐率为 7.446 MBps.

ReiserFS 使用了特殊的、优化的平衡树来组织所有的文件系统数据. 读、写密集型 Trace 测试结果表明, ReiserFS 读性能比写性能至少高 4 倍.

XOBFS 侧重于对象在存储设备上的存放策略, 更关注的是优化磁盘存取这一块, 故这里主要采用的是同步读写模式.

XOBFS 较其他几种典型的文件系统性能高, 主要是由于 XOBFS 采用定长、大粒度的分配方式结合扩展 Hash 方法大大加速了大文件的处理. 节省了索引开销(地址块索引、Inode 索引), 提高了查找效率.

4 结 论

本文通过使用合成 Trace, 比较了 XOBFS 以及其他几种通用文件系统的长期使用性能. 测试结果表明使用位图方式分配空间, 分配粒度为定长块(512 KB) 的 XOBFS 文件系统在同步读写的情况下吞吐率显著优于通用文件系统. 使用大块分配策略, 使得磁盘已分配空间和空闲空间的连续度高. 在读写对象时, 磁头在一次寻道操作后顺序读写扇区, 提高了 I/O 吞吐率. 为了让对象存储系统具有可扩展性, XOBFS 对属性进行集中式管理. 与分散方式相比, 集中存放属性具有更高的吞吐率, 并且面对高可扩展的属性, 系统性能不会因为增加属性管理而退化.

参 考 文 献

[1] Gibson G A, Nagle D F, Courtright II W, et al. NASD scalable storage systems [C] //Proc of the USENIX 1999 Extreme Linux Workshop. Monterey: USENIX, 1999: 1-6

[2] Tang Hong, Gulbeden Aziz, Zhou Jingyu, et al. The Panasas active scale storage cluster-delivering scalable high bandwidth storage [C] // Proc of the 2004 ACM/IEEE Conf of Supercomputing. Los Alamitos, CA: IEEE Computer Society, 2004: 53-58

[3] T'so T Y, Tweedie S. Planned extensions to the Linux EXT2/EXT3 file system [C] //Proc of the Freenix Track: 2002 USENIX Annual Technical Conf. Monterey: USENIX, 2002: 235-244

[4] Factor M, Meth K, Naor D, et al. Object storage: The future building block for storage systems [C] //Proc of the 2nd Int IEEE Symp on Mass Storage Systems and Technologies. Piscataway, NJ: IEEE, 2005: 119-123

[5] Wang Feng. File system workload analysis for large scientific computing applications [C] // Proc of the 21st IEEE/12th NASA Goddard Conf on Mass Storage Systems and Technologies. Piscataway, NJ: IEEE, 2004: 67-83

[6] Fagin R, Nievergelt J, Pippenger N, et al. Extendible hashing: A fast access method for dynamic files[J]. ACM Trans on Database Systems, 1979, 4(3): 315-344

[7] Heller S. Extensible hashing. Dr. Dobb's Journal of Software Tools, 1989, 14(11): 66-70

[8] Wang Feng, Brandt Scott A, Miller Ethan L, et al. OBFS: A file system for object-based storage devices[C]//Proc of the 21st IEEE/12th NASA Goddard Conf on Mass Storage Systems and Technologies. Piscataway, NJ: IEEE, 2004: 101-118

[9] Xie Liming, Feng Dan, Qin Lingjun. Research and design the method of object attributes management on object based storage system[J]. Journal of Computer Research and Development, 2007, 44(Suppl I): 115-121 (in Chinese)
(谢黎明, 冯丹, 覃灵军. 对象存储系统中属性管理方法研究与实现[J]. 计算机研究与发展, 2007, 44(增刊I): 115-121)

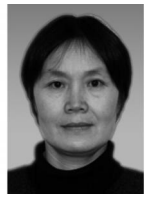
[10] O'Neil E J, O'Neil P E, Weikum G. The LRU-K page replacement algorithm for database buffering[C]//Proc of ACM SIGMOD'93. New York: ACM, 1993: 297-306

[11] Reiser H. Kernel korner: Trees in the Reiser 4 files system, Part I[J]. Linux Journal, 2002, (104): 8

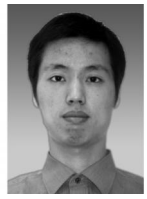
[12] Reiser H. Reiser 4, Part II: Designing trees that cache well[J]. Linux Journal, 2003, (109): 68-72

[13] Best S, Gordon D, Haddad I. Kernel korner: IBM's journaled file system[J]. Linux Journal 2003, 2003, (105): 9-18

[14] Qin Lingjun. Research on key technologies of object-based active storage[D]. Wuhan: Huazhong University of Science and Technology, 2006 (in Chinese)
(覃灵军. 基于对象的主动存储关键技术研究[D]. 武汉: 华中科技大学, 2006)



Liu Jingning, born in 1957. Aassociate professor. Her main research interests include computer architecture, computer-control and high-speed channel technology.
刘景宁, 1957 年生, 副教授, 主要研究方向为计算机存储及网络存储系统、计算机控制及通道接口技术.



Xie Liming, born in 1985. Master. His main research interests include file system design.
谢黎明, 1985 年生, 硕士, 主要研究方向为文件系统设计、计算机网络存储系统.

Feng Dan, born in 1970. Professor and PhD supervisor. Senior member of China Computer Federation. Her main research interests include computer architecture and information storage.
冯　丹, 1970 年生, 教授, 博士生导师, 中国计算机学会高级会员, 主要研究方向为计算机系统结构和信息存储.

Lü Man, born in 1984. Master. His main research interests include embedded system and file system design.
吕　满, 1984 年生, 硕士, 主要研究方向为嵌入式系统、文件系统设计.

Research Background

With the increasing demand of high I/O throughput, highly parallel data transfer and highly scalable storage system, especially in the supercomputing field, the traditional storage system has been difficult to meet the requirements. A new storage model, object-based storage system (OBS) came into being. In OBS systems, local storage space allocation is managed by intelligent OSD (object-based storage device). But, at present, the OSD is mainly managed by the common file system. However, when the common file system deals with the flat namespace, especially in the course of long-term use, the performance degenerates seriously.

This paper presents an extensible hashing object-based storage file system called XOBFS, which implements resource management in OSD. XOBFS uses fixed-length block allocation strategy and manages free block bitmap combination. Based on expanding hash manage object attribute, the same object attributes are stored in adjacent hash bucket. XOBFS applied in OSD has a lot of advantages, for example, the metadata has small scale, long-term performance is not degraded, attribute of the object is managed effectively and so on. Test results show that, big-object based XOBFS throughput rate is better than traditional file system. XOBFS provides an effective method for storage management in OSD.

This work is supported by the National Basic Research Program ("973" Program) of China under grant of No. 2004CB318201, the National High Technology Research and Development Program("863" Program) of China under grant No. 2009AA01A401, 2009AA01A402, the National Science Foundation of China No. 60703046, and Changjiang Innovative Group of Education of China No. IRT0725.