计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220555
60(3): 539−554, 2023

一种 wandering B+ tree 问题解决方法

杨勇鹏　蒋德钧
（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）
（yangyongpeng18b@ict.ac.cn）

A Method for Solving the wandering B+ tree Problem

Yang Yongpeng and Jiang Dejun
（*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190）
（*University of Chinese Academy of Sciences*, *Beijing* 100049）

Abstract　In order to narrow the gap between the random write and sequential write performance of HDDs and SSDs, file systems and block storage systems usually use the log-structured technique to convert random write to sequential write. Therefore, modifications on log-structured storage system data and metadata are performed as out-of-place writes. In log-structured storage systems, B+ trees are often used to manage metadata. The tree node adopts the out-of-place update method, which will cause the tree node to be updated recursively, so it faces the wandering B+ tree problem. Currently, the main ideas of the existing methods are: The logical index and physical address of the tree node are separated, and a separate data structure and physical device space are used to store the mapping of the logical index and physical address of the tree node, thereby avoiding recursive updating of the tree node. However, the existing schemes not only introduce additional space overhead but also have the problem of non-sequential writing in the additional physical device space. We propose an IBT B+ tree, internal node based translation B+ tree, which embeds the logical index and physical addresses into the tree node. Based on the dirty linked list design, a non-recursive update algorithm for flushing the IBT B+ tree is proposed. The IBT B+ tree not only solves the problem of wandering B+ tree but also does not introduce additional data structure and space overhead. In this paper, the IBT B+ tree and the B+ tree designed by NAT, proposed in F2FS, are implemented respectively. On this basis, the Monty-Dev block storage system is designed and implemented to evaluate the two B+ trees. Experiments show that on HDD and SSD, the IBT B+ tree is better than the NAT B+ tree in both write amplification and flushing efficiency.

Key words　log-structured storage system; block storage system; wandering B+ tree; IBT B+ tree; write amplification

摘　要　为了应对磁盘和固态硬盘随机写和顺序写性能差异较大的问题，文件系统和块存储系统通常采用日志结构（log-structured）技术将随机写转换为顺序写. 因此，对于日志结构存储系统数据和元数据的修改都以异地写的方式执行. 在日志结构存储系统中，B+ tree 常被用于管理元数据，这就会导致 wandering B+ tree 问题，即树结点异地更新会导致树结构递归更新. 目前，现有工作主要通过分离树结点的逻辑索引和物理地址，并使用额外的数据结构和物理设备空间存放树结点逻辑索引和物理地址的映射，从而避免递归更新树结构. 但现有方法既引入额外空间开销，又存在额外物理设备空间非顺序写的问题. 提出 IBT B+ tree，将树结点逻辑索引和物理地址均存放在树结构中. 同时，基于 IBT B+ tree 结构引入 dirty 链表设计，并提出了非递归更新的 IBT B+ tree 下刷算法. IBT B+ tree 既解决了 wandering B+ tree 问题，又不引入额外

收稿日期：2022-06-16；**修回日期**：2023-01-12
通信作者：蒋德钧（jiangdejun@ict.ac.cn）

的数据结构和物理设备空间,消除了固定物理设备空间的非顺序写.分别实现 IBT B+ tree 和基于 F2FS 中 NAT 设计的 B+ tree,在此基础上设计实现 Monty-Dev 块存储系统以评价 2 棵 B+ tree.实验表明,在 HDD 和 SSD 介质上,IBT B+ tree 在写放大和下刷效率方面均优于 NAT B+ tree.

关键词 日志结构存储系统;块存储系统;wandering B+ tree;IBT B+ tree;写放大

中图法分类号 TP391

随着大数据时代的到来和发展,数据量呈现爆炸式增长.根据 IDC(International Data Corporation)的预测,全球数据领域将从 2018 年的 33ZB 增长到 2025 年的 175ZB[1].为了满足数据存储的需求,数据存储系统的规模和存储介质的容量都在不断地增长.各大存储设备厂商纷纷推出大容量存储设备来满足大量数据存储的需求.例如,西部数据已经推出单盘容量达 18TB 的机械硬盘(hard disk drive, HDD)[2],和单盘容量达 20TB 的瓦记录磁盘(shingled write disk, SWD)[3].存储厂商 Nimbus Data 推出单盘最大容量高达 100TB 的固态硬盘(solid state disk, SSD)[4].

对于 HDD,文献 [5] 提到,Unix 文件系统只能利用 5%~10% 的磁盘写带宽,其他时间都消耗在磁盘寻道上.Sprite LFS[5](log-structured file system)采用日志结构(log-structured)技术,该系统假设:文件都缓存在内存中,容量逐渐增大的内存可以很有效地满足读请求的处理需求.基于该假设,Sprite LFS 着重优化写性能.Sprite LFS 的物理地址分配方式为写时分配,文件系统的元数据和数据的更新方式均为异地写(out-of-place write).Sprite LFS 将所有的写请求转换为顺序写后,几乎可以消除所有的磁盘寻道开销.基于 Sprite LFS, NILFS[6] 在 Linux 系统上实现了日志结构文件系统.

对于 SSD,文献 [7–8] 的研究表明:频繁地随机写请求处理,会导致严重的内部碎片,影响 SSD 的持续性能表现.SFS[9] 研究发现:SSD 的随机写带宽和顺序写带宽的差异超过 10 倍;随机写会增加单位写请求的闪存块平均擦除次数,从而减少 SSD 的使用寿命.为解决上述问题,SFS 以 NILFS 为基础针对闪存介质特性,实现新的文件系统,提升系统吞吐率,降低闪存块平均擦除次数,减少 SSD 元数据管理开销,从而提升 SSD 寿命.

近年来,业界又先后推出了 JFFS[10], F2FS[11-14] 等日志结构文件系统.除了文件系统外,日志结构技术还被应用于块存储系统,例如 BW-RAID[15] 存储系统的 ASD 子系统[16-17],ASD 系统向上提供标准块设备接口,将上层写请求转换为本地磁盘 RAID 的顺序写,以提升 BW-RAID 系统的吞吐率.本文把这类系统统称为日志结构存储系统.

虽然日志结构存储系统可以将所有的写请求转换为顺序写,但是因为日志结构存储系统要求数据和元数据的修改都采用异地更新的方式,它面临元数据异地更新带来的连锁更新问题.树状结构通常被用于保存元数据,例如 B+ tree, RB-tree(red-black tree),而一旦树状结构的某个结点执行了异地更新,就会触发递归更新树状结构,这一问题通常被称作 wandering tree 问题[18].因为 B+ tree 被广泛应用于存储系统中,所以本文主要关注 wandering B+ tree 问题.

