高技术通讯 2024 年 第 34 卷 第 4 期:366-378

doi:10.3772/j.issn.1002-0470.2024.04.004

一种日志结构块存储系统一致性模型①

杨勇鹏② 蒋德钧③

(中国科学院计算技术研究所 北京 100190)

(中国科学院大学 北京 100049)

摘 要 随着物理设备容量增大,日志结构块存储系统一致性模型及异常恢复的时间和空间复杂度都在增加。一致性状态作用域大小与异常恢复复杂度成反比,与写请求冲突概率成正比。首先,提出单一一致性状态定义(CTS),降低异常恢复复杂度。其次,分析一致性状态生成的充要条件和开销,引入 WSL 链表设计一致性状态生成算法;在一致性状态生成算法的基础上,本文提出多 WMT 元数据管理结构 MCT,将一致性状态生成与用户写请求冲突降低至链表结点级别。最后,以一致性模型为基础设计异常恢复算法,并设计实现日志结构块存储系统 SCB。实验表明,相对于 ASD 系统,SCB 系统吞吐率提升 135.59%,99.90% 尾延迟降低 42.89%,fileserver 负载性能提升 25.00%,异常恢复时间为 ASD 的 1/23。相对于 dm-thin 系统,SCB 系统吞吐率提升 225.72%,varmail 负载性能提升 46.67%。

关键词 日志结构存储系统;块存储系统;异常恢复;一致性状态;一致性模型

日志结构(log-structured)技术被应用于基于硬盘(hard disk drive,HDD)和固态盘(solid state disk,SSD)的块存储系统中,解决存储介质随机访问和顺序访问性能差异大的问题。随着物理设备容量逐渐增大[1-2],日志结构块存储系统维护的映射关系规模也在增大,因此映射关系的一致性模型和异常恢复时重构的复杂度都在增加。

日志结构技术最初由 Sprite LFS(log-structured file system)[3]提出,目的是解决 Unix 文件系统访问 HDD 寻道时间过长的问题。采用日志结构设计的文件系统,用户数据和文件系统元数据写 HDD 的方式均为顺序写。掉电或宕机等异常事件不可避免,因此文件系统的异常恢复一致性和效率非常重要。日志结构文件系统的异常恢复通常分为 2 个阶段,即检查点(checkpoints)和向前回滚(roll-forward)[3]阶段。checkpoints 定义文件系统的一致性状态,系统异常恢复总能找到一个最新的一致性状态,根据该信息文件系统可正常挂载、访问。若需要恢复最新的一致性状态之后写入的数据,文件系统需要执行 roll-forward,扫描有限区域,将最新的一致性状态之后写入的数据和文件元数据修改回放,并将新的一致性状态更新至物理设备。

文件系统的非同步写请求,将数据拷贝至页高速缓存(page cache)即可返回。基于该设计,日志结构文件系统可阶段性生成一致性状态,且在一致性状态生成期间不允许处理用户同步写请求。因此,若参照日志结构文件系统的设计,生成一致性状态时,块存储系统收到的写请求需要等待一致性状态生成并持久化成功后再处理,严重影响写请求延迟。

现有的日志结构块存储系统,按照逻辑地址和物理地址的映射关系(logical to physical, L2P)管

① 中国科学院战略性先导科技专项(XDB44030200)资助项目。

② 男,1993 年生,博士生;研究方向:块存储系统和文件系统;E-mail: yangyongpeng15@mails.ucas.ac.cn。

③ 通信作者,E-mail: jiangdejun@ict.ac.cn。

(收稿日期:2023-02-24)

理,L2P 表(记录块存储系统所有 L2P)的更新策略可分为 3 类。第 1 类,HPDA(hybrid parity-based disk array)[4] 和 LDM(log disk mirroring)[5] 系统采用非易失内存(non-volatile random-access memory,NVRAM)持久化 L2P 表。第 2 类,Gecko[6] 和 LSVD(log-structured virtual disks)[7] 系统在内存中全量缓存 L2P 表,Gecko 全量更新 L2P 表,LSVD 不更新 L2P 表。第 3 类,BW-RAID(redundant array of inexpensive disks)[8] 存储系统的 ASD(allocate on-demand storage device)子系统[9-10],增量更新、增量恢复 L2P 表。上述系统中,只有 ASD 系统的设计原则可应对大容量存储设备面临的随着设备容量增大 L2P 表规模增大、异常恢复复杂度增加的问题。

为达到增量恢复 L2P 表的目标,ASD 系统提出最小时间戳技术,该技术定义一致性状态、缩小一致性状态作用域。ASD 为每个写请求分配一个整数时间戳,代表写请求向下转发的相对时间。ASD 将逻辑空间等分成多个 Subtree,其中记录了 L2P。多个 Subtree 组成一个 Group,每个 Group 维护一个一致性状态,即最小时间戳,也是一个整数时间戳,代表该时间戳之前 ASD 收到的写请求都已经写完成,且 L2P 已持久化至物理设备,异常恢复无需处理。ASD 提出 LAM(log area management)技术,在物理设备上固定区域记录异常恢复时需要扫描的物理设备空间,执行 roll-forward 将最小时间戳之后 ASD 收到的写请求对应的 L2P 更新至 L2P 表。

ASD 将逻辑空间分片,使得不同的逻辑空间拥有不同的一致性状态,因此 ASD 的一致性状态由多个最小时间戳组成。由此导致 2 个问题。(1) Group 生成一致性状态时,Group 所包括的逻辑空间的写请求不能及时处理;(2) 异常恢复需要维护多个 Subtree 和最小时间戳的关系,且判定一个数据块对应的 L2P 是否已经持久化需要查询相应的最小时间戳。ASD 如此设计会造成异常恢复的时间和空间复杂度过高。上述 2 个问题存在矛盾,分片的逻辑空间越小冲突概率越小,但一致性状态维护成本越高,反之亦然。

因此,本文主要聚焦基于大容量物理设备的日志结构块存储系统的一致性状态定义与异常恢复机制,旨在提升写请求的处理效率,同时降低异常恢复的时间和空间复杂度。本文的主要贡献有如下 3 点。

(1) 提出一种日志结构块存储系统一致性状态定义(consistent timestamp,CTS),并设计一致性状态生成算法。CTS 降低一致性状态维护开销。一致性状态生成仅发生在 L2P 表下刷前且开销小,对写请求处理的干扰可忽略不计。

(2) 基于一致性状态定义和生成算法,设计一致性模型和异常恢复算法,使得日志结构块存储系统异常恢复复杂度与物理设备容量大小无关。

(3) 以上述贡献为基础,设计实现日志结构块存储系统 SCB(single consistent state block device),通过该系统评价一致性模型的收益与开销。

