软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(8):1968−1981 [doi: 10.13328/j.cnki.jos.005202]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种基于小数据同步写的回写 I/O 调度器*

刘 星[1,2], 江 松[1,4], 王 洋[1], 范小朋[1], 须成忠[1,3]

[1](中国科学院 深圳先进技术研究院,广东 深圳 518055)

[2](中国科学院大学 深圳先进技术学院,广东 深圳 518055)

[3](Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202, USA)

[4](Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76010, USA)

通讯作者: 王洋, E-mail: yang.wang1@siat.ac.cn

摘 要: 小数据同步写普遍存在于各种计算机环境中,并且可以由计算机系统的不同层次软件产生,从底层操作系统一直到上层应用软件都可以生成小数据同步写请求.然而,操作系统的文件系统是以块作为最小逻辑可寻址单位,小数据写将会导致严重的写放大问题,使得系统的 I/O 性能大幅度降低.为了解决上述问题,提出了一种 I/O 调度器,并将其命名为 Hitchhike.该调度器可以识别小数据写,并通过对其他数据块中的数据进行压缩,将小数据嵌入到压缩出来的空间中,从而将小数据和该数据块一起写入到磁盘上,以异步回写的方式完成小数据的同步写,不仅有效缓解了磁盘的写放大问题,也极大地提高了小数据同步写的效率.基于 Linux 2.6.32 的 Deadline 调度器实现了 Hitchhike 原型系统,并利用 Filebench 基准测试来测试调度器在吞吐量、I/O 延迟等方面的性能.通过与传统 I/O 调度器的性能进行比较,可以发现,Hitchhike 调度器能够显著地提高小数据同步写的性能高达 48.6%.

关键词: 同步写;小数据写;I/O 调度器;写放大

中图法分类号: TP316

中文引用格式: 刘星,江松,王洋,范小朋,须成忠.一种基于小数据同步写的回写 I/O 调度器.软件学报,2017,28(8):1968−1981. http://www.jos.org.cn/1000-9825/5202.htm

英文引用格式: Liu X, Jiang S, Wang Y, Fan XP, Xu CZ. Writeback I/O scheduler based on small synchronous writes. Ruan Jian Xue Bao/Journal of Software, 2017,28(8):1968−1981 (in Chinese). http://www.jos.org.cn/1000-9825/5202.htm

Writeback I/O Scheduler Based on Small Synchronous Writes

LIU Xing[1,2], JIANG Song[1,4], WANG Yang[1], FAN Xiao-Peng[1], XU Cheng-Zhong[1,3]

[1](Shenzhen Institutes of Advanced Technology, The Chinese Academy of Sciences, Shenzhen 518055, China)

[2](Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences, Shenzhen 518055, China)

[3](Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202, USA)

[4](Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76010, USA)

* 基金项目: 国家重点基础研究计划(973)(2015CB352400); 国家自然科学基金(61572487, 61672513, 61572377, U1401258, 61550110250); 广东省自然科学基金(2015B010129011, 2016A030313183); 深圳市海外高层次人才创新创业专项(孔雀计划)(KQCX20140521115045446); 深圳市技术攻关项目(JSGG20150512145714248, JSGG20160229200957727)

Foundation item: National Basic Research Program of China (973) (2015CB352400); National Natural Science Foundation of China (61572487, 61672513, 61572377, U1401258, 61550110250); Science and Technology Planning Project of Guangdong Province (2015B010129011, 2016A030313183); Shenzhen Overseas High-Caliber Personnel Innovation Funds (KQCX20140521115045446), Research Program of Shenzhen (JSGG20150512145714248, JSGG20160229200957727)

收稿时间: 2016-08-07; 修改时间: 2016-09-21, 2016-11-11; 采用时间: 2016-12-01; jos 在线出版时间: 2017-01-12

CNKI 网络优先出版: 2017-01-12 10:36:00, http://www.cnki.net/kcms/detail/11.2560.TP.20170112.1036.004.html

Abstract: Small and synchronous writes are pervasive in various environments and manifest in various levels of software stack, ranging from device drivers to application software. Given a block interface, small write can cause serious write amplifications, which can substantially degrade the overall I/O performance. To address this issue, this paper presents a block I/O scheduler, named Hitchhike. Hitchhike is able to identify small writes, and embed them into other data blocks through data compression. With Hitchhike, a writeback buffer for small synchronous writes can be enabled, not only removing the write amplification, but also dramatically improving the performance of small synchronous writes. Hitchhike is implemented based on the Deadline I/O schedulers in Linux 2.6.32, and evaluated by running Filebench benchmark. Testing results show that compared to traditional approaches, Hitchhike can significantly improve the performance of synchronous small writes up to 48.6%.

Key words: synchronous write; small write; I/O scheduler; write amplification

随着云计算和大数据的出现,以数据为中心的数据密集型计算已经成为当前 IT 行业的一大重要趋势[1−3].相应的计算中涉及到的磁盘数据也逐渐增加,系统对磁盘的访问越来越频繁,存储设备成为系统性能瓶颈的可能性也越来越大.目前,主流的存储设备包括机械硬盘和固态硬盘.在物理上,物理磁盘的最小可寻址单元是扇区(sector),通常为 512 字.为了能够更高效地访问磁盘,在逻辑上文件系统通常会采用最小逻辑可寻址单元——块(block)或者页(page).块(block)是文件系统的一种抽象,只能基于块来访问文件系统.所以,块的大小不能小于扇区的大小,且只能是扇区的整数倍,通常为 4KB,如操作系统 Ubuntu 14.04 默认的块大小就为 4KB,这也远远大于处理器在内存中的最小可寻址单位——字节(byte).

小数据写是指欲写的数据大小远远小于块大小的写操作.例如,键值(vey-value,或者 KV)存储系统就是一个典型的小数据写系统.在键值存储系统(key-value store)或者是面向对象的存储系统(object-based storage)中,键值对可以非常小,并且可能需要被立即持久化.例如,Twitter 的每条微博被压缩后只有 362 字节,其中微博内容含有 46 个字节[4].当用户在 Twitter 上发表一条微博时,通过网络客户端向服务器发送一条写入请求,键值系统必须首先通过块接口在非易失性存储设备上存储该数据,然后才能向用户确认请求完成.因为用户发出的请求大多是由在线服务产生,如果是用批处理写请求,会使得很多写请求不得不被推迟,严重影响服务质量[5].为了满足基于块粒度的访问方式,数据总是需要以固定长度的块进行传输,即使只请求少量字节的数据,设备驱动程序也会从存储设备上写入一个完整块数据,因而小数据同步写可引起写放大效应.

文件系统的元数据也属于小数据,主要由描述文件系统结构特征的系统数据组成,每一条元数据的大小为几个或者几十个字节.虽然文件系统的元数据在整个存储空间中所占比例很小,约为 0.1%~1%[6],但是系统对元数据的访问占到了整个文件系统访问的 50%~80%之多[7].同时,在如 Facebook 等社交网站的实际应用中,数据小于 500 字节的键值对在内存中占据了将近 90%[8]的空间.因此,小数据写是普遍存在的.