针对 wandering tree 问题,现有日志结构存储系统均提出了各自的解决方法:例如,NILFS2[19] 使用 B+ tree 管理所有元数据,引入特殊的内部文件 DAT(data address translation file)管理所有 B+ tree 树结点;DAT 文件的每个数据块为 B+ tree 的树结点,在文件内的偏移作为 B+ tree 树结点的逻辑索引(用于索引在内存中的树结点).但是,DAT 文件的地址映射关系仍然使用 B+ tree 维护,以此保存逻辑索引和物理地址的映射(元数据块和数据块在物理设备上的地址),因此当 DAT 文件更新时,DAT 文件的映射管理仍然面临 wandering B+ tree 的问题.

F2FS 采用多级间接索引管理文件映射,同样面临元数据结构递归更新问题.为了缓解这一问题带来的性能影响,F2FS 使用物理设备上固定的 NAT(node address table)[11] 区域存放元数据块逻辑索引和物理地址的映射关系.1 个设备上共有 2 个交替使用的 NAT 区域,目的是为了确保每次下刷 NAT 块均为原子操作.此外,对于采用日志结构技术的块存储系统 ASD 而言,它采用 2 层 RB-tree 管理地址映射,通过固定区域存放地址映射的反向映射来解决 wandering RB-tree 问题.

综上,上述系统均使用额外的数据结构和物理设备空间管理树结点逻辑索引和物理地址的映射,以此解决元数据管理的 wandering tree 问题.但引入额外的数据结构和物理设备空间,会增加系统设计的复杂度、降低写物理设备的连续度.

采用日志结构设计的文件系统和块设备系统在业界均有广泛的应用,然而针对大容量块存储系统,

若采用高扇出、树操作效率高的 B+ tree 管理块设备逻辑地址和物理地址的映射关系，则需要解决 wandering B+ tree 问题，现有的块设备解决方法及日志结构文件系统的解决方法均需要引入额外的数据结构和物理设备空间. 为解决 wandering B+ tree 问题，本文提出 IBT（internal node based translation）B+ tree. 本文的主要贡献有 3 个方面：

1）提出 IBT B+ tree 树结构. 中间结点记录孩子结点的逻辑索引和物理地址，避免引入额外数据结构和物理设备空间.

2）提出 IBT B+ tree 下刷算法. 引入每层 dirty 链表按层次管理所有 dirty 树结点，自底向上按层次下刷 IBT B+ tree，避免递归更新树结构.

3）设计实现 Monty-Dev 块存储系统，评价 IBT B+ tree 在写放大和下刷效率方面的优化效果.

1 相关工作

对于树状数据结构，父亲结点需要记录孩子结点的地址信息. 对于需要持久化到物理设备的树结构，则需要通过在中间结点中记录的孩子结点信息，确定孩子结点的物理地址. 本文将树结点在内存中的唯一标识称作树结点逻辑索引，将树结点在物理设备上的地址称作树结点物理地址. Linux 内核中的 ext4, dm-btree 等模块的树状数据结构，通过中间结点记录孩子结点的物理地址访问孩子结点，因此树结点逻辑索引和物理地址相同. 其中，dm-btree 使用 RB-tree 组织缓存在内存中的树结点，访问孩子结点时首先检查孩子结点是否在 RB-tree 中，如果在则直接访问树结点，否则需要从物理设备加载.

日志结构存储系统要求树结点的更新方式为异地写，且为写时分配物理地址. 如果树结点逻辑索引和物理地址相同，则在树结点下刷时需要递归更新祖先结点中记录的孩子结点物理地址，因此存在 wandering tree 问题. 本节分别介绍 NILFS2, F2FS, ASD 系统针对 wandering tree 问题的解决方法.

1.1 NILFS2

NILFS2 使用 B+ tree 管理文件映射和 inode. 对于 wandering B+ tree 问题，NILFS2 的解决方法是：普通文件映射及管理 inode 的 B+ tree 树结点，由一个文件名为 DAT 的特殊文件管理；所有管理用户文件映射的 B+ tree 树结点是 DAT 文件的一个数据块；DAT 文件的地址映射也由 B+ tree 管理，该 B+ tree 的树结点逻辑索引与物理地址相同.

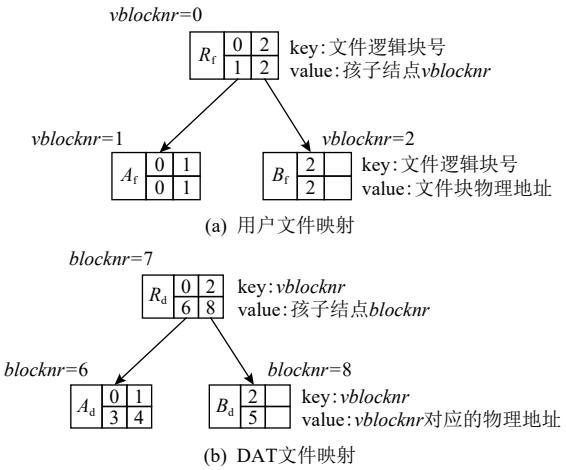


Fig. 1 The case of NILFS2 user file and DAT file mapping [19]

图 1 NILFS2 的用户文件和 DAT 文件映射案例[19]

图 1 介绍 NILFS2 的解决方法.图 1(a)为管理普通用户文件映射的 B+ tree 示意图，树结点逻辑索引为 *vblocknr*，即树结点在 DAT 文件中的偏移；图 1(b)为管理 DAT 文件映射的 B+ tree 示意图，树结点逻辑索引和物理地址为 *blocknr*.

若要读取用户文件逻辑地址为 0 的块，则需要查找图 1(a)所示的 B+ tree，访问结点 R_f 确定下一个要访问的孩子结点 *vblocknr* = 1；若 DAT 文件的第 1 号块不在内存，则通过图 1(b)中所示的 B+ tree 查找映射，自上至下访问结点 A_d，确定 DAT 文件第 1 号块的物理地址为 4，即为结点 A_f 的物理地址；通过查找结点 A_f，可确定用户文件第 0 号逻辑块的物理地址为 0.

图 1(a)中树结点修改下刷只需要修改 DAT 文件，无需递归更新图 1(a)；而图 1(b)中树结点修改下刷时，仍然需要递归更新.

综上，NILFS2 可以解决除管理 DAT 文件映射的 B+ tree 之外所有元数据面临的 wandering B+ tree 问题. 因此，NILFS2 的方法可以部分解决 wandering B+ tree 问题，但仍然存在递归更新 B+ tree 树结点的问题.