1 相关工作

与本文主题相关的研究工作主要包括 2 部分:日志结构文件系统和块存储系统。本节主要从一致性状态定义与异常恢复机制 2 个角度分析上述 2 类研究。

1.1 日志结构文件系统

Sprite LFS[3] 提出日志结构技术,并实现基于该技术的文件系统,NILFS2(new implementation of a log-structured file system)[11] 是其在 Linux 内核[12] 中的实现。SFS[13] 在 NILFS2 基础上,针对闪存介质进行了优化。F2FS(file system for flash storage)[14] 则基于原生闪存介质进行全新设计,按数据和元数据类型划分冷热区域,不同区域的写请求处理均为顺序写闪存设备。

NILFS2 和 F2FS 的异常恢复机制有相同点也有差别。相同点有:(1)在物理设备固定区域存放 2 份一致性信息,同步一致性状态仅写其中一份,2 个区域交替使用;(2)异常恢复从固定区域获取最新的一致性状态信息,即可构建出可挂载、访问的文件系统实例;(3)通过一致性信息中记录的活跃区域信息,异常恢复可确定在最新的一致性状态持久化成功后,写入数据和元数据的起始地址,并以该信息为依据执行异常恢复操作。

二者不同点有:NILFS2 可以部分恢复最后一次持久化一致性状态之后写入的用户数据,而 F2FS 仅恢复同步写数据。二者均可确保文件系统一致性及同步写语义,只是异常恢复效率与数据可恢复程度设计的取舍有所不同。

综上,现有日志结构文件系统的一致性状态定义与异常恢复机制的设计都遵循 checkpoints 和 roll-forward 原则,通过固定区域记录一致性信息,并指向异常恢复需要扫描的物理设备区域,扫描物理设备以增量恢复文件系统的数据和元数据。

1.2　非日志结构块存储系统

本节介绍 Linux 内核[12]中 2 个典型的非日志结构块存储系统 bcache 和 dm-thin 系统。

bcache 系统是一种 SSD 缓存系统,采用 B + tree 管理 SSD 中缓存的数据,使用日志(journal)技术记录写请求对 L2P 表的修改以持久化写请求,避免频繁更新 B + tree。异常重启时,通过回放日志即可恢复 L2P 表。

dm-thin 系统为一种采用精简配置的标准块设备,可提供快照功能。dm-thin 为写请求动态分配物理地址,使用 B + tree 管理 L2P 表。dm-thin 没有采用日志结构技术或日志技术持久化 L2P 表修改,而是通过用户态进程定时查询设备状态触发元数据同步的方式增量更新 L2P 表,以减少异常宕机丢失的数据。dm-thin 系统支持 sync 语义,可确保 sync 之前收到的写请求对应的 L2P 修改都已持久化。

1.3　日志结构块存储系统

本节介绍 5 个日志结构块存储系统相关的工作:HPDA、LDM、Gecko、LSVD 和 ASD 系统。

1.3.1　基于非易失介质的日志结构系统

HPDA 系统和 LDM 系统,利用日志结构技术加速 HDD 组成 RAID1 收到的小粒度写请求。二者均使用 NVRAM 存放 L2P 表,将写请求数据写入 RAID1,更新 NVRAM 中记录的 L2P 表后,写请求便可向上层回调。

因此,上述 2 个系统均可保证返回的写请求均已持久化,异常或正常恢复时,仅需扫描非易失设备即可恢复 L2P 表。

1.3.2　Gecko 系统

Gecko 系统利用日志结构技术加速 HDD 组成 RAID 的写请求处理效率。Gecko 系统利用一块 SSD 记录反向映射(physical to logical,P2L)信息,L2P 表常驻内存,且不持久化。异常或正常重启时,Gecko 系统扫描 SSD 中记录的 P2L 信息构建 L2P 表。

1.3.3　LSVD 系统

LSVD 为 SSD 缓存系统,通过客户端 SSD 加速后端 S3 服务,二者均采用日志结构设计。LSVD 系统的日志结构格式如图 1 所示。

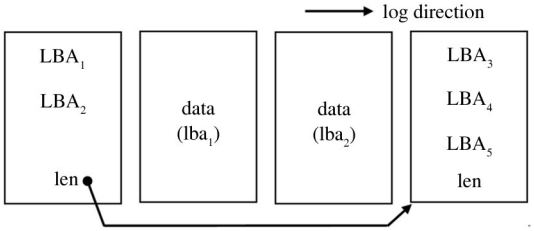

图 1　LSVD 系统的日志结构[7]

图中日志结构头部信息包含一个递增的序列号,代表日志头写入的相对时间。日志结构头部还包含指向数据的 CRC 校验,重启时用以检查数据和日志结构头是否一致。

LSVD 系统的 L2P 表常驻内存,阶段性将 L2P 表全量写至物理设备固定区域,阻塞用户数据写并生成检查点(checkpoint),即一致性状态。检查点包括一个序列号和下次将要写入的物理地址,检查点生效则代表该序列号之前写入数据对应的 L2P 均已持久化。异常重启时,读取检查点获取最近一致性状态,roll-forward 扫描检查点中记录的位置,读取日志结构头信息,更新 L2P 表。

1.3.4　ASD 系统

首先介绍 ASD 系统的物理设备布局,图 2 为其示意图。

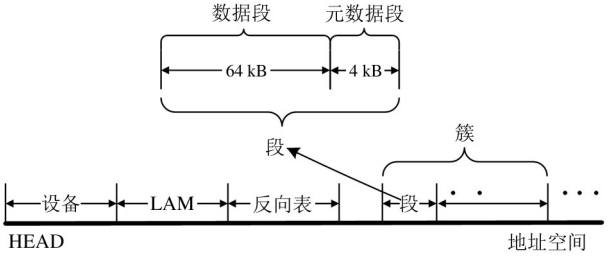

图 2　ASD 系统物理设备布局[9-10]

主要区域的含义如下所述。

(1)设备信息存放虚拟设备的信息,如设备大小、名称等,作为构建块设备实例的基本信息。

(2)LAM 信息记录了所有的活跃簇,该部分等分为 2 个区域,记录 2 个版本的 LAM 信息,交替使用以实现原子更新。异常恢复以此区域中记录的活跃簇信息为依据,扫描指定物理设备空间。

(3)段:由数据段和元数据段组成,其中元数据段记录数据段中各个数据块的归属信息,即 P2L 信息,由此可构造出多个 L2P。数据块中记录用户数据或设备的 L2P 信息。

(4)反向表:L2P 表切分为多个块同步至数据块,反向表描述虚拟设备的 L2P 表存放在哪些数据块中。

(5)簇:长度固定的多个段组成一个簇。

ASD 将逻辑空间等分为多个较小的逻辑空间 Subtree,ASD 的 L2P 定义为一个 Subtree 内逻辑地址连续且物理地址连续的一段映射关系,又称作一个 Extent。多个 Subtree 组成一个 Group,在内存中由红黑树组织,每个 Group 对应一个红黑树结点,所有 Group 组成 L2P 表。每个红黑树结点在 L2P 表下刷时会写至物理设备的一个数据块。由于每个 Subtree 的逻辑块映射信息可能存在 3 个状态:都有映射、都没有映射、部分有映射,因此每个红黑树结点可能包含数量不等的 Subtree。Subtree 包含的 L2P 数量会变化,而 Group 固定对应一个数据块,因此 Subtree 在不同时刻可能归属于不同的 Group。所以,每个 Subtree 可能对应不同的最小时间戳。