此外,一些小数据的写还要求是同步写,例如文件系统的元数据写就是同步写.当系统发出一个同步写请求后,系统会被阻塞,直到数据被安全写入到磁盘后,系统才能继续执行;而对于异步写,系统是非阻塞的,当系统发出一个异步写请求后,会继续执行后续操作.异步写的读写效率高,但是如果计算机发生意外情况宕机,则系统存在数据丢失的风险.另外,当文件系统需要新写入一个数据块的时候,相关的数据块位图(bitmap)也需要被同步更新到磁盘中.如果数据被写入到磁盘中,而位图元数据没有被更新,那么在发生宕机时,系统读到的将是旧的位图信息,这将造成新写入的数据块也无法被识别.

基于上述事实,小数据可总结为写包含以下 3 个特点:(1) 小数据写的数据量小;(2) 小数据写普遍存在;(3) 小数据写通常需要同步写入磁盘.但是,为了满足文件系统以块的方式访问磁盘,小数据同步写可造成严重的写放大问题.

针对小数据写操作引起的写放大问题,在研究现有文件系统中数据与元数据组织架构的基础上,本文引入了一种全新的数据嵌入及复用技术.在虚拟硬盘驱动、I/O 调度器和文件系统等不同的软件层级上,针对写数据请求,能够识别其中的小数据写,并对其他数据块中数据进行压缩,为这个小数据创造可嵌入的空间.该方式不仅满足小数据的同步写需求,而且避免了可能的写放大问题;同时,以异步回写的方式提高了同步写的性能.本文提出的这种小数据嵌入技术是一种不受块接口限制的、可以将对小数据的更新在块设备上快速持久化的创新设计.

该技术虽然只针对于单机文件系统情况,但仍可部署在本地文件系统之上,应用于云存储下的分布式文件系统中,具有很强的可扩展性,为云存储集群优化小数据读写性能.本文的主要贡献包括以下几点:

(1) 提出一种新的调度算法,能够缓解小数据写的放大效应,实现更高效的小数据写操作;
(2) 提出了一种在 I/O 调度器中识别小数据写的方法,为优化小数据写提供可靠支持;
(3) 提出了一种磁盘的同步回写方法,通过该方式,可以将小数据写同步保存到磁盘,又可以缓存小数据写,等小数据写积累了一定量再一次写入到磁盘中.

本文第 1 节介绍相关工作.第 2 节介绍基于同步写的回写 I/O 调度器的设计与实现,主要从 Hitchhike 调度器的架构、小数据识别、数据持久化方式和快速恢复等方面进行描述.第 3 节对该 I/O 调度器进行性能测试,同时对实验结果进行对比分析.第 4 节对全文进行总结.

1 相关工作

所谓小数据,是指数据大小远远小于块大小的数据,但是以块粒度为访问单位的系统中,小数据的写入需要以块的方式进行,从而造成写放大效应,浪费 I/O 带宽.目前存在许多关于小数据写问题的研究,为了提高读写性能,对小数据的访问通常会使用回写模式(write-back),即当回写缓存中积累了一定数量的小数据写后再一次性存储到磁盘上[9],比如 LevelDB 中使用 MemTable 来缓存小数据写;同时,该思想也被运用于 Facebook 的开源键值数据库 RockDB 和 Apache 的开源项目 HBase 中.

在软件方面,美国加州大学伯克利分校(University of California at Berkeley)的 Rosenblum 等人发表了论文《The Design and Implement of a Log-Structured File System》[10],提出了一种日志式文件系统,该文件系统在内存中引入了一种存储单元——段(segment),磁盘数据的更新都缓存在这个存储单元中;然后在段满时,一次性将该存储单元中的数据写入到磁盘,以获取更高效的磁盘 I/O 效率.后来,该思想也被广泛运用于键值存储系统中,如 LevelDB,BigTable[11]等.虽然该思想有效地提高了小数据的写操作,但是它所运用的 LSM 树(log-structured merge tree)[12]依然存在写放大问题.为了解决这个问题,美国韦恩州立大学 Wu 等人提出了 LSM-trie 方案[9]来改善该问题.但是回写模式只适用于小数据异步写的情况,当用户程序是同步写时,日志式文件系统技术则不能改善小数据写放大问题.

此外,在对高效虚拟块设备的研究中,针对元数据,文献[13]中提出了另一种思路.类似于 QCOW2 等虚拟硬盘驱动,在 Selfie 中维护了一个内存映射表,用于将虚拟地址(VBA)映射到映像地址(IBA).虚拟块地址是指向虚拟机的块地址空间,映像地址指向的是硬盘上的逻辑地址.一个新数据块的同步写操作要求更新该表上的对应项,同时要求立即持久化,同样也会产生小数据同步写操作.Selfie 通过对数据块进行压缩,并将映射表的信息嵌入到该数据块中,从而通过一次 I/O 操作同时完成元数据和数据的持久化,减少了元数据的 I/O 次数.另外,一旦系统发生崩溃,可以通过已嵌入到数据块中的元数据信息恢复映射表.Selfie 主要针对的是由虚拟设备自身产生的小数据写入,没有处理其他层次的小数据同步写.

回写模式允许系统在内存中缓存若干小数据写,然后在适当的时候一起写入磁盘中,这种“积少成多”的方式存在一个问题:内存中的数据存在断电丢失数据的风险.为使缓存在内存中的数据不会因为断电而丢失,目前常用的解决方法是使用电容或电池保护的内存(NVRAM)[14−16]或者字节寻址的非易失性变相内存 PCM[17,18].在发生宕机、断电等意外情况后,内存仍可以在随后一段时间内保存小数据,然而这些只是针对那些拥有特殊硬件支持的高端系统的解决方案.此外,电池的可充电次数的限制、电池剩余容量的估计以及电池更换等都是需要解决的问题.随着内存与高速缓存越来越大,“电池的大小和缓存的位置已经成为新兴服务器设计中的明显障碍”[19].除了成本高昂以外,这种电池/电容和 PCM 内存都存在疲劳和磨坏的问题.再者,使用 NVRAM 或 PCM 意味着特殊内存中的数据和硬盘中的数据都要在系统崩溃后进行恢复,建立一个完整一致的映像.这样一来,若想仅把其中 1 个组件从失效的服务器移到新的服务上就不可能了.类似 NVRAM 的性能可以通过 Soft update 在没有额外硬件消耗的情况下实现[20],然而,这仅当存在异步写的假设时才成立.另外,也有人提出使用闪存来存储或者缓存元数据[21,22],但由于闪存也是块接口,小数据写入造成的写放大问题仍没有被解决.

目前,一些技术利用特殊的硬件可以为每个块提供少量的额外空间,目前还利用了一些特殊的硬件来为每个块提供少量的额外空间,这些额外空间原本是设计用来存放一些用于纠错或者从故障中恢复的信息.例如,现有的 SCSI 驱动器支持在每个 512 字节的扇区上附加一个 8 字节的“数据完整性区域(data integration field)”来存放纠错码[23].在 NAND 闪存设备中,也包含了类似的区域 OOB(out-of-band),它是每 512 个字节数据附加 16 字节,这些 OOB 区域可以用来存放 ECC 校验码等多种元数据.目前,有些研究已经使用了上述这些空间来存放元数据,例如,back-pointer 技术在数据块里保存一个反向指针指向相应的元数据,文件系统依赖数据和元数据的写入顺序来保证文件系统的一致性[24].出现故障之后,该方式还有助于重建文件系统的元数据[25].