1.2 F2FS

F2FS 将底层设备划分为多个 segment，segment 是 F2FS 管理的最小单元. 除物理设备头部固定空间占用外，剩余每个 segment 用来存放数据或者元数据，F2FS 的元数据主要包括 2 类：inode 块和间接索引块. F2FS 通过 inode 块存放文件信息，使用多级间接索引管理文件的地址映射. 多级间接索引有着索引管理简单的优点，但是不抗稀疏，不能支持 Extent. 由于使用了多级间接索引，F2FS 也存在 wandering tree 问题.

F2FS 的解决方法是：引入 NAT 的设计，将文件系统元数据块的逻辑索引和物理地址分离；使用固

定的 NAT 区域存放 NAT ID 和物理地址的映射关系，避免修改一个元数据块就需要递归更新上级元数据块中记录的下级元数据块物理地址；F2FS 在格式化时，按照设备大小计算出管理既定物理设备空间所需的 NAT 块个数上限，NAT 区域中块数量为上限的 2 倍，2 个空间存放 NAT 块的不同版本，交替使用.

NAT 块管理示意图如图 2 所示，图中有 4 个 segment，每个 segment 包括 512 个块，每个块大小均为 4KB；每个块包括 455 个 NAT entry，即 455 个 NAT ID 到物理地址的转换；由于使用固定物理空间存放 NAT，因此可通过每个 NAT entry 相对于 NAT 区域起始地址的差值计算得到 NAT ID；每 2 个连续的 segment 存放 512 个有效 NAT 块，NAT 块更新时写不同的 segment，由 version bitmap 标识某一个 NAT 块的最新版本在哪个 segment 中. 因此，通过 2 个 segment 和 bitmap 的设计，可确保 NAT 块每次更新均为异地写.

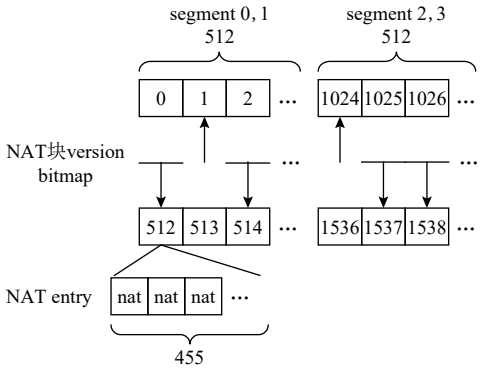


Fig. 2 NAT block and version bitmap[11]

图 2 NAT 块和 version bitmap[11]

随着设备的增大，文件系统的元数据量也在增大，因此 NAT 区域就需要管理更多的 NAT 块. 但 NAT 的随机修改会导致严重的写放大，为解决该问题，引入了 journal 的设计，将最近对 NAT 的修改记录在 CP 区域中，journal 无可用空间时再更新 NAT 并下刷.

综上，F2FS 采用 NAT 和 journal 的设计解决了 wandering tree 问题. 但 NAT 设计无法利用负载的空间局部性，随着 NAT 空间的增大，写放大问题会更严重，且物理设备的访问模式不是顺序写.

1.3 ASD

ASD 系统提供 Linux 标准块设备语义，以虚拟块设备(virtual device, VD)的形式提供存储服务. ASD 系统在物理设备之上创建一个日志结构存储池，存储池之上有多个逻辑地址空间，每个逻辑地址空间对应一个虚拟块设备 VD，每个 VD 向上提供存储服务. 对于块设备，最重要的元数据信息是逻辑地址和物理地址的映射关系.

ASD 的元数据管理结构如图 3 所示. ASD 系统使用 Extent 管理一段逻辑地址连续且物理地址连续的映射关系. 定长逻辑空间内的多个 Extent 组成一个 Subtree，多个 Subtree 组成一个 Group，多个 Group 组成整个逻辑地址空间. Group 信息常驻内存，Subtree 和 Extent 信息在内存中的缓存可在内存压力大的时候释放. 由于 Group 信息与设备容量相关，因此随着设备容量的增大，内存占用也会增大；另外，由于采用 Extent 的方式管理，一个 Extent 作为一个单独的内存分配单元，大量分配 Extent 后回收、释放可能存在内部碎片问题，释放了多个 Extent 但是内存占用却没有减少，且内存占用上限越大内部碎片越严重.

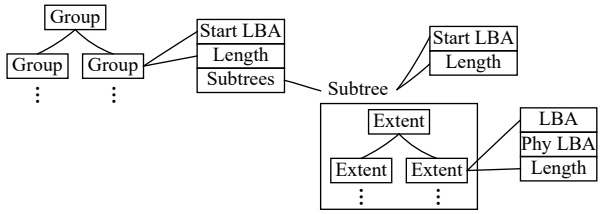


Fig. 3 ASD system metadata structure[16-17]

图 3 ASD 系统元数据结构[16-17]

上述 Extent, Subtree, Group 结构均采用 RB-tree 进行组织，每个 Group 在内存中索引多个 Extent. 对于 Group 信息的下刷，需要先调整树结构，使得一个 Group 索引到的 Extent 信息可以记录到物理设备上一个定长的数据块中，Group 的下刷方式为异地写. 因此，ASD 也存在 wandering tree 的问题，ASD 的解决方法是：

1) Group 信息写到物理设备上之后，将多个 Group 的逻辑索引和物理地址的映射以异地写的方式更新到数据区域；

2) 将 Group 信息的物理地址及其归属信息写入物理设备头部固定的反向表区域；

3) 系统重启时通过扫描固定区域重构 RB-tree.

综上，ASD 采用 2 级逻辑索引和物理地址的映射解决 wandering RB-tree 问题. 该方法的优点是固定位置写较少，但管理复杂度过高.

2 wandering B+ tree 问题分析和结构设计

本节首先分析 wandering B+ tree 问题，提出该问题的解决方法：IBT B+ tree. 分别通过 B+ tree 树结点布局、链表设计和树操作设计，以及第 3 节提出的 IBT

B+ tree 下刷算法论述 wandering B+ tree 的解决方法.

2.1 wandering B+ tree 问题分析

首先，通过图 4 介绍 wandering B+ tree 问题，并明确解决问题的难点. 图 4 展示了 1 棵树结点逻辑索引与物理地址相同的 3 阶 B+ tree 树结点更新过程. 图 4 左图下刷结点 A 时，需要为结点 A 分配新的物理地址生成结点 A_1；图 4 中间的图，生成结点 A_1 后，分配结点 D_1 使其指向结点 A_1；图 4 右图分配结点 R_1 同理. 因此，非树根结点分配物理地址需要递归更新父亲结点至树根结点，严重影响 B+ tree 下刷效率.