ASD 正常运行时,为每个 Group 维护一个最小时间戳。异常恢复需要为每个 Subtree 维护一个最小时间戳,若异常恢复涉及某个 Subtree 中的逻辑块,则仅恢复该 Subtree 最小时间戳之后写入的用户数据对应的 L2P。

日志结构块存储系统的异常恢复目标:读取最新的 L2P 表,并将最新 L2P 表之后写入数据对应的 L2P 更新至 L2P 表中并持久化。

ASD 的异常恢复流程为:读取反向表信息,通过反向表读取数据块中记录的 L2P 信息,构建出最新版本 L2P 表,并维护 Subtree 与最小时间戳的关系;扫描 LAM 指向区域的所有元数据段,解析出 L2P 信息,若时间戳大于该逻辑块所在 Subtree 的最小时间戳,则更新 L2P 表。

因此,ASD 系统与日志结构文件系统的一致性模型存在差异:一致性状态定义不同;异常恢复扫描区域与最新的一致性状态无关。前者会导致异常恢复的复杂度增大;后者是块设备语义导致的,L2P 表下刷可能与数据写并发,因此 LAM 指向的区域可能会发生变化。

综上,ASD 的一致性状态定义为 Group 最小时间戳的集合,Group 数量越多,复杂度越高。ASD 的异常恢复机制与日志结构文件系统基本一致,但由于块存储系统语义与文件系统存在差异,因此实现机制不同。

上述 5 个日志结构块存储系统,除 LSVD 系统和 ASD 系统外,均需要全量恢复 L2P 表;除 ASD,HPDA 和 LDM 系统增量更新 L2P 表,其余系统不持久化 L2P 表或全量更新 L2P 表。

2 问题分析

本节以日志结构块存储系统为研究对象,以增量更新、恢复 L2P 表为研究目标。首先,分析一致性状态作用域大小与一致性状态生成复杂度的关系。接着,本节介绍采用单一一致性状态定义,生成一致性状态的充要条件和复杂度分析。

2.1 一致性状态定义

一致性状态定义的适用范围可以是整个逻辑地址空间,也可以是部分逻辑地址空间,本文将该适用范围称作一致性状态作用域。如果块存储系统的一致性状态作用域为全局,那么块存储系统仅有一个一致性状态,即单一一致性状态。

对于块存储系统,由于一致性状态生成不能阻塞用户写请求,因此需要一种机制来区分数据是在一致性状态之前或之后写完成,即区分数据新旧的机制。然而,这种机制的作用域是单个逻辑块还是更大的逻辑地址空间,则取决于一致性状态的定义。一致性状态作用域大小决定了一致性状态维护开销和异常恢复机制复杂度,二者成反比。

ASD 使用整数时间戳作为区分数据新旧的机制,通过 Group 数量的最小时间戳集合定义 ASD 的一致性状态。运行时最小时间戳的作用域为 Group,根据 1.3.4 节介绍,异常恢复时最小时间戳的作用域为 Subtree。由于 Group 与 Subtree 的对应关系不固定,因此时间戳为全局时间戳,即每个写请求拥有不同的整数时间戳。

ASD 的一致性状态维护分为 2 种场景。(1)系统正常运行需要维护 Group 数量的最小时间戳,并同步至物理设备;(2)系统异常恢复需要维护 Subtree 数量的最小时间戳。

因此,ASD 一致性状态维护的时间和空间复杂度与 Group、Subtree 粒度相关,随着 L2P 表规模扩大,一致性状态维护的空间和时间开销呈线性增长。

一致性状态作用域为一个 L2P 是该问题的最差情况。该问题的最优解是采用单一一致性状态定义:确保所有该一致性状态之前写入的数据对应的 L2P 均已持久化,无需异常恢复。一致性状态的作用域为全局,因此区分数据新旧的机制也要作用于全局。本文采用时间戳机制区分数据新旧,则将单一一致性状态定义为 CTS。

若基于 ASD 现有设计,采用 CTS 定义一致性状态,生成一致性状态的时间开销为所有 Group 生成一致性状态时间开销的总和。生成一致性状态时,所有写请求均不能得到处理,即造成冲突区域增大。

因此,采用单一一致性状态定义,则需要解决 2 个问题:降低生成一致性状态的时间复杂度;降低写请求与一致性状态生成的冲突粒度,从而降低冲突概率。

2.2 生成一致性状态

生成 CTS 的基本原则:在最新的 CTS 之后,若存在一个时间戳 *ts*,使得 L2P 表下刷成功后所有小于等于该 *ts* 的 L2P 均已同步成功,则该 *ts* 即为 CTS。若要生成 CTS,则需要维护上一次 CTS 之后所有写请求的状态。因此,实现一致性状态生成,需要维护的必要信息为通过逻辑块地址(logical block address,LBA)索引如表 1 所示的内容。

其中 *refs* 字段与 LAM 设计相关。首先明确 LAM 与簇的关系,当一个簇没有可用数据块,且簇中所有写请求对应的 *ts* 均小于等于 CTS,则该簇可以从 LAM 中删除,不影响系统一致性。因此,在 L2P 表同步成功后,若 *ts* 小于等于 CTS,则 *refs* 指向的内存和簇资源即可释放。所有簇的引用都释放后,LAM 可不包含该簇。

表 1 生成 CTS 需要的信息

字段	含义
ts	写请求对应的时间戳
state	写请求是否完成,P(*pending*)/C(*completed*)
pbid	物理设备的 LBA
refs	指向需要释放的资源

综上,生成 CTS 的充要条件为:维护 LBA 至表 1 的映射,将该映射关系的集合称作 WST(write I/O state table)。

2.3 生成一致性状态复杂度分析

为维护一致性状态的必要信息,写请求下发之前和回调之后均需要更新 L2P 表,且写请求下发之前修改 WST 必须按照时间戳大小串行插入,否则遍历 WST 生成 CTS 可能出现 CTS 语义错乱,从而造成数据不一致。

按照 2.2 节提出的生成 CTS 基本原则,生成一致性状态需要遍历 WST 实例。然后,再次遍历 WST,将 *ts* 小于等于 CTS 的 L2P 合并至 L2P 表并下刷。一个 L2P 合并完成后,即可将其从 WST 中删除。

若每个块设备仅有一个 WST 实例,则会存在如下访问冲突:

(1)写请求需要 2 次修改 WST,第 2 次修改仅更新 *state* 信息。

(2)一致性状态生成至少需要 2 次遍历 WST 实例和数次删除操作,且可能与用户读写请求冲突,从而影响用户读写请求处理。

若每个块设备拥有 2 个 WST 实例,分别称作 wst_1 和 wst_2,wst_1 为活跃状态,则在通过 wst_1 生成 CTS 并下刷 L2P 表后,需要将 wst_1 中所有 CTS 之后插入的 L2P 插入 wst_2,最后 wst_1 将会被销毁。相对于单一 WST 实例,第 1 类冲突仍然存在,可消除第 2 类的删除冲突。将 wst_1 中 CTS 之后插入的信息迁移至 wst_2 可能与用户请求冲突,且需要处理用户请

求向上层回调前更新 wst_1 与迁移操作的并发。

综上,仅增加 WST 实例并不会降低复杂度、减少一致性状态生成导致的冲突,反而会引入更多问题。

3 一致性模型设计

根据第 2 节分析,采用 CTS 定义一致性状态需要降低一致性状态生成的复杂度,并减少与写请求的冲突。本节分别针对这 2 个问题,提出一致性状态生成算法和一致性模型,并在此基础上设计日志结构块存储系统异常恢复算法。

3.1 一致性状态生成算法设计

按照 2.1 节生成一致性状态的原则,通过 WST 生成一致性状态,存在 2.3 节提出的 2 个冲突问题。为避免、降低上述冲突,提出辅助数据结构 WSL(write I/O state list)链表,维护表 1 中的 *ts*、*state*、*refs*,将这些信息称作 WSR(write I/O state record)。引入 WSL 后,WST 仅剩 LBA 至 *pbid* 的映射,因此将其称作 WMT(write I/O mapping table)。WSL 链表结点分为 2 部分:结点头部和 WSR 数组,其中头部信息如表 2 所示。

表 2 WSL 链表结点头部

字段	含义
next	指向下一个链表结点
count	链表结点中 WSR 总数
start	有效记录的起始位置
pendings	*pending* 状态写请求数量
lock	保护链表结点的锁

写请求向物理设备转发之前,在 WSL 尾部追加一条 WSR 信息,若无可用空间则分配新链表结点,并将 *next* 指向新分配的链表结点。写请求追加 WSR 后,更新 *count*,并保存 WSR 指针。写请求向上回调前,通过 WSR 指针更新 *state* 为 *completed*。

写请求的 *ts* 和 *state* 信息都存在 WSL 中,因此只需要通过 WSL 链表即可生成 CTS,生成 CTS 的具体算法如算法 1 所示。

算法 1 生成 CTS

```
1:  do {
2:      wsl_list = wsl_node = wsl_list→next
3:      if (wsl_node→count == MAX_RECORDS&&wsl_node→pendings == 0)
4:          cts = last_ts(wsl_node)
5:  else
6:      cts = last_completed_ts(wsl_node)
7:      break;
8:  end if
9:  } while (list_empty(wsl_list→next))
10: return cts;
```

算法 1 遍历 WSL,第 3 行判断链表结点是否满,是否所有 WSR 数组中记录的写请求 *state* 均为 *completed*,是则调用 *last_ts* 获取 WSR 数组最后一项对应的 *ts*,否则跳入第 6 行。第 6 行调用 *last_completed_ts* 函数,遍历 WSR 数组找到状态为 *pending* 的 WSR 的前一个 WSR,获取其 *ts* 作为 CTS。

根据算法 1 设计,WSL 仅维护最新 CTS 之后收到的写请求 *state*,因此算法 1 时间复杂度与 L2P 表规模无关,仅与最新 CTS 之后的写请求数量相关。

WSL 链表结点内部 WSR 紧密排列,不存在删除和中间插入操作,因此空间利用率最大。WSL 存储信息与 WMT 并无交叉,因此 WMT 和 WSL 链表设计的空间利用率不低于 WST。块存储系统的请求处理基本遵循先来先服务的原则,CTS 之后收到的写请求大概率没有回调,因此 WSL 空间利用率高。

综上,算法 1 的时间复杂度仅与 WSL 规模相关,且 WSL 空间利用率高。因此,通过 WSL 生成 CTS 的时间复杂度可控,可解决 2.1 节提出的第 1 个问题。

3.2 一致性模型设计

本节以算法 1 为基础,设计 L2P 表下刷流程和写请求模型,解决 2.1 节提出的第 2 个问题。

引入算法 1 后,L2P 表的下刷流程大致为:执行算法 1 生成 CTS,合并 WMT 中的 L2P 至 L2P 表,并将 L2P 表持久化至物理设备。

采用 WSL 后,写请求回调更新 *state* 操作只会修改 WSL,可消除 2.3 节提出的第 1 类冲突和第 2

类冲突中的遍历冲突。采用单一 WMT，L2P 合并后在 WMT 中删除 L2P 与写请求仍然存在冲突。若采用多 WMT 设计，执行算法 1 生成 CTS 后，当前活跃 WMT 变为非活跃状态不再允许插入，之后写请求仅修改活跃 WMT。

算法 1 执行与禁止 WMT 插入之间存在时间窗口，该窗口内可能发生 CTS 之后收到的写请求更新即将变为非活跃状态的 WMT。所以，将 WMT 合并至 L2P 表后，L2P 表中可能存在 CTS 之后写入的数据对应的 L2P。然而，上述 L2P 已经持久化成功，即便 L2P 表下刷完成后异常宕机，异常恢复模型也需要保证将该 L2P 恢复至 L2P 表。因此，合并 WMT 至 L2P 表可为全量合并，采用算法 1 和多 WMT 设计，便可彻底消除 2.3 节中提出的第 2 类冲突。

采用多 WMT 设计，每个 WMT 便对应一个 CTS，所以若忽略资源占用和维护复杂度，WMT 实例数量没有限制。引入多 WMT 设计后，将元数据管理结构称作 MCT（multi consistent state table），结构图如图 3 所示。

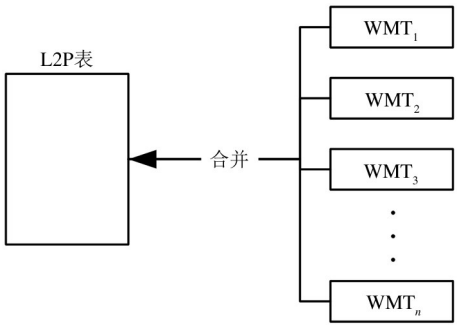

图 3　MCT 结构图

以 2 个 WMT 实例 wmt_1 和 wmt_2 为背景，WSL 链表实例为 *wsl*，说明写请求的处理过程和 L2P 表下刷。MCT 初始状态 1 的 *wsl* 如表 3 所示，wmt_1 和 wmt_2 均为空。