然而,这些方法在解决小数据写问题上仍有一定的限制性,无法成为一个通用的方案来解决不同层面的小数据写问题:首先,这些额外的小空间需要特定硬件支持,例如,最常用的 SATA 磁盘并不提供这一接口;其次,这些空间不一定是空闲的,例如 SCSI 磁盘,当 DIF 或者 DIX 特性被使用时,8 字节的校验码区域就无法另做他用.除此之外,这些额外空间都是固定大小,只能存放小于该大小的数据.但通常情况下,元数据的大小是变化的,甚至不可预知的,所以此类额外设备方法也无法彻底解决小数据写的问题.

2 基于同步写的回写 I/O 调度器的设计与实现

当小数据同步写请求通过块接口时,文件系统为满足基于块粒度的访问方式,不得不访问一整个单位的块数据.而一个单位块的数据大小远远大于小数据,使得实际写入的数据仅占整个单位块的一小部分比例,从而造成了写放大,浪费了 I/O 带宽.针对此问题,本文提出了一种新型的 I/O 调度器——Hitchhike,通过该调度器来缓解因小数据同步写而造成的写放大问题.主要包括以下内容:(1) 提出了一种小数据识别的方法,能够在调度器中识别数据块中数据在前后两次操作中的异同;(2) 提出了两种模式的数据持久化方式;(3) 提出了一种应对意外情况的数据恢复方法.

2.1 Hitchhike调度器的架构

当文件系统向块设备发出一个读/写请求时,内核利用通用块层(generic block layer)处理来自系统中的对块设备的I/O请求.通常,每个I/O操作是访问磁盘上一组连续的块.但由于请求的数据可能不在同一个块或相邻的块中,所以通用块层会请求多次块 I/O 操作,每次块 I/O 操作都由内核中定义的“Bio 结构体”描述,它记录了待处理数据的缓存位置、本次 I/O 操作在块设备中的起始扇区等信息.

I/O 请求不会被立即响应,而是先提交到 I/O 调度器中,I/O 调度器位于通用块层和块设备之间,它能够对具有相邻磁盘扇区的请求进行合并操作,并根据预先定义的调度策略对待处理的 I/O 请求进行排序.排序是为了使请求的处理顺序和磁头移动方向相同,从而明显减少磁头寻道次数.在 Linux 内核中,最常用的两种调度器分别是最后期限(deadline)算法和完全公平队列(complete fairness queueing,简称 CFQ)算法.Deadline 调度算法不仅根据起始扇区进行排序,而且根据请求的最后期限进行排序,从而避免请求饿死.CFQ 调度算法是触发 I/O 请求的所有进程能够获得公平的磁盘 I/O 带宽.不同的 I/O 调度器的这种方式对大数据的写入很适合,也不容易造成 I/O 浪费,但是对小数据同步写却造成写放大问题,并没有解决小数据同步访问和基于块粒度访问的冲突.

本文研究的是一种基于同步写的回写 I/O 调度器,该调度器与传统调度器最大的一个不同在于,它针对小数据同步写实现了一种回写机制,但却保证不会因意外情况而造成数据丢失.因此,在保证数据块能够安全恢复的前提下,Hitchhike 克服了传统上通常为同步写而采用的写透策略(write-through)所带来的低效性,不仅可以缓解小数据写放大问题,而且可以回写的方式达到异步的 I/O 性能.Hitchhike 的架构如图 1 所示.

从图中可以看出,Hitchhike 调度器不仅具有传统调度器所具有的合并和排序功能,而且加入一个回写缓存空间(write back buffer),它负责管理所有被缓存的数据块;同时还加入了小数据识别模块,能够对小数据进行识别.因此,Hitchhike 具有以下 5 个特征.

(1) Hitchhike 调度器维护了一个回写缓存空间以及能够识别写请求是否为小数据写.判别写请求是否为小数据写的关键之一在于能够较快、较准确地计算出操作前后数据块中数据的变化,如少量数据的追加或者修改.为此,需要为 I/O 调度器分配一块大小适合的回写缓存器空间(write back buffer)来缓存

被频繁追加或修改的数据块.在回写存储空间被用满的情况下,Hitchhike 调度器会使用置换算法,置换出部分数据块.

(2) 通过遍历调度器的请求队列,Hitchhike 调度器寻找适合被压缩的数据块进行压缩,以此减小实际数据在块中占用的空间大小,并将小数据嵌入到块的剩余空间中.I/O 调度程序的主要任务是为待处理的块 I/O 请求分配磁盘 I/O 资源,具体来说,这种资源的分配是通过将物理地址相邻的块 I/O 请求进行合并,并对块 I/O 请求进行重新排序来决定块 I/O 请求的执行次序.在不降低磁盘寻址操作的情况下,Hitchhike 调度器会选择一个待处理的数据块,通过数据压缩算法对其进行压缩.被选中的数据块需要能够被压缩,并且具有一定的压缩比,使得压缩后的数据块可以腾出足够大小的空间来寄存小数据.

(3) Hitchhike 调度器会根据压缩后剩余的空间大小选择一个小数据,并将其嵌入到被压缩的数据块中,然后和其他数据块一起存储到磁盘上.选择适合的小数据也是本文研究的一个关键点,被压缩的小数据需要考虑的因素包括:能否被嵌入到压缩后的数据块、用于计算小数据的数据块是否频繁有小数据写操作等.

(4) 当遇到意外情况时,系统通过扫描嵌入了小数据的数据块进行快速恢复系统.在出现意外情况时,为了保持系统的一致性,系统需要保证已被写入的数据不能丢失.系统扫描出嵌入了小数据的数据块,并能够通过嵌入的小数据来恢复意外情况之前的状态.

(5) 在系统正常运行期间,系统不需要访问嵌入了小数据的数据块.由于回写缓冲空间的存在,被嵌入了小数据的数据块同样会被缓存到回写缓存空间中.所以,Hitchhike 调度器需要识别哪些数据块已经缓存在回写缓存空间中.当系统需要访问该数据块时,能够直接访问回写存储空间,而不必访问磁盘空间,从而实现了利用 Hitchhike 的回写缓存达到小数据同步写的高性能.