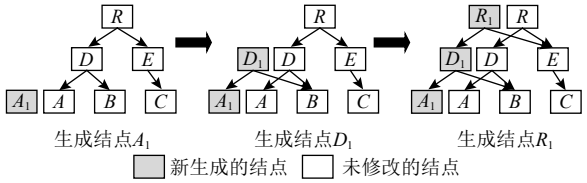


Fig. 4 Example of wandering B+ tree problem

图 4 wandering B+ tree 问题案例

综上，要解决 wandering B+ tree，需要满足日志结构系统设计需求：树结点写时分配物理地址，且更新方式为异地写；同时，避免 B+ tree 树结点下刷时递归更新非叶子结点.

如第 1 节所述，针对 wandering tree 问题，NILFS2, F2FS, ASD 解决方法的共同点是：树结点逻辑索引和物理地址不同，即树结点的逻辑索引和物理地址分离，额外的数据结构和物理设备空间维护树结点逻辑索引和物理地址的映射关系.

因此，解决 wandering B+ tree 问题的难点是：

1）支持树结点逻辑索引和物理地址分离，且不引入新的数据结构和物理设备空间，降低复杂度；

2）避免 B+ tree 下刷时递归更新树结构.

针对这 2 个难点，2.2 节提出第 1 个难点的解决方法，2.3 节和 2.4 节支撑第 3 节提出的 IBT B+ tree 下刷算法解决第 2 个难点.

2.2 IBT B+ tree 树结点布局

IBT B+ tree 的中间结点和叶子结点结构图如图 5 所示，图 5 中各字段的含义如表 1 所示. 接下来，分别介绍表 1 中各字段的用途.

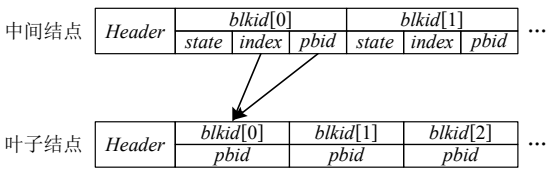


Fig. 5 The internal node and leaf node of IBT B+ tree

图 5 IBT B+ tree 中间结点和叶子结点

Table 1 The Meaning of the Fields in Fig.5

表 1 图 5 各字段的含义

字段	含义
blkid	块设备的逻辑地址
state	孩子结点状态，dirty/clean
index	树结点逻辑索引
pbid	物理地址
Header	树结点的汇总信息

1）*state* 字段为孩子结点的状态，dirty 代表孩子结点需要持久化到物理设备，clean 代表孩子结点已经持久化且可以在内存紧张时回收. 因此，*state* 仅描述内存中的树结点内容是否与设备上的一致.

2）如图 5 所示，中间结点记录孩子结点的逻辑索引 *index*，随着树高的增长，所有非树根结点的 *index* 和 *pbid* 都会存放在其父亲结点中，只有树根结点的 *index* 和 *pbid* 不存放在树结构中. 因此，需要额外的数据结构和物理设备空间存放树根信息，记作 Superblock，包括 *index*, *pbid*, *height*（见 2.4 节）. 由于所有树结点的更新方式均为异地写，因此树根结点需要记录在物理设备的固定位置上，B+ tree 初始化时才能找到树根.

3）中间结点记录孩子结点的 *index* 和 *pbid*，在内存中的 B+ tree 树结点，以 *index* 为关键字，使用红黑树进行组织. 各类 B+ tree 操作执行过程中，如果树结点不在红黑树中，则需要通过 *pbid* 从底层设备读取树结点.

2.3 IBT B+ tree 链表设计

2.2 节提出树结点逻辑索引和物理地址分离的方法，通过树结点逻辑索引管理内存中的树结点，通过物理地址加载不在内存中的树结点，从而支持写时分配物理地址. 因此，需要在此基础上解决 2.1 节提出的第 2 个难点：设计非递归更新的 IBT B+ tree 下刷算法. 主要思路是：按照层次管理 B+ tree dirty 树结点，使得 IBT B+ tree 能够按照层次进行下刷. 本节主要论述按照层次管理 dirty 树结点的方法，IBT B+ tree 下刷算法在第 3 节论述.

扩展并修改 B+ tree 链表维护方法. 传统 B+ tree 只维护叶子结点链表，用来管理所有叶子结点，按照关键字升序或降序排列. 本文修改并扩展树结点链表：B+ tree 的每一层均有一个 dirty 链表；引入全局 clean 链表. 2 类链表设计主要有 2 个特点：

1）B+ tree 各层 dirty 链表管理相应层中状态为 dirty 的树结点，dirty 树结点按照 LRU 的方式管理，而

非按照关键字排序;

2) clean 链表管理所有状态为 clean 的树结点, clean 链表也采用 LRU 的方式管理, 目的是为了在内存紧张时释放不经常访问的树结点.

链表的使用, 特别是 dirty 链表, 将在 2.4 节介绍. 第 3 节将介绍如何通过 dirty 链表更新树结点物理地址.

2.4 IBT B+ tree 基本操作设计

B+ tree 在插入、查找、删除操作中对 key-value 的管理, 与采用 shadow-clone 设计的 B+ tree[20-21] 没有差异. 为使用 lock-coupling 细粒度加锁协议支持 B+ tree 并发操作[22], B+ tree 采用预先 rebalance、分裂的方式调整树结构, 避免 B+ tree 树操作执行过程中对孩子结点的修改导致父亲结点 rebalance、分裂.

对于 wandering B+ tree 问题的解决方法, 2.2 节已经给出支持树结点逻辑索引和物理地址分离的树结构设计. 本节主要从支撑写时分配物理地址、更新物理地址的角度进行论述. 支撑上述特性的主要解决方法是 dirty 和 clean 链表设计. 本节从 B+ tree 树操作的角度介绍 2 类链表的管理. 由于 clean 链表用来管理 clean 树结点、支撑内存回收, 只需要维护树结点的 LRU 逻辑即可. 因此, 本节主要介绍 dirty 链表的管理.

dirty 链表的变化主要源自插入操作和删除操作. 下面分别通过案例介绍这 2 个操作中 dirty 链表的管理. 本节所有的案例均围绕图 4 所示的 3 阶 B+ tree 进行介绍, B+ tree 树结点的 key-value 按照 key(逻辑地址)升序排序.

2.4.1 插入操作

图 6 为将〈10, 11〉映射插入 1 棵 3 阶 B+ tree 的主要步骤. 下面分步介绍插入操作:

1) 初始状态如图 6(a)所示, 所有树结点状态均为 clean.