表 3　状态 1 的 *wsl*

ts	1	2	3	4	5
state	P	P	P	P	P

写请求向物理设备转发之前，在 *wsl* 中添加一项 WSR 实例 *wsr*，并将其 *state* 标识为 *pending*。写请求回调后，将 L2P 插入 wmt_1，随后在 *wsl* 中将 *wsr* 的 *state* 更新为 *completed*，MCT 到达状态 2。状态 2 的 *wsl* 和 wmt_1 分别如表 4 和表 5 所示。

表 4　状态 2 的 *wsl*

ts	1	2	3	4	5
state	C	C	P	C	P

表 5　状态 2 的 wmt_1

LBA	*pbid*	*ts*
3	1	1
7	2	2
11	4	4

状态 2 之后，若 L2P 表下刷被触发，下刷操作：执行算法 1，生成 CTS 为 2；将 wmt_1 中的 L2P 合并至 L2P 表。L2P 表下刷完成后，MCT 到达状态 3，*wsl* 如表 6 所示，小于等于 CTS 的记录已经被清除。

表 6　状态 3 的 *wsl*

ts	3	4	5
state	P	C	P

ts 为 3 和 5 的写请求返回后，L2P 将会被插入 wmt_2，MCT 到达状态 4，如表 7 所示。如果在下一次 L2P 表下刷之前发生异常宕机，wmt_2 中时间戳为 3 和 5 的 L2P 将会丢失，因此异常恢复需要将 wmt_2 中记录的 L2P 恢复至 L2P 表中。

表 7　状态 4 的 wmt_2

LBA	*pbid*	*ts*
5	3	3
9	5	5

根据上述 L2P 表下刷流程和写请求模型设计，算法 1 生成 CTS 与写请求添加和更新 *wsr* 存在冲突，后者时间复杂度为 O(1)。算法 1 仅在 L2P 表下刷时才会执行，并非高频操作，且与写请求的冲突为链表结点级别。因此，一致性状态生成与写请求的冲突概率小，且冲突操作时间复杂度低。

综上,采用算法 1 和多 WMT 设计的元数据管理结构 MCT 后,L2P 表下刷与写请求模型设计将冲突粒度降至链表结点级别,解决了 2.1 节提出的第 2 个问题。

3.3 异常恢复算法设计

日志结构块存储系统最重要的元数据是 L2P 表,正常状态确保:正常返回的写请求已持久化至物理设备;最新 L2P 表之后写入的数据对应的 L2P,可通过扫描固定物理设备空间获取。

因此,异常恢复的目标是:将异常重启之前,最新 CTS 之后写入数据对应的 L2P 合并至 L2P 表并下刷,同时更新 CTS。异常恢复算法如算法 2 所示。

算法 2 异常恢复算法

```
1: cts = last_cts = 读取最新 CTS
2: 读取并构建最新 L2P 表
3: 读取最新 LAM 信息,获取需要扫描的簇集合 clusters
4: for cluster ← clusters
5:   for mseg ← cluster
6:     if (mseg→ts > last_cts)
7:       mesg⇒ wmt_1 / * mseg 中记录的 L2P 插入 wmt_1 * /
8:       cts = max(cts, mseg → ts);
9:     end if
10:  end for
11: end for
12: 合并 wmt_1 至 L2P 表并下刷,更新 CTS 为 cts
```

下面结合 3.2 节中的案例介绍算法 2。

第 1 行读取最后一次 L2P 表下刷对应的 CTS,即为 2。第 2 行,读取并在内存中构建出最新的 L2P 表,以便后续增量修改、下刷 L2P 表。第 3 行自 LAM 区域读取活跃簇信息,即为异常恢复时需要扫描的物理设备空间。

读取最新的一致性状态后,第 4 ~ 5 行遍历所有簇中的元数据段信息 *mseg*, 该信息可解析出 LBA 与 *ts*、*pbid* 的映射关系。第 6 行判定若写请求晚于 CTS,则需要异常恢复,第 7 行将其插入 wmt_1 中。第 8 行更新 *cts* 为当前扫描到的最大 *ts*。第 4 ~ 11 行执行结束后,必要信息读取结束,结合 3.2 节,可得到表 8 所示的 wmt_1。

表 8 异常恢复阶段 wmt_1

LBA	*pbid*	*ts*
11	4	4
5	3	3
9	5	5

相对于表 7,表 8 多出 *ts* 为 4 的 L2P,该现象符合 CTS 语义,且该 L2P 为 LBA 为 11 的最新版本,重复插入也不会影响一致性。由于 *ts* 按照写请求的下发顺序分配,写请求基本遵循时间戳越小越早完成的规律,因此重复插入事件并不会大规模发生。

wmt_1 构造完成后,相应的 CTS 为扫描到的最大 *ts*,第 10 行执行一轮 L2P 表下刷,即可完成 L2P 表的异常恢复。

3.4 评价系统

评价系统物理设备布局沿用 ASD 设计,L2P 表采用文献[15]提出的元数据管理结构 IBT B + tree,该数据结构解决了 wandering B + tree 问题[16]。WMT 采用文献[17]提出的查询、修改操作可并发的 B + tree。结合上述一致性状态定义、一致性模型和异常恢复模型,实现日志结构块存储系统 SCB(single consistent state block device)。

为评价一致性模型开销,本文在 SCB 基础上去掉 WSL 设计,因此该版本不支持异常恢复。将该版本称作 wSCB(SCB without WSL support)。

4 系统评测

本节测试的对比对象为 ASD、wSCB、SCB 和 dm-thin 系统,主要测试 Fio[18] 负载和 Filebench[19] 负载。测试 Fio 负载,对比 ASD 与 wSCB,量化单一一致性状态定义 CTS 的收益,对比 wSCB 和 SCB 量化一致性保证的开销,对比 SCB 系统和 dm-thin 系统量化日志结构块存储系统的开销和收益。测试 Filebench 负载,评价 SCB 在各类负载下的性能和异常恢复效率表现。

4.1 评测环境

测试的软硬件环境如表 9 所示,本节 4 个测试对象系统均以表 9 所示的 SSD 设备作为底层物理设备,测试过程中的变量主要有:L2P 表是否都缓存

在内存中;WMT 的内存占用上限。由于 4 个测试对象系统元数据管理结构的空间利用率不同,占用的内存和设备空间大小不同,因此本节测试将 4 个系统的 L2P 表均缓存在内存中以规避该问题。WMT 内存占用上限配置不同,性能表现会有差异,因此后续章节会分析其对性能的影响,不同测试用例均会说明 WMT 内存占用上限。

表 9　测试服务器软硬件配置

类别	参数
操作系统	CentOS Linux 7.8.2003,内核 3.10.0-957.12.2
Fio	3.1
Filebench	1.4.9.1
CPU	Intel(R) Xeon(R)E5645@2.40 GHz,2 路,24 线程
内存	Ramaxel DDR3,12 GB,1 333 MHz
SSD	Intel SSD DC P3700 Series,400 GB,NVMe