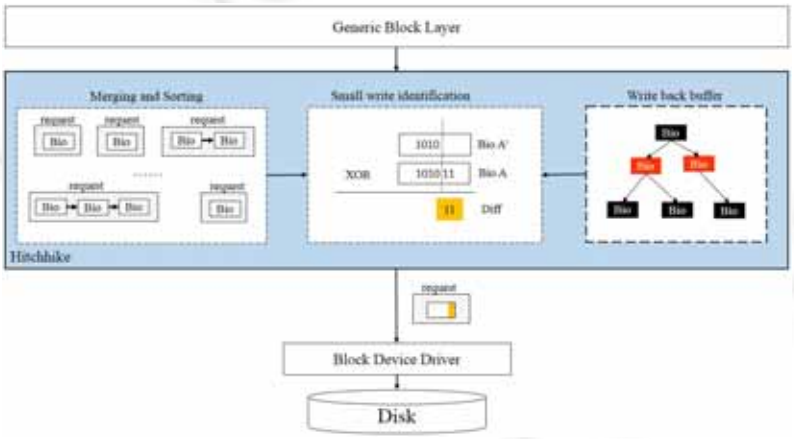

Fig.1 Hitchhike I/O scheduler architecture

图 1 Hitchhike I/O 调度器的架构图

2.2 回写缓存空间(**write back buffer**)

在内核中,通用块层用来描述块 I/O 操作的核心数据结构是 Struct bio.每个 Bio 结构体都包含了块 I/O 操作的起始扇区以及 I/O 操作相关的内存区.从通用块层向 I/O 调度器提交的 I/O 请求中会包含若干 Bio 结构体.回写缓存空间(Write back buffer)主要负责缓存一些被频繁访问的 Bio,这些 Bio 将会被用于后续的小数据识别.

为了能够在回写缓存空间中快速查找到一个相应的 Bio,回写缓存空间(write back buffer)通过使用红黑树来组织管理所有缓存的 Bio,该红黑树是根据 Bio 的起始扇区号构建的.回写缓存空间不仅会缓存用于小数据识别的数据块,而且需要缓存那些可潜在用来嵌入小数据的数据块.当回写缓冲空间(write back buffer)满了时,需要将聚集小数据的数据块和其相应的所有宿主数据块一起写入到磁盘.所以在回写缓存空间中,会保存两棵红

黑树:一棵用于缓存用于小数据识别的数据块,一棵用于缓存嵌入过小数据的数据块.如图 2 所示.

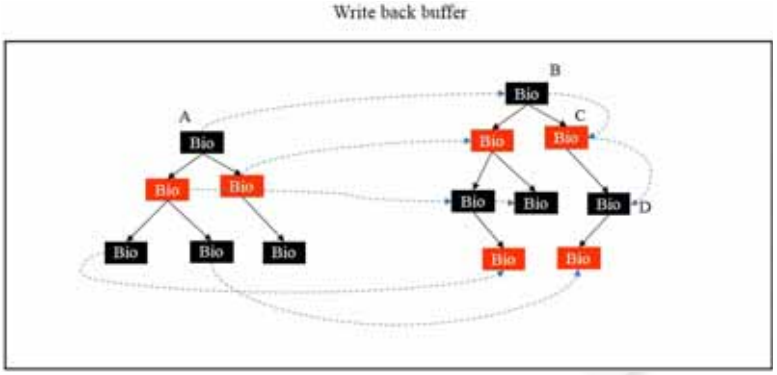

Fig.2 Writeback buffer

图 2 回写缓存空间

在图 2 中,左边的红黑树缓存的是用于小数据识别的数据块,右边的红黑树缓存的是用于嵌入过小数据的数据块.图中虚线表示的是由左边数据块计算得到的若干次小数据写嵌入到了右边的数据块中,例如,某进程向数据块 *A* 中进行了 3 次小数据同步写,这 3 次小数据写分别被嵌入到数据块 *B*,*C*,*D* 中.

当回写缓冲空间满了时,Hitchhike 调度器采用 LRU(least recently used)算法将一些数据块从回写缓冲空间中置换出来.该 LRU 算法是通过使用两条链表实现的:活跃链(active list)和不活跃链(inactive list),前者链接的数据块是最近计算过小数据写的数据块,后者则表示最近没有收到过小数据写请求.当回写空间满了时,Hitchhike 调度器会首先将不活跃链中的数据块置换出去.例如,假设 *A* 满足条件,在将 *A* 置换出去的同时,*B*,*C* 和 *D* 也将被置换出去,并且覆盖先前嵌入了 *A* 的小数据的对应块 *B*′,*C*′和 *D*′.这样操作的原理在于:3 次小数据的写集成在 *A* 中写入磁盘,嵌入每次小数据写的 *B*′,*C*′和 *D*′则不再有用了,覆写之后,系统仍然处于一致状态,从而实现了小数据同步写的回写机制,但代价是小数据的宿主块多了一次延迟覆盖写.具体的管理流程如图 3 所示.

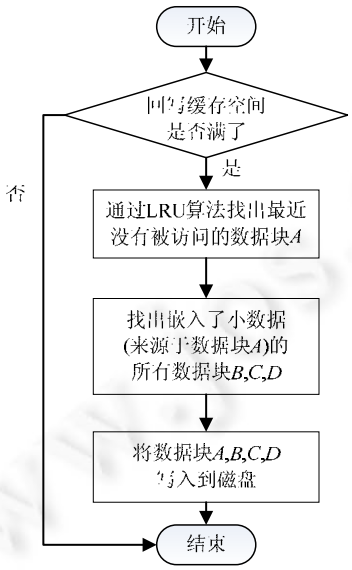

Fig.3 Replacement algorithm of writeback buffer

图 3 回写缓存空间的置换算法

2.3 小数据识别

为了缓解因小数据同步写而造成的写放大问题,系统需要解决的第 1 个问题就是小数据识别.如果在文件系统上解决该问题,可以比较容易地识别出小数据写.但我们目前采用的是面向 I/O 调度器的设计,为了对文件系统透明,小数据的识别只能在 I/O 调度器中完成.

当有一个写请求 Bio 从通用块层提交到 Hitchhike 调度器后,Hitchhike 会根据该 Bio 结构体中 I/O 操作的起始扇区号,以 $O(\log n)$的速度遍历回写缓存空间(write back buffer),查找具有相同起始扇区号的 Bio,其中,n 为缓存空间中 Bio 的个数.例如,在写请求中有一个 Bio,并将该 Bio 命名为 A,Bio A 的起始扇区号为 32.如果在回写缓存空间中已经缓存了一个起始扇区号为 32 的 Bio,并将该 Bio 命名为 A',通过对 A 和 A'中的数据进行异或运算(XOR),运算的结果就是块 A 和块 A'之间数据的差异,该差异就是本次写请求实际需要写入的数据.具体过程如图 4 所示.

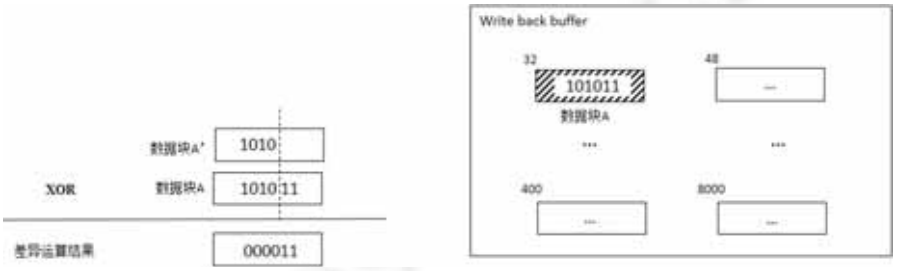

Fig.4 Exclusive OR

图 4 异或运算

在图 4 中,A 中的数据为 101011,A'中的数据为 1010,A 和 A'中的数据进行异或运算结果为 000011.差异部分越少,实际需要被写入的数据就越小;反之亦然.如果请求为小数据写,进行异或运算后得到的差异部分就越少,然后使用 LZ4 压缩算法对异或运算的结果进行压缩.LZ4 是一种无损压缩算法,它具有很高的压缩速度.压缩后的异或运算结果的大小可用来判断该写请求是否为小数据写.