2) 图 6(b)将结点 *R* 标记为 dirty, 并将结点 *R* 插入 *list*[0]; 由于结点 *R* 的 key-value 数量达到上限, 因此需要分裂结点 *R*, 分配 *D* 和 *E* 这 2 个结点. 将结点 *R* 的 key-value 平均分配给结点 *D* 和 *E*. 将结点 *D* 和 *E* 中记录的最小 key 与结点 *D* 和 *E* 的逻辑索引分别作为 key 和 value 插入到结点 *R* 中. 此时不需要为结点 *D* 和 *E* 分配物理地址. 树高加 1, 相应地调整链表.

3) 图 6(c)和图 6(d), 逐层向下访问到结点 *A*, 将〈10, 11〉插入到结点 *A* 中.

如图 6(b)所示, 新创建的链表为 B+ tree 的第 2 层. 因为除树根结点外, B+ tree 其他任何层都可能存在兄弟结点, 为降低 B+ tree 树操作复杂度, 仅允许树根结点分裂出孩子结点. 综上, 新插入的 dirty 链表只发生在第 2 层, 这一特性适用于任何高度大于 1 的 B+ tree.

图 6 所示的每个阶段操作过程中, 都将所有需要访问的树结点状态标识为 dirty, 同时除树根结点外, 将所有 dirty 树结点在其父亲结点中的状态标识为 dirty.

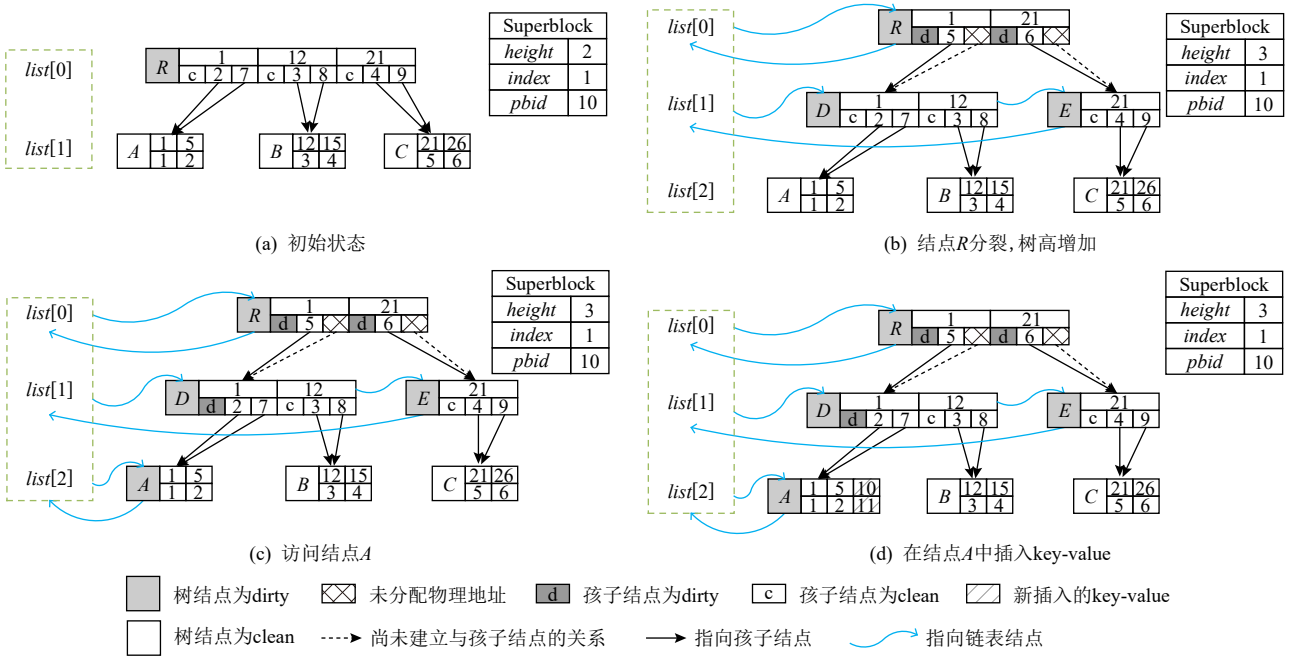


Fig. 6 Insert operation (insert〈10, 11〉)

图 6 插入操作(插入〈10, 11〉)

由于 dirty 链表的引入，2 个并发的插入操作①和②，①先执行，②后执行；如果插入②导致树高加 1，①获得的树高信息不正确，则树结点可能插入错误的链表. 因此，B+ tree 无法支持插入和插入的并发. 由于 clean 链表对树高不敏感，B+ tree 仍可支持插入和查找操作并发执行.

2.4.2　删除操作

以图 7(a)所示的 3 阶 B+ tree 为例，介绍删除〈10, 11〉映射：

1）访问树根结点 *R*，将结点 *R* 标记为 dirty，并加入到 dirty 链表 *list*[0] 中.

2）图 7(b)，结点 *R* 只有 1 个孩子. 因此，将结点 *D* 中记录的 key-value 拷贝到结点 *R* 中，删除结点 *D* 和相应的链表. 同时，结点 *R* 继承结点 *D* 的类型，图 7(b)中 *D* 为中间结点，也存在结点 *R* 只有 1 个孩子且为叶子结点的情况. 因此，结点 *R* 也可能成唯一的叶子结点.

3）图 7(c)和图 7(d)，逐级向下访问到结点 *A*，将〈10, 11〉从结点 *A* 中删除.

如图 7(b)所示，被删除的 dirty 链表为 B+ tree 的第 2 层. 与插入操作同理，dirty 链表删除操作仅删除第 2 层链表，这一特性适用于任何高度大于 1 的 B+ tree.

删除操作可能引起树高的修改，因此删除操作与删除操作、删除与插入操作均不能并发执行.