4.2　Fio 测试

本节主要通过 Fio 测试对比 4 个系统的吞吐率、延迟,并结合一致性模型分析 SCB 系统的收益与开销。Fio 测试的公共参数配置如表 10 所示。Fio 测试仅测试 4 kB 随机写请求,因此将 4 个对比系统的粒度均配置为 4 kB。

表 10　Fio 配置参数

参数	取值
设备大小/GB	300
iodepth	256/1
粒度/kB	4
engine	libaio
读写	randwrite

4.2.1　吞吐率和延迟测试

测试吞吐率,iodepth 为 256,写入数据量为 200 GB,WMT 内存占用上限为 30 MB,吞吐率测试结果如图 4 所示。

分析图 4,可得到如下结论。

(1) 相对于 ASD,wSCB 和 SCB 的吞吐率分别提升 140.13% 和 135.59%;

(2) 相对于 wSCB,SCB 的吞吐率降低 1.89%,

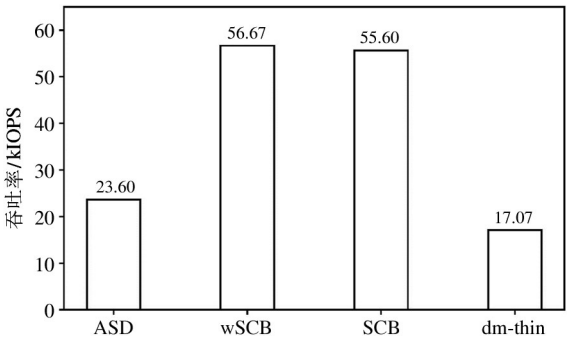

图 4　吞吐率测试对比图

绝对值相差较小;

(3) 相对于 ASD、wSCB 和 SCB,dm-thin 吞吐率分别降低 27.67%、69.88% 和 69.30%。

对于第(1)点,wSCB 相对于 ASD 的性能提升,主要原因是采用了全局一致性状态,且避免了 L2P 表下刷与写请求冲突,提升了写请求处理效率。

对于第(2)点,对比 wSCB 和 SCB 用以评价一致性保证的开销。wSCB 和 SCB 系统的主要差异为写请求和 L2P 表下刷。写请求处理需要修改 WSL,L2P 表下刷需要遍历 WSL。从结果看,WSL 并发冲突的开销很小,SCB 测试过程中 L2P 表共计下刷 138 次,写请求与 L2P 表下刷冲突概率很小。因此,一致性模型对吞吐率的影响很小。

对于第(3)点,dm-thin 吞吐率低的主要原因是 L2P 表下刷频繁。经统计,按图中自左至右顺序,L2P 表下刷量分别为 87.47 GB、48.79 GB、49.05 GB 和 176.80 GB。dm-thin 的 L2P 表下刷量较其他系统更大,且 dm-thin 下刷 L2P 表会阻塞写请求。

WSL 对写请求的影响:在写请求下发和回调阶段分别增加 WSL 修改操作,写请求下发可能存在内存分配,而回调不存在。接下来,本节通过平均延迟、99.0% 尾延迟、99.9% 尾延迟评价一致性模型对延迟的影响。

Fio 测试的 iodepth 为 1,测试时长为 30 min,WMT 内存占用上限为 30 MB,统计延迟数据,可得图 5。

根据图 5,可得出如下结论。

(1)平均延迟,相对于 ASD,wSCB 和 SCB 分别降低了 21.95% 和 23.83%,wSCB 比 SCB 高 3 μs;

(2)99.0% 尾延迟,相对于ASD,wSCB和SCB分

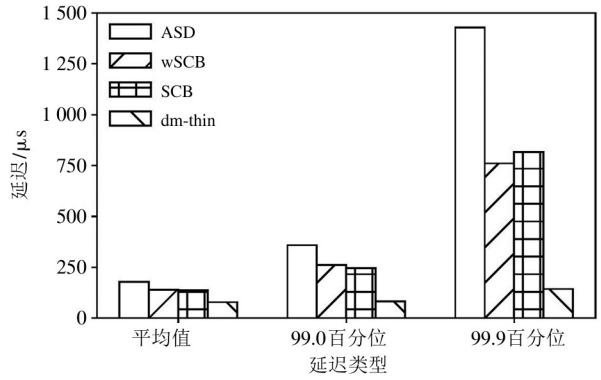

图 5　各版本系统延迟对比

别降低了 27.30% 和 31.38%,wSCB 比 SCB 高 15 μs;

(3)99.9% 尾延迟,相对于 ASD,wSCB 和 SCB 分别降低了 46.72% 和 42.89%,wSCB 比 SCB 低 44 μs;

(4) dm-thin 系统的 3 类延迟分别比 SCB 系统低 42.41%、66.85%、82.43%。

第(1)点,SCB 和 wSCB 的平均延迟差别很小,因此 WSL 修改操作对写请求的影响可以忽略不计。相对于 ASD,SCB 平均延迟降低主要源自写请求模型对写请求处理路径的简化。写请求仅访问规模相对较小的 WMT 和 WSL,避免 L2P 表访问、减少冲突概率,从而降低了写请求处理的时间复杂度。

第(2)点和第(3)点,相对于 ASD 系统,wSCB 和 SCB 系统的尾延迟降低幅度较平均延迟更大。主要原因是新的写请求模型减少了写请求处理路径上的访问冲突,简化了请求处理。

第(4)点,dm-thin 系统的延迟表现更优,主要原因是 dm-thin 系统写请求处理逻辑更简单。前三者处理写请求,均需要写用户数据和 P2L 信息,确保 L2P 持久化成功后,写请求才能向上回调。然而,dm-thin 系统无需保证请求返回前 L2P 已持久化成功,只需要将用户数据写至物理设备。

对比前 3 点,wSCB 和 SCB 对尾延迟的提升幅度更大,因此引入新的写请求模型后,写请求处理路径上资源冲突降低了,避免了耗时较长的操作出现在写请求的生命周期。通过第(4)点可以看出,SCB 系统为确保请求回调前 L2P 持久化成功,降低了系统的延迟表现。

4.2.2　WMT 配置的影响评价

本节测试 WMT 在不同内存上限配置下,SCB 吞吐率的变化情况,以及算法 1 的开销。Fio 测试 iodepth 为 256,写入数据量为 200 GB,WMT 内存占用上限作为变量,吞吐率及算法 1 耗时结果如图 6 所示。