系统可能会对同一个数据块进行多次小数据同步写,例如,某程序以同步写的方式向一个日志文件中追加一些新的记录,那么文件的最后一个数据块会进行多次小数据的同步写.为了区分异或运算的先后次序,压缩后的差异运算结果需要使用时间戳进行标识.此外,还需要记录该 Bio 的起始扇区号,以区分不同数据块的差异运算结果.经过异或运算之后,Bio A 将会替换回写缓存空间中的 Bio A',Bio A 将会用于下次异或运算.

当 I/O 调度器需要向磁盘写入另一个数据块 B 时,如果有被压缩过的差异运算结果,那么对数据块 B 中的数据进行压缩.压缩后的数据大小会变小,从而可以腾出部分空间,Hitchhike 调度器会将选择一个适当大小的被压缩过的差异运算结果嵌入到该部分空间中,然后将这个数据块写入到磁盘中;同时,将数据块 B 也缓存到回写缓存空间(write back buffer)中.具体过程如图 5 所示.小数据识别的流程如图 6 所示.

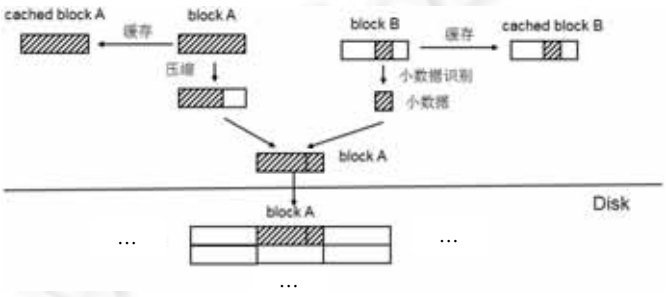

Fig.5 Small data embedding

图 5 小数据嵌入过程

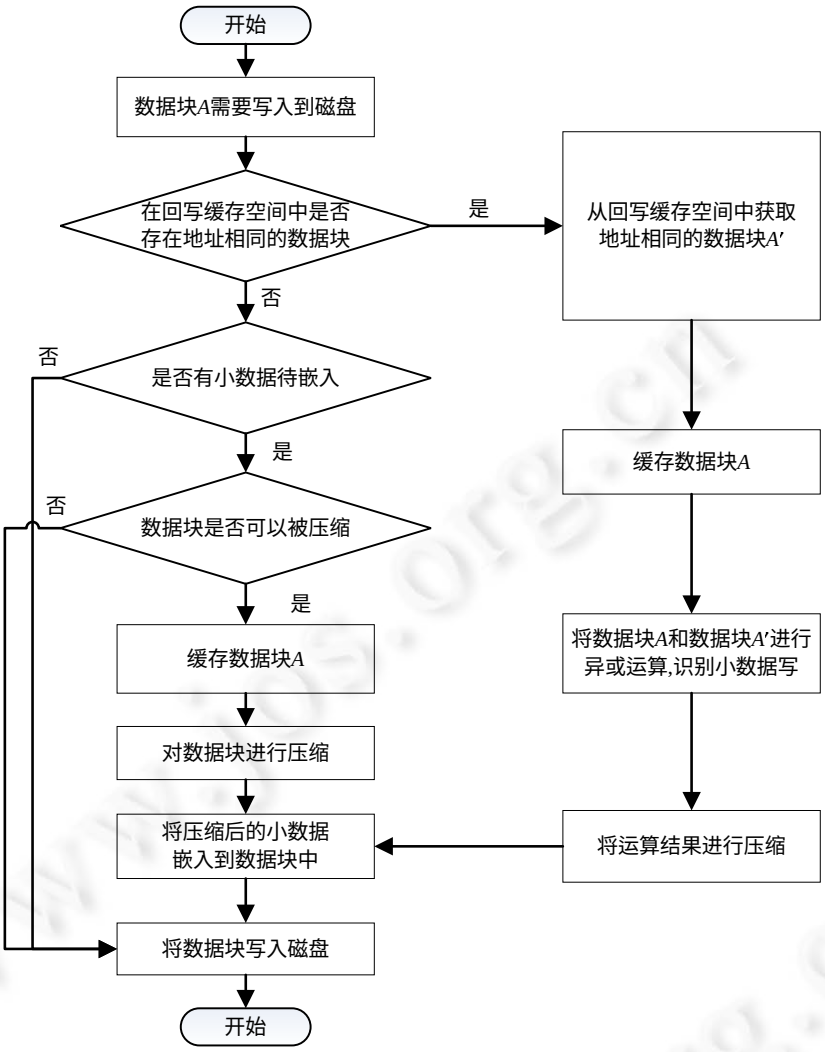

Fig.6 Process of small writes

图 6 小数据写的流程

2.4 数据持久化方式

对于写请求,传统 I/O 调度器首先一般是对待处理的写请求进行排序,然后从排序后的写请求队列中选择一个写请求,最后将整个写请求中的整个数据块写入到磁盘上.而 Hitchhike 调度器则会先识别写请求是否为小数据写,然后将小数据写嵌入到另一个将要被写入磁盘的数据块中.所以,Hitchhike 调度器将会采用两种模式在磁盘上持久化数据.

- 整块模式(full-block mode).这种模式也是传统 I/O 调度器使用的数据持久化方式,即,将整个数据块写入写入该块的磁盘地址中.
- 部分块模式(partial-block mode).即只将数据块中追加或更新的部分嵌入到一个宿主数据块中,并与之一同写入到磁盘.

对于第 2 种持久化方式,同一块的多次追加或者更新操作会被分散到多个宿主数据块上.另外,考虑到效率问题,Hitchhike 不会单独地跟踪每个更新的确切地址.因此,这些被分散到多个块上的更新不能用服务读请求.取而代之的是被缓存在回写缓存空间中的数据块来服务读请求.

由于回写缓冲空间的大小是有限制的,当回写缓冲区达到上限之后,会选择性地从回写缓冲空间中置换出一些数据块.此时,对于那些以部分块模式被持久化的数据块,嵌入在它们中的更新最终会以整块模式被写回到磁盘上.而这里的延迟写就是 Hitchhike 调度器能够在回写技术中实现同步写的关键.因为在整个块回写操作之前,对同一个数据块上的多次小数据写可以被积累,利用批量写达到了增加性能的目的.

2.5 快速恢复

在部分块模式持久化方式中,会将数据块中追加或更新的部分嵌入到某一个宿主数据块中,直到该数据块从回写缓冲空间中置换出来,才会以整块模式持久化到磁盘上.当系统发生意外情况出现宕机时,回写缓冲区中的数据块已经以部分块模式的模式将数据持久化到磁盘,还有一些小数据写寄存在其他宿主数据块上.因此在发生意外情况时,系统可以恢复到原本的样子.