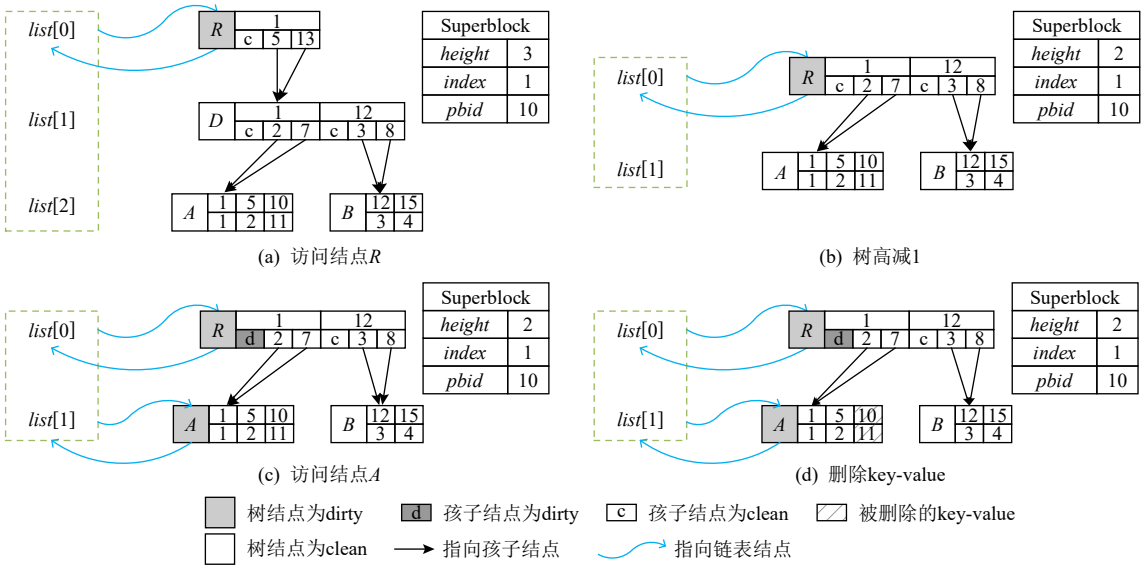


Fig. 7　Remove operation（remove〈10, 11〉）

图 7　删除操作（删除〈10, 11〉）

3　IBT B+ tree 下刷

本节针对 2.1 节提到的第 2 个难点，提出非递归更新的 IBT B+ tree 下刷算法，从而解决 wandering B+ tree 问题. 通过第 2 节中 IBT B+ tree 的相关介绍可知，IBT B+ tree 除树根结点外有如下特性：所有状态为 dirty 的树结点的父亲结点状态均为 dirty；dirty 结点的状态对父亲结点不透明. 利用上述 2 个特性，下面分别通过下刷算法和具体案例介绍 IBT B+ tree 的下刷.

B+ tree 下刷存在 2 种情况：下刷所有树结点，包括 B+ tree 创建完成并插入多次关键字之后的状态和 B+ tree 非首次下刷且所有树结点状态均为 dirty；部分下刷，包括部分 B+ tree 树结点为 dirty 的情况和经过多次删除后 B+ tree 为空.

多次插入、删除操作后，dirty 树结点的内存占用或下刷时间间隔达到一定阈值，需要将 B+ tree 所有状态为 dirty 的树结点下刷到物理设备.

为避免递归更新 B+ tree 树结构，IBT B+ tree 下刷采用 2 阶段提交的方式.

第 1 阶段，自 dirty 链表的倒数第 2 层向上下刷所有的 dirty 链表：

1）根据该 dirty 链表上所有树结点，下刷该树结点所有状态为 dirty 的孩子；

2）为孩子结点分配物理地址并下刷，将孩子结点的物理地址记录到父结点中，并更新父结点中记录的状态.

第 2 阶段，将 Superblock 写到固定的物理设备空间.

B+ tree 下刷算法如算法 1 所示：

算法 1. IBT B+ tree 下刷算法.

① procedure *flush_dirty_nodes*(*A*)
② for *child* ← *A* do
③ if *is_dirty*(*child*) then
④ *write*(*child*);
⑤ 更新结点 *A* 中 *child* 的物理地址;
⑥ end if
⑦ end for
⑧ end procedure
⑨ procedure *flush_dirty_lists*(*tree*)
⑩ *i* ← *depth*(*tree*);
⑪ while *i*−1 > 0 do
⑫ for *A* ← *dirty_list*[*i*−1] do
⑬ *flush_dirty_nodes*(*A*);
⑭ end for
⑮ *i*−−;
⑯ end while
⑰ end procedure
⑱ procedure *flush_btree*(*tree*)
⑲ *flush_dirty_lists*(*tree*);
⑳ *write*(*root*); /*第 1 阶段提交*/
㉑ 将树根的物理地址记录在 *superblock* 中;
㉒ *write*(*superblock*); /*第 2 阶段提交*/
㉓ end procedure

图 8 展示了 B+ tree 下刷的重要阶段. 接下来, 以图 6(d)所示的 3 阶 B+ tree 为例, 结合算法 1 和图 8 介绍 B+ tree 的下刷.

1) 对于图 6(d)的 B+ tree, 调用算法 1 中的函数 *flush_btree* 下刷整棵 B+ tree. *flush_btree* 调用 *flush_dirty_lists* 自 *list*[1] 开始至树根链表, 下刷所有 dirty 链表.

2) 如图 8(a)所示, 对于 *list*[1] 的结点 *D*, 调用函数 *flush_dirty_nodes*, 结点 *D* 只有一个状态为 dirty 的孩子结点 *A*. 因此, 如图 8(b)所示, 为结点 *A* 分配物理地址并记录在结点 *D* 中, 同时将结点 *D* 中记录的结点 *A* 状态标更新为 clean. 同样的方式处理 *list*[1] 上所有树结点.

3) 图 8(c)下刷 *list*[0], 为结点 *D* 和 *E* 分配物理地址并下刷. 至此, 函数 *flush_dirty_lists* 执行完成.

4) 图 8(d)为树根结点 *R* 分配物理地址 15, 写树根结点 *R* 并更新 Superblock, 完成第 1 阶段提交.

5) 下刷 Superblock 完成第 2 阶段提交.

4 IBT B+ tree 评价系统设计实现

本文以日志结构块设备系统为背景, 在第 2 节和第 3 节提出了 IBT B+ tree, 解决了 wandering B+ tree 问题. 但是, 仅有 IBT B+ tree 只可进行树操作测试, 难以测试在不同负载下的表现.