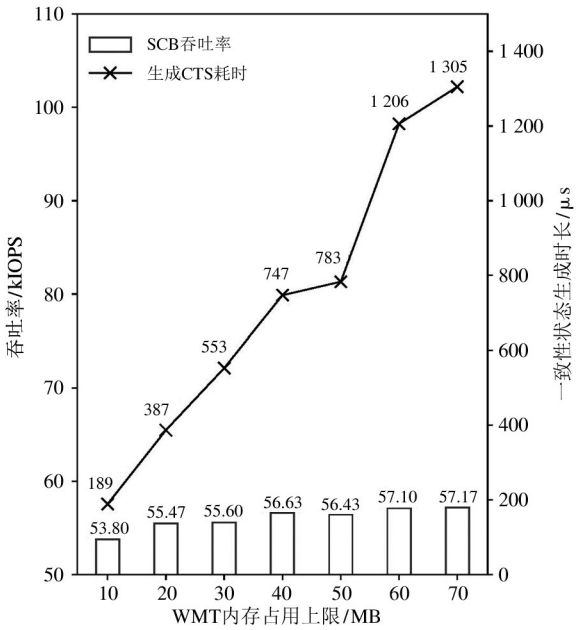

图 6　WMT 内存占用上限与吞吐率和算法 1 耗时

分析图 6,可有如下结论。

(1) WMT 内存占用上限为 10 MB 和 20 MB,带宽提升 3.1%,提升幅度最大,其余情况提升幅度不足 1.0%,且提升绝对值较小;

(2) 随着 WMT 内存占用上限的提升,算法 1 的执行耗时呈线性增长,但绝对值较小。

第(1)点,WMT 内存占用上限超过 40 MB 之后,SCB 再提升该值对吞吐率的贡献可忽略不计。WMT 内存占用上限增大可降低 L2P 表下刷频率,从而减少元数据下刷量。但由于 L2P 表下刷不直接影响写请求处理效率,所以对吞吐率影响不大。该现象与文献[15]的结论一致。

第(2)点,算法 1 开销仅会影响 L2P 表下刷效率,耗时越长 L2P 表一轮下刷时间越长。经统计,图 6 中不同 WMT 内存占用上限配置下,算法 1 耗时与 L2P 表下刷时长的比例约为 0.02% ~0.04%,且绝对值较小。因此,一致性状态生成对 L2P 表下刷效率影响很小。

综上,一致性模型对吞吐率和尾延迟均有较大

幅度提升,且一致性状态生成的开销很小。延迟测试说明:写请求模型对写请求处理路径有较大幅度精简,且一致性模型的开销极小,对 99.9% 尾延迟的贡献最大。通过不同 WMT 内存上限配置下的测试,可见 WMT 内存上限配置对吞吐率的提升效果存在上限,且在不同配置下算法 1 开销均很小。算法 1 执行不直接影响写性能,对 L2P 表下刷效率影响也很小。

4.3　Filebench 评测

选取 Filebench 的 2 个负载 fileserver 和 varmail,分别测试对比 ASD、SCB 和 dm-thin 系统。块设备逻辑空间大小为 200 GB,SCB 系统 WMT 内存占用上限为 30 MB。将块设备格式化为 ext4 文件系统,2 种负载分别执行 1 h,每个测试执行 3 次,结果取平均值,对负载参数作了部分修改,如表 11 所示。

表 11　Filebench 负载参数

负载	文件数	线程数	文件大小	I/O 粒度
fileserver	200 000	50	1 MB	16 kB
varmail	1 000 000	16	32 kB	16 kB

将 Filebench 测试结果按 ASD 系统作归一化处理,结果如图 7 所示。

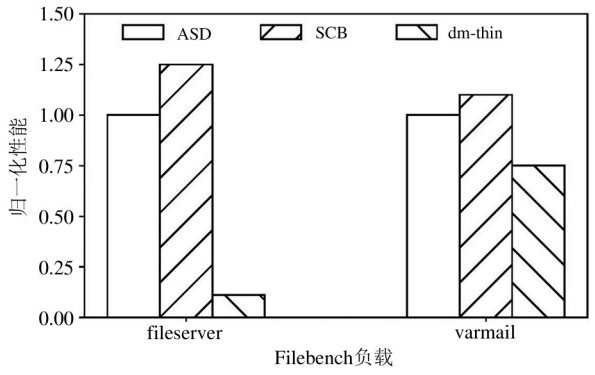

图 7　Filebench 测试结果对比

根据图 7,可有如下结论。

(1) fileserver 负载,相对于 ASD,SCB 提升约 25.00%;

(2) varmail 负载,相对于 ASD,SCB 提升约 10.00%;

(3) dm-thin 测试 2 类负载,分别比 SCB 系统低 91.20%、31.82%。

相对于 Fio 测试,Filebench 的 2 个负载的提升幅度均较低。主要原因是 Filebench 负载经过 ext4 文件系统后,文件系统向块存储系统转发的写请求基本为顺序写。对于顺序写,ASD 下刷 L2P 表与写请求处理冲突概率小,因此降低一致性状态生成与写请求的冲突概率不能体现出来。

根据第(1)点和第(2)点,SCB 系统相对于 ASD 系统,fileserver 的提升幅度高于 varmail。上述现象是读请求处理与写请求、L2P 表下刷冲突导致的,fileserver 和 varmail 的读写比例分别为 1∶2 和 1∶1。SCB 处理读请求,需要先查找规模较小的 WMT,如果不存在则查 L2P 表,所以读请求最多与一个 WMT 实例访问存在冲突。WMT 与 L2P 表合并时,读请求才可能遇到冲突。而 ASD 处理读请求,与写请求、L2P 表下刷均存在冲突。因此,写比例越高,ASD 的读写冲突越严重,SCB 的优化效果越明显。

第(3)点,dm-thin 在读比例较高的负载 varmail 测试中的性能表现,优于写比例较高的 fileserver 负载测试。主要原因是 dm-thin 的 L2P 表下刷频繁,如 4.2.1 节,导致写吞吐率低。dm-thin 和 SCB 系统处理读请求均需要查询 B + tree,二者效率相同,因此读比例越高,差异越小。

4.4　异常恢复评测

dm-thin 异常重启无需恢复 L2P 表,因此异常恢复测试仅针对 ASD 系统和 SCB 系统。异常恢复评测采用 Filebench 负载,测试参数与 4.3 节 Filebench 测试参数相同。在测试结束后立即触发系统强制重启,以模拟系统异常宕机。由于块存储系统被文件系统打开,因此 Filebench 测试结束并不会将所有 L2P 表下刷。ASD 系统异常恢复会产生临时文件,而 SCB 系统不会。为避免临时文件访问介质的性能差异干扰,本节测试将 ASD 临时文件存放在基于内存的文件系统中。经过 3 轮异常重启测试,统计执行时长,结果如表 12 所示。

表 12　异常恢复时长

负载	ASD 恢复时长/s	SCB 恢复时长/s
fileserver	259	13
varmail	475	21

fileserver 和 varmail 负载测试下,SCB 异常恢复时长分别为 ASD 的 1/23 和 1/20,有一个数量级的优化效果。ASD 系统异常重启的主要开销包括:在临时文件中维护 Subtree 和最小时间戳的关系与 L2P 表;对比数据新旧和更新 L2P 需要频繁读写临时文件。而 SCB 系统仅有一个 CTS,对比数据新旧无需读写临时文件。