最简单的方式是扫描整个整个磁盘,将嵌入了小数据的数据块读出来.然后取出小数据部分,并从磁盘中读取计算该小数据写的数据块,根据小数据写的时间戳,依次和该数据块进行异或运算,最后算出多次小数据同步写之后的数据.

但是全盘扫描将会很耗时间,效率比较低.由于磁盘上数据布局的优化,程序对磁盘的访问存在空间局部性,在一段时间内,程序对磁盘的访问会集中在某块区域.根据这一特性,对磁盘进行更粗粒度的逻辑划分,构建一个逻辑结构.将磁盘划分为若干存储区(zone),每个存储区(zone)的大小是固定的,并且远远大于一个块的大小.根据存储区中数据块的持久化方式,将存储区分成了 *Z*-存储区(*Z*-zone)和 *N*-存储区(*N*-zone).*Z*-存储区代表的在该区域中存在以部分块模式进行持久化的数据块,而 *N*-存储区是指该区域中所有数据块的持久化方式均以整块模式完成,具体如图 7 所示.

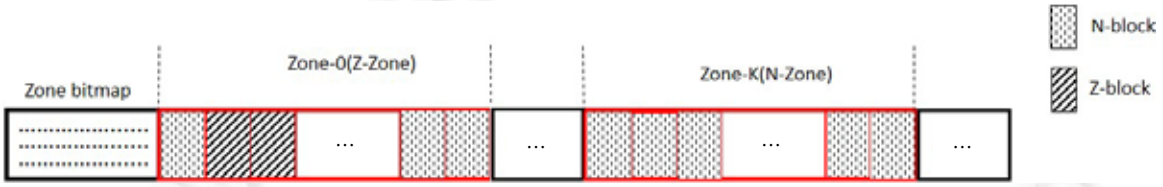

Fig.7 Disk image format

图 7 磁盘镜像格式

在图 7 中,*Z*-block 代表的是以部分块模式进行持久化的数据块,*N*-block 代表是以整块模式进行持久化的数据块.根据扇区号,磁盘被分为若干个存储区,每个存储区的类型只能是 *Z*-存储区或者 *N*-存储区,并且每种类型的存储区可能被分散到磁盘不用区域,因此,系统还需要在磁盘上维护一张存储区位图(zone bitmap),该位图将会标识每个 Zone 的类型.在最开始的时候,系统中没有以部分块模式进行持久化的数据块,此时所有存储区都为 *N*-存储区.当且仅当系统向 *N*-存储区中写入第 1 个 *Z*-block 时,需要更新存储区位图,将该存储区的类型从 *N*-存储区变为 *Z*-存储区.当 *Z*-存储区中所有的数据块都已整块模式进行了持久化之后,才会将该存储区的类型更改为 *N*-存储区.

当系统发生意外情况而进行重启时,系统首先读取存储区位图,确定磁盘上的哪些存储区是 *N* 存储区,然后再扫描 *N* 存储区中的数据块,将 *N*-block 数据块找出,然后进行数据的恢复.从而在系统恢复时减小系统扫描的磁盘区域,从而加快系统恢复.例如,当系统发生宕机时,磁盘划分如图 7 所示,当系统重启时,通过扫描存储区位图(zone bitmap),发现存储区 0 是属于 *Z*-存储区,从而系统只需扫描存储区 0 中的数据块.

3 实 验

本节对新提出的一种基于同步写的回写 IO 调度器进行验证,实验证明了该调度器的可行性.同时,把该调度器与传统调度器——Deadline 和 CFQ 进行对比,证明了该策略对小数据同步写有较好的性能提升,能够降低系统延迟,提高系统吞吐率.

3.1 实验环境

本实验中,我们是基于 linux 内核 2.6.32 版本中的 Deadline 调度器上实现了 Hitchhike 调度器原型,并且所有实现都是在一台主机上完成的,主机具体配置见表 1.

Table 1 System configuration used in the experiments

表 1 实验的系统配置

硬件	配置
RAM	2GB
CPU	Inter core i3-3240 CPU @ 3.40GH×4
Disk	500G
Operate system	Ubuntu 14.04TLS,32bit
内核	Linux 2.6.32

在实验中,我们使用的文件系统块大小为 4KB,4KB 也是文件系统块大小常用的一个值.本实验对数据块以及异或运算结果的压缩使用的是 LZ4(extremely fast compression algorithm)压缩算法,该压缩算法有较快的压缩速度和较好的压缩比.此外,本实验使用的基准测试是 Filebench.Filebench 是一款文件系统性能的自动化测试工具,它通过快速模拟真实应用服务器的负载来测试文件系统的性能,能够自动化生成负载.为了模拟小数据同步写,本实验修改 filebench 中的 filemicro_writefsync.f 文件,通过该文件,可以模拟一个小数据同步写的工作负载.该负载主要完成的内容为:循环地向一个文件中写入指定大小的小数据,每次追加后都使用 *fsync*()函数将数据同步到磁盘上,从而模拟小数据同步写环境.

为了测试 Hitchhike 调度器在不同大小的小数据同步写下的性能,本实验写入数据的大小为 64B~2 048B.每次实验运行时间为 5 分钟.为了对比 Hitchhike 调度器的小数据同步写性能,本实验还测试了 Deadline 调度器和 CFQ 调度器的小数据同步写性能.Deadline 调度器和 CFQ 调度器是使用的最为广泛的两种调度器.由于采用电梯策略,I/O 调度器会优先把靠近磁头移动方向最近的数据块写入到磁盘上,然而这会造成远离磁头的请求长时间得不到处理的现象.为了避免请求饿死,Deadline 调度器给每个请求一定的处理期限,该期限可以保证那些远离磁头移动方向的数据块可以得到及时处理.而 CFQ 调度器为了确保 I/O 带宽的公平分配,借此实现完全公平的调度算法.

3.2 实验分析

在该实验中,我们通过改变小数据同步写请求的数据大小来测试不同大小的小数据同步写在 I/O 调度器中的写,为确保每次小数据写请求能够同步到磁盘上,每次小数据写之后都执行 *fsync*(),*fsync* 函数可以确保修改过的数据块立即写入到磁盘上,并且等待磁盘写操作结束才会返回.

图 8 显示的是不同大小的小数据同步写在不同调度器中的写延迟,横坐标表示的是每次写入数据的大小,纵坐标表示的是每次同步写的写延迟.

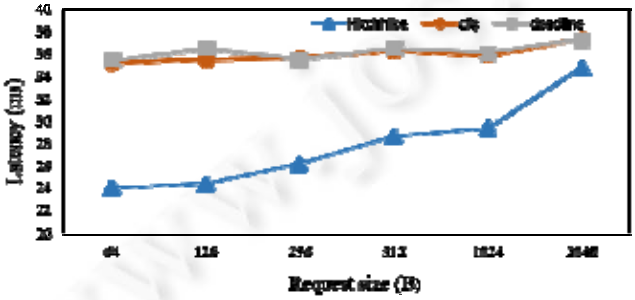

Fig.8 Latency changes of small synchronous writes under different request sizes

图 8 在不同大小的请求下,小数据同步写延迟的变化

从图 8 可知,不同大小的小数据同步写在 CFQ 和 Deadline 两种调度器上的写延迟基本相似,都固定在一个较为稳定的值.相比之下,在内核使用 Hitchhike 调度器时,不同大小的小数据同步写表现出的写延迟不相同,并且延迟均低于 CFQ 和 Deadline.从图中可以看到,随着写入数据的增大,使用 Hitchhike 调度表现出的写延迟逐渐升高,并且逐步接近于使用传统调度器的写延迟.