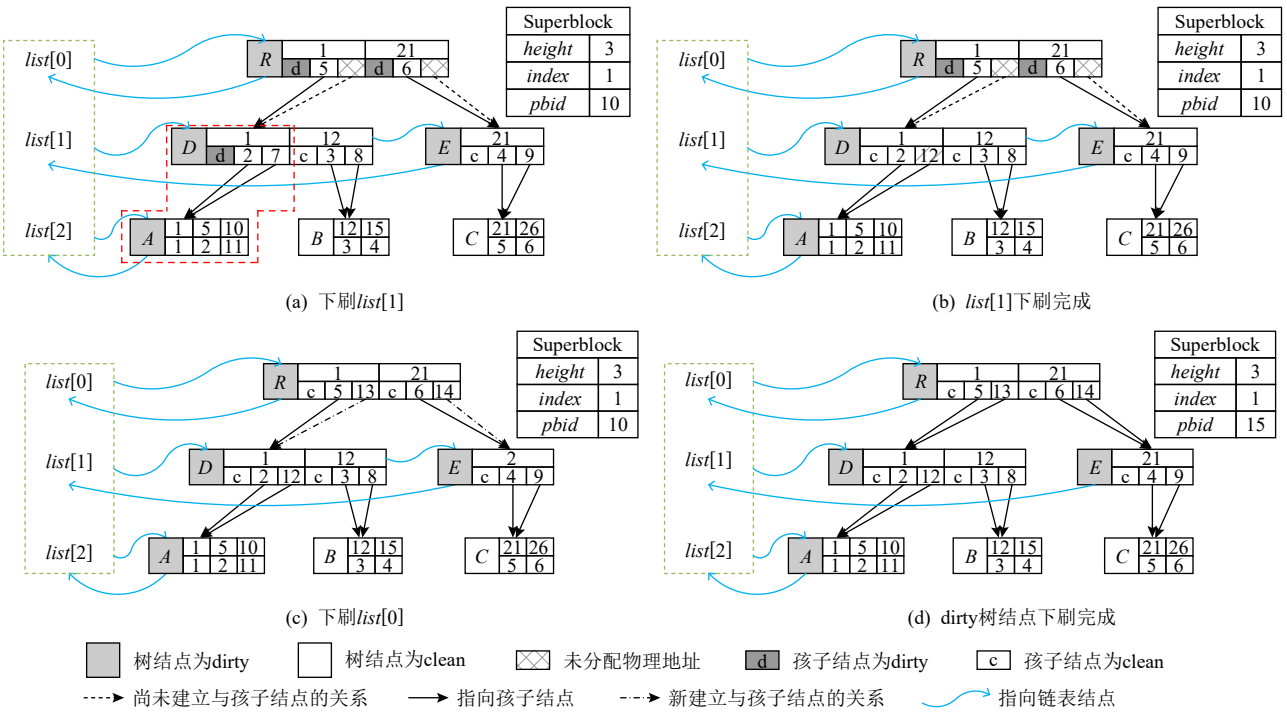


Fig. 8 Flush dirty node of IBT B+ tree

图 8 下刷 IBT B+ tree 的 dirty 结点

为全面评价 IBT B+ tree 在 Fio 和 Filebench 场景下的写放大和下刷效率表现，需要设计实现能够覆盖 IBT B+ tree 支持的各类树操作和下刷，且避免 I/O 上下文干扰的日志结构块存储系统 Monty-Dev.

为对比评价第 3 节所提出解决方法的效果，拟将 IBT B+ tree 与采用 NAT 方案设计的 B+ tree(后文称作 NAT B+ tree)进行对比. 其中，NAT 方案设计参考 Linux 内核 5.14 版本[19]F2FS 的实现. 由于 NAT 块通过 F2FS 内部 inode 以文件映射的方式管理，因此这部分不能完全复用，转而使用 radix-tree 管理所有缓存的 NAT 块，基本与 F2FS 的实现方式等价；其余部分均参考 F2FS. 另外，对于 NAT 块的 version bitmap 需要为每个树结点预留 1 b，计算方法与 f2fs-tools[23] 中格式化操作的计算方法不同.

本文以 Linux 系统为平台，设计实现 Linux 内核标准块设备 Monty-Dev 系统. 下面分别介绍评价系统 Monty-Dev 的总体设计、元数据管理和系统 layout.

4.1 Monty-Dev 系统整体架构

该设计暂不考虑宕机恢复和下刷数据块的反向映射信息，B+ tree 树结点和 NAT 块均缓存在内存中，本文聚焦写放大和固定物理设备空间非顺序写开销的评价. 为评价删除操作的影响，Monty-Dev 系统支持 Linux 内核中通用块层的 Discard I/O 语义.

Monty-Dev 系统总体架构如图 9 所示. 使用横向的虚线标识系统的分层，下面逐层介绍. “用户接口”用来管理 Monty-Dev 系统，比如创建、删除设备和修改系统配置等. 其他系统可通过“标准块设备接口”，直接向 Monty-Dev 提交读写 I/O 请求. ext4/xfs 等文件系统实例，可向通用块层提交读写 I/O、Discard I/O 请求，Discard I/O 用来支持文件系统的 trim 操作.

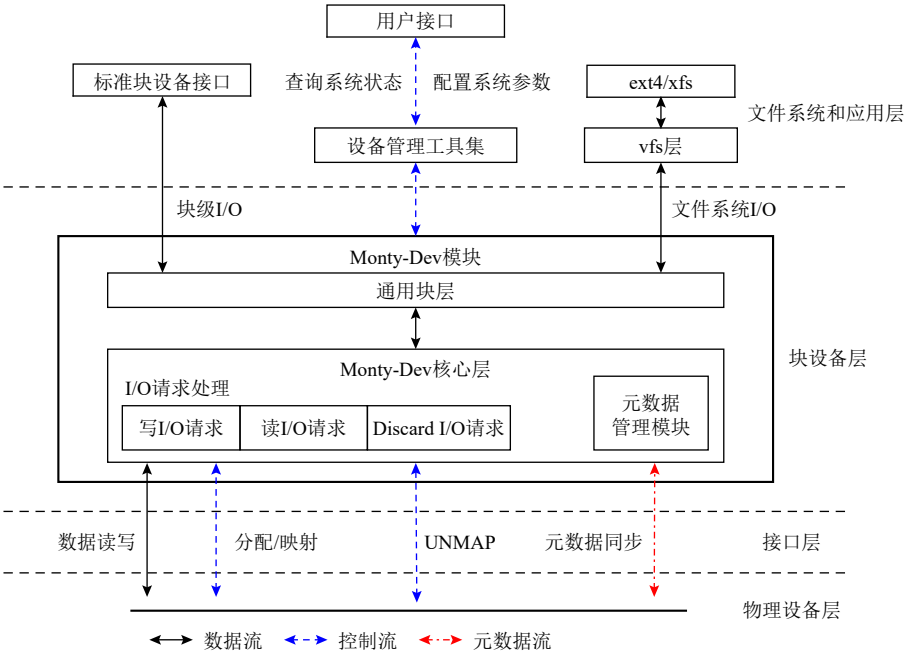


Fig. 9 Architecture of Monty-Dev system

图 9 Monty-Dev 系统整体架构

Monty-Dev 核心模块实现了通用块层的部分接口，处理上层发往通用块层的请求. 基于 Monty-Dev 系统可执行 Fio 测试，将 Monty-Dev 设备格式化为特定文件系统可支持 Filebench 测试.