5 结 论

本文主要研究了日志结构块存储系统面临的随着设备容量增大,异常恢复复杂度增加的问题。现有的日志结构块存储系统,存在一致性状态作用域大小和异常恢复复杂度的矛盾。本文提出单一一致性状态定义 CTS,以减少一致性状态维护开销。在 CTS 的基础上,首先,本文设计出一致性状态生成算法,提出元数据管理结构 MCT,降低一致性状态生成与写请求冲突。接着,本文设计异常恢复算法,实现增量恢复 L2P 表。最后,本文以上述设计为基础,实现日志结构块存储系统 SCB。通过 Fio 测试对比 ASD 和 SCB,SCB 吞吐率提升 135.59%;延迟测试中,SCB 对尾延迟优化更为明显,99.9% 尾延迟降低 42.89%;SCB 生成 CTS 耗时,仅占 L2P 表下刷总时长的 0.02% ~ 0.04%。通过 Filebench 测试对比 ASD 和 SCB,SCB 对写比例较大的负载优化效果更为明显;异常恢复效率有一个数量级的提升。相对于 dm-thin 系统,SCB 系统的吞吐率优势明显,提升 225.72%;由于日志结构设计的开销,SCB 系统延迟表现较差;fileserver 负载和 varmail 负载性能分别提升 1036.36% 和 46.67%。本文提出的日志结构块存储系统一致性模型与 L2P 表和 WMT 具体采用何种数据结构无关,因此可适用于侧重不同场景的各类数据结构。

参考文献

[1] Western Digital. WD gold enterprise class SATA HDD [EB/OL]. (2022-07-01)[2022-12-21]. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-gold/product-brief-wd-gold-hdd.pdf.

[2] Western Digital. Ultrastar DC SN840[EB/OL]. (2022-04-01)[2022-12-21]. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/data-center-drives/ultrastar-nvme-series/product-brief-ultrastar-dc-sn840-gov.pdf.

[3] ROSENBLUM M, OUSTERHOUT J K. The design and implementation of a log-structured file system[J]. ACM Transactions on Computer Systems (TOCS), 1992,10(1):26-52.

[4] MAO B, JIANG H, WU S, et al. HPDA: a hybrid parity-based disk array for enhanced performance and reliability[J]. ACM Transactions on Storage (TOS), 2012,8(1):1-20.

[5] WU S, MAO B, CHEN X, et al. LDM: log disk mirroring with improved performance and reliability for SSD-based disk arrays[J]. ACM Transactions on Storage (TOS), 2016,12(4):1-21.

[6] SHIN J Y, BALAKRISHNAN M, MARIAN T, et al. Gecko: contention-oblivious disk arrays for cloud storage[C]// Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13). San Jose, USA: USENIX Association, 2013:285-97.

[7] HAJKAZEMI M H, ASCHENBRENNER V, ABDI M, et al. Beating the I/O bottleneck: a case for log-structured virtual disks[C]// Proceedings of the 17th European Conference on Computer Systems. Rennes, France: Association for Computing Machinery, 2022:628-643.

[8] NA W, MENG X, SI C, et al. A novel network RAID architecture with out-of-band virtualization and redundant management[C]// Proceedings of the 14th International Conference on Parallel and Distributed Systems (ICPADS 2008). Melbourne, Australia: IEEE, 2008:105-112.

[9] 柯剑, 朱旭东, 那文武, 等. 动态地址映射虚拟存储系统[J]. 计算机工程, 2009,35(16):17-19,22.

[10] 那文武, 孟晓烜, 柯剑, 等. BW-VSDS: 大容量、可扩展、高性能和高可靠性的网络虚拟存储系统[J]. 计算机研究与发展, 2009,46(s2):88-95.

[11] KONISHI R, AMAGAI Y, SATO K, et al. The Linux implementation of a log-structured file system[J]. ACM SIGOPS Operating Systems Review, 2006,40(3):102-107.

[12] The Linux Foundation. Linux kernel[EB/OL]. (2023-01-05)[2023-01-05]. https://git.kernel.org/pub/

scm/linux/kernel/git/torvalds/linux. git/.

[13] MIN C, KIM K, CHO H, et al. SFS: random write considered harmful in solid state drives[C]//Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 12). San Jose, USA: USENIX Association, 2012:139-154.

[14] LEE C, SIM D, HWANG J, et al. F2FS: a new file system for flash storage[C]//Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15). Santa Clara, USA: USENIX Association, 2015: 273-286.

[15] 杨勇鹏, 蒋德钧. 一种 wandering B + tree 问题解决方法[J]. 计算机研究与发展, 2023,60(3):539-554.

[16] BITYUTSKIYA B. JFFS3 design issues [EB/OL]. (2005-11-27) [2022-10-15]. http://linux-mtd. infradead. org/tech/JFFS3design. pdf.

[17] 杨勇鹏. SSD 缓存系统的内元数据结构研究与实现[D]. 北京:中国科学院大学,2018.

[18] AXBOE J. Flexible I/O tester[EB/OL]. (2017-09-28)[2023-01-08]. https://github. com/axboe/fio.

[19] Github. Filebench[EB/OL]. (2011-07-09)[2023-02-01]. https://github. com/filebench/filebench.

A consistency model for log-structured block storage system

YANG Yongpeng, JIANG Dejun

(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

(University of Chinese Academy of Sciences, Beijing 100049)

Abstract

With the increase of physical device capacity, the time and space complexity of log-structured block storage system consistency model and crash recovery are increasing. The scope of the consistent state is inversely proportional to the complexity of crash recovery and directly proportional to the probability of conflicts with user write requests. A single consistent timestamp (CST) is proposed to reduce the complexity of crash recovery. Then, the necessary and sufficient conditions and overhead of consistent state generation are analyzed; the write I/O state list (WSL) linked list is introduced to design a consistent state generation algorithm. Building on the consistency state generation algorithm, a multi write I/O mapping table (WMT), metadata management structure——multi consistent state table (MCT) are proposed. As a result, conflicts between user requests and generation of consistent state are reduced to the access of linked list nodes. Based on the consistency model, a crash recovery algorithm is designed, and a log-structured block storage system——single consistent state block device (SCB) is designed and implemented. Experimental results show that compared with allocate ondemand storage device (ASD), SCB's throughput is increased by 135.59%, and 99.90% tail latency is reduced by 42.89%, fileserver workload performance is increased by 25.00%, crash recovery time is 1/23 of ASD. Compared with dm-thin, SCB's throughput is increased by 225.72%, and varmail workload performance is increased by 46.67%.

Key words: log-structured storage system, block storage system, crash recovery, consistent state, consistency model