写延迟的降低,主要归功于 Hitchhike 调度器使用了部分块模式的持久化方式.该方式可以将小数据同步写嵌入到了其他数据块中,只需花费 1 次 I/O 操作即可实现对小数据和其他数据块的存储.从 I/O 次数方面考虑,部分块模式的持久化方式暂时节约 I/O 操作,从而使得平均写延迟降低.而且节约的 I/O 操作越多,写延迟降低得就会越多.表 2 显示了小数据写识别率.

Table 2 Recognition ratio of small writes

表 2 小数据写识别率

数据大小(B)	64	128	256	512	1 024	2 048
命中率(%)	98.5	97	93.8	87.6	75.1	50

该命中率的计算公式如下:

$$命中率=\frac{小数据写的次数}{完成的I/O操作总次数}.$$

命中率越高,表明小数据写所占比例越高,从而使用部分块模式持久化的次数也就越多,可以节约的 I/O 操作也相应地越多;相应的,写延迟也就降低了.从表 2 中可以看出:当每次写入的数据大小为 64B 时,命中率最高,达到了 98.5%,此时的写延迟也是最低的;随着数据大小的增大,命中率逐渐降低,写延迟也在逐渐增大.

图 9 显示的是不同大小的小数据同步写在不同调度器中的吞吐量.而实验中的吞吐量并不是很大,其原因主要有两个方面:(1) 本次实验模拟的是小数据写,所以基准测试生成的负载不高;(2) 每次向文件中写入小数据后,通过 *fsyn*()将小数据同步到磁盘上,而同步写的效率较低.从图 9 中可以看出:无论每次同步写入的数据多大,使用 Deadline 调度器和 CFQ 调度器时,系统的吞吐率都维持在 55ops/s.当系统使用 Hitchhike 调度器时,系统的吞吐率高于前两者.在同步写为 64B 时,使用 Hitchhike 的 I/O 调度器,系统的吞吐量比使用 CFQ 和 Deadline 的高出 48.6%;并且随着每次同步写数据量的增大,使用了 Hitchhike 的 I/O 调度器的系统的吞吐量在逐渐接近于使用了 CFQ 和 Deadline 的系统吞吐量.这个趋势也和写延迟相似.

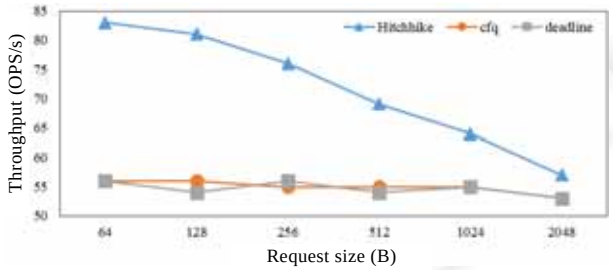

Fig.9 Throughput changes of small synchronous writes under different request sizes

图 9 在不同大小的请求下,系统吞吐量的变化

Hitchhike 调度器使用了压缩算法 LZ4 对数据块中的数据进行压缩.LZ4 是一种快速的无损压缩算法.使用压缩算法将会对系统的 CPU 造成额外开销.图 10 显示了系统处于内核状态下,CPU 使用的时间百分比.在图 10 中,横坐标表示时间,单位为 s;纵坐标表示的是 CPU 处于系统内核模式下的百分比.从图中可以看出:当系统使用 Hitchhike 调度器时,内核态下的 CPU 使用的时间百分比会稍高于使用 Deadline 调度器时的 CPU 使用时间百分比.其中,在系统使用 Deadline 调度器时,内核态下的 CPU 使用的时间百分比平均为 1.67%;在系统使用 Hitchhike 调度器时,内核态下的 CPU 使用的时间百分比平均为 2.03%,只比前者高出 0.36%.虽然 Hitchhike 调度器使用了压缩算法,这将会给内核态下的 CPU 带来一定的额外开销,但这个额外开销消耗的 CPU 资源较小.

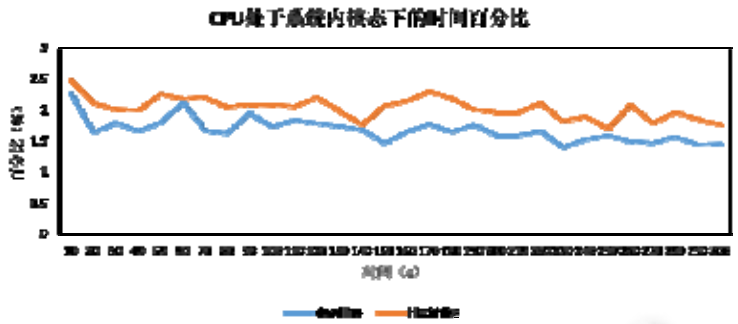

Fig.10 Percentage of time when CPU is running in system kernel space

图 10 CPU 处于系统内核态下的时间百分比

Hitchhike 调度器除了使用压缩算法 LZ4 对数据块中的数据进行压缩以外,还维护了一个回写缓存空间(write back buffer).无论是压缩算法还是回写缓存空间,都会对系统内存造成一定的额外开销.图 11 显示了在分别使用 Hitchhike 调度器和 Deadline 调度器时,系统内存的使用情况.在图 11 中,纵坐标表示已使用的内存(不包括交换分区(swap))和总内存的比值;横坐标表示时间,单位为 s.为了识别小数据,Hitchhike 调度器会在回写空间中缓存数据块.随着时间的推移,缓存的数据块越来越多,消耗的内存也就越来越多.所以随着时间的推移,系统使用的内存呈线性增长.但是回写缓存空间可存储的数据块数目是有限制的,当回写缓存空间的可用容量使用完之后,系统会使用最近最少使用的(LRU)算法置换出一些数据块.从图中可以看出,在 210s 之后,系统已使用的内存趋于稳定值,基本维持在 41%左右.使用 Deadline 调度器时,内存平均使用率为 35.98%.相比之下,使用 Hitchhike 调度器时,内存平均使用了 39.39%,稍低于前者,但是并没有给系统造成很大的系统负担.