因此，通过 Monty-Dev 系统测试 Fio 和 Filebench 负载，可覆盖 B+ tree 的查找、插入、删除操作及其并发，以及元数据下刷.

4.2 元数据管理模块

Monty-Dev 的元数据管理模块结构图如图 10 所示，后文称作 Meta tree.

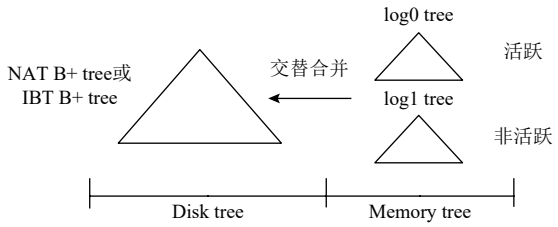


Fig. 10 Structure diagram of Meta tree

图 10 Meta tree 结构图

Meta tree 管理块设备逻辑地址和物理地址的映射关系，支持系列元数据操作，如查询、插入、更新、

UNMAP. Meta tree 包括 2 部分，为方便描述分别称作 Disk tree 和 Memory tree，其中 Disk tree 为 NAT B+ tree 或 IBT B+ tree，本文评测中树结点大小和数据块大小均为 4KB.

仅通过 IBT B+ tree 或者 NAT B+ tree 管理元数据，元数据下刷会阻塞 I/O 回调时对元数据的修改，影响数据和元数据写并发，无法评价元数据下刷对用户 I/O 的影响. 因此，本文引入 Memory tree，包括 log0 tree 和 log1 tree，统称为 log tree. log tree 采用文献 [24] 提出的基于 lock-coupling 扩展协议的 B+ tree，2 棵树常驻内存，仅记录映射关系修改的 log，用来聚合 log(映射关系的修改，包括插入、更新、UNMAP). 2 棵树交替使用，活跃 log tree 处理元数据修改，非活跃 log tree 进行合并. Disk tree 需要下刷到物理设备，存储所有映射关系.

log tree 功能需求有：

1) 将上层写 I/O 对元数据的修改以 log 的方式记录下来，记录的内容为逻辑块号和对映射关系的修改操作.

2) log tree 的内存占用达到一定阈值或其他条件触发下刷，如设备关闭、下刷超时. Meta tree 下刷时需要遍历 log tree 中所有 log 将映射关系的修改按照 log 类型在 Disk tree 上执行相应的树操作. 合并结束后，删除 log tree.

3) 查询操作时，先查询 log tree 再查询 Disk tree，因为最新的映射关系修改都记录在 log tree 中.

综上，log tree 需要提供如下功能：插入、更新、遍历. 这些需求文献 [24] 提出的 B+ tree 都可以满足，且有着映射关系插入、查找可并发的优点.

通过 Meta tree 管理元数据，解决 B+ tree 下刷时不能更新映射关系的问题，从而避免元数据下刷阻塞用户 I/O 请求，影响系统整体性能表现，同时也避免了 I/O 上下文对元数据下刷的干扰.

4.3 Monty-Dev 系统 layout

使用 IBT B+ tree 管理逻辑地址和物理地址的映射关系，则需要预留空间满足第 3 节中所述的 2 阶段提交下刷 B+ tree 的需求，以及存储块设备系统的必要信息.

使用 NAT B+ tree，则需要 NAT 区域和 Checkpoint 区域：NAT 区域与 F2FS 中 NAT 区域的功能相同；Checkpoint 区域记录 NAT 块的 version bitmap.

5 测评与分析

本节基于第 4 节提出的分别采用 NAT B+ tree 和 IBT B+ tree 管理元数据的 2 个版本 Monty-Dev 系统（为方便描述，后文分别称作 NAT 版本和 IBT 版本）进行评测. 本节主要通过 Fio[25] 和 Filebench[26] 进行测试，以评价在不同负载下，NAT 版本和 IBT 版本在元数据写放大和下刷效率方面的差异.

5.1 测试环境

第 5 节的相关测试均在表 2 所示的测试环境上进行. 5.2 节和 5.3 节的测试均分别在 SSD 和 HDD 上进行测试，以验证 IBT B+ tree 在 2 种介质上的优化效果.

Table 2 The Hardware Configuration of Testing Server

表 2 测试服务器硬件配置

类别	参数
CPU	Intel（R）Xeon（R）E5645@2.40GHz，2 路，24 线程
内存	Ramaxel DDR3，32GB，1333MHz
磁盘	Seagate Constellation ES ST1000NM0011，1TB，SATA
SSD	Intel SSD DC P3700 Series，400 GB，NVMe

测试的软件环境如表 3 所示. 由于 NAT 版本和 IBT 版本的元数据构成存在差异：NAT 版本的元数据包括 NAT 块和 B+ tree 树结点，而 IBT 版本仅包括 B+ tree 树结点. 因此，NAT 版本和 IBT 版本在相同测试用例下，元数据的下刷量可能存在差异. 此外，NAT 版本存在固定物理设备空间随机写的问题，而 IBT 版本不存在. 因此，元数据下刷效率也可能存在差异.

Table 3 The Software Configuration of Testing Server

表 3 测试服务器软件配置

类别	版本
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

5.2 Fio 测试

2 个版本的 Monty-Dev 系统，采用精简配置设计，读测试和读写混合测试均存在空读的问题，且 B+ tree 树结点均缓存在内存中，IBT B+ tree 和 NAT B+ tree 的查询效率没有差异，因此仅测试 Fio 随机写.

Fio 测试的参数如表 4 所示，以表 4 中参数连续测试 2 次，分别测试首次写和覆盖写的吞吐率表现，测试 3 次取平均值. 由于 Fio 随机写测试发送的 I/O 请求为伪随机请求序列，因此在参数不变的情况下每次 Fio 测试的 I/O 序列均相同. 第 2 轮测试即为覆盖写，覆盖写测试对于 IBT B+ tree 和 NAT B+ tree 而言均为插入更新操作. 因此，该测试用例可先后评价 NAT 版本和 IBT 版本在插入和插入更新场景下的差异.

Table 4 Fio Configuration Parameters

表 4 Fio 配置参数

参数类别	参数取值
设备大小/TB	4
iodepth	512
粒度/KB	4
engine	libaio
读写	randwrite
数据量/GB	200

Meta tree 合并下刷频率受 2 个因素制约：log tree 内存占用上限；B+ tree 的 dirty 树结点内存占用上限. 其中，log tree 内存占用上限越小，下刷频率越高；dirty 树结点内存占用上限影响一轮元数据下刷量