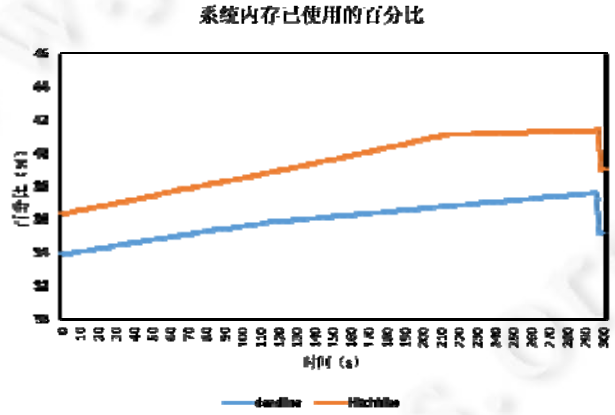

Fig.11 System memory usage

图 11 系统内存使用情况

4 小 结

本文提出了一种基于同步写的回写 I/O 调度器——Hitchhike.该调度器管理了一个回写缓存空间.该空间可以用于缓存已写入的数据块,通过对同一物理地址的数据块中的数据进行比较来识别小数据写请求.通过对其他数据块中的数据进行压缩,节省出数据所占空间,然后将小数据嵌入到节省出的空间中,从而可以在一次 I/O 操作中同时实现对其他数据块和小数据的存储.本文是基于 Linux 内核 2.6.32 版本中的 Deadline 调度器实现的,并且通过运行基准测试程序来验证 Hitchhike 调度器对小数据同步写的性能.通过和 Deadline 调度器和 CFQ 调度器比较可知,Hitchhike 调度器可以降低小数据写延迟,增大系统对小数据写的吞吐量.此外,使用

Hitchhike 调度器对系统资源的额外开销也较小.

References:

[1] Bryant RE. Data-Intensive supercomputing: The case for DISC. Pdl.cmu.edu, 2007.

[2] Hey T, Trefethen A. The data deluge: An e-science perspective. In: Proc. of the Grid Computing: Making the Global Infrastructure a Reality. 2003. 809−824. [doi: 10.1002/0470867167.ch36]

[3] Szalay AS, Kunszt PZ, Thakar A, Gray J, Slutz D, Brunner RJ. Designing and mining multi-terabyte astronomy archives: The Sloan digital sky survey. ACM SIGMOD Record, 1999,29(2):451−462. [doi: 10.1145/342009.335439]

[4] How much text verus metadata is in a tweet? 2011. http://goo.gl/EBFIFs

[5] Decandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 2007,41(6):205−220. [doi: 10.1145/1323293.1294281]

[6] Miller EL, Greenan K. Reliable and efficient metadata storage and indexing using NVRAM. 2008.

[7] Ousterhout JK, Costa HD, Harrison D, Kunze JA, Kupfer M, Thompson JG. A trace-driven analysis of the UNIX 4.2BSD file system. ACM SIGOPS Operating Systems Review, 1985,19:15−24. [doi: 10.1145/323647.323631]

[8] Atikoglu B, Xu Y, Frachtenberg E, Jiang S, Palaczny M. Workload analysis of a large-scale key-value store. ACM SIGMETRICS Performance Evaluation Review, 2012,40(1):53−64. [doi: 10.1145/2318857.2254766]

[9] Wu X, Xu Y, Shao Z, Jiang S. LSM-Trie: An LSM-tree-based ultra-large key-value store for small data. In: Proc. of the 2015 USENIX Annual Technical Conf. USENIX Association, 2015. 71−82.

[10] Rosenblum M, Ousterhout JK. The design and implementation of a log-structured file system. In: Proc. of the ACM SIGOPS Operating Systems Review. ACM Press, 1996. 1−15. [doi: 10.1145/121132.121137]

[11] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems, 2008,26(2):205−218. [doi: 10.1145/1365815.1365816]

[12] O'Neil P, Cheng E, Gawlick D, O'Neil E. The log-structured merge-tree (LSM-tree). Acta Informatica, 1996,33(4):351−385. [doi: 10. 1007/s002360050048]

[13] Wu X, Shao Z, Jiang S. Selfie: Co-Locating metadata and data to enable fast virtual block devices. In: Proc. of the ACM Int'l Systems and Storage Conf. ACM Press, 2015. 1−11. [doi: 10.1145/2757667.2757676]

[14] Chen PM, Ng WT, Chandra S, Aycock C, Rajamani G, Lowell D. The Rio file cache: Surviving operating system crashes. ACM Sigplan Notices, 1996,31(9):74−83. [doi: 10.1145/248209.237154]

[15] Wang Y, Davis K, Xu Y, Jiang S. iHarmonizer: Improving the disk efficiency of I/O-intensive multithreaded codes. In: Proc. of the IEEE Int'l Parallel and Distributed Processing Symp. IEEE Computer Society, 2012. 921−932. [doi: 10.1109/IPDPS.2012.87]

[16] Srinivasan K, Bisson T, Goodson G, Voruganti K. iDedup: Latency-aware, inline data deduplication for primary storage. In: Proc. of the USENIX Conf. on File and Storage Technologies. USENIX Association, 2012.

[17] Chen J, Wei Q, Chen C, Wu L. FSMAC: A file system metadata accelerator with non-volatile memory. In: Proc. of 2013 IEEE the 29th Symp. on Mass Storage Systems and Technologies (MSST). IEEE, 2013. 1−11. [doi: 10.1109/MSST.2013.6558440]

[18] Condit J, Nightingale EB, Frost C, Ipek E, Lee B, Burger D, Coetzee D. Better I/O through byte-addressable, persistent memory. In: Proc. of the ACM Symp. on Operating Systems Principles (SOSP 2009). 2009. 133−146. [doi: 10.1145/1629575.1629589]

[19] Non-Volatile cache for host-based raid controls. 2011. http://www.dell.com/downloads/global/products/pvaul/en/NV-Cache-for-Host-Based-RAID-Controllers.pdf

[20] Ganger GR, Mckusick MK, Soules CAN, Patt YN. Soft updates: A solution to the metadata update problem in file systems. ACM Trans. on Computer Systems, 2000,18(2):127−153. [doi: 10.1145/350853.350863]

[21] Chen F, Koufaty DA, Zhang X. Hystor: Making the best use of solid state drives in high performance storage systems. In: Proc. of the Int'l Conf. on Supercomputing. 2011. 22−32. [doi: 10.1145/1995896.1995902]

[22] Huang H, Hung W, Shin KG. FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption. ACM SIGOPS Operating Systems Review, 2005,39(5):263−276. [doi: 10.1145/1095809.1095836]

[23] Wallace G, Douglis F, Qian H, Shilane P, Smaldone S, Chamness M, Hsu W. Characteristics of backup workloads in production systems. In: Proc. of the USENIX Conf. on File and Storage Technologies. USENIX Association, 2012.

[24] Chidambaram V, Sharma T, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Consistency without ordering. In: Proc. of the USENIX Conf. on File and Storage Technologies. USENIX Association, 2012.

[25] Lu Y, Shu J, Wang W. ReconFS: A reconstructable file system on flash storage. In: Proc. of the USENIX Conf. on File and Storage Technologies. USENIX Association, 2014. 75−88.

刘星(1991－),男,江西萍乡人,硕士,主要研究领域为云存储,操作系统.

江松(1969－),男,博士,研究员,博士生导师,主要研究领域为操作系统,文件和存储系统,高性能计算,互联网与云计算.

王洋(1966－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为并行分布计算,云计算,虚拟化技术.

范小朋(1978－),男,副研究员,CCF 专业会员,主要研究领域为移动计算,云计算,大数据分析,车联网.

须成忠(1965－),男,博士,研究员,博士生导师,主要研究领域为并行与分布式系统,互联网与云计算,高性能计算,移动嵌入式系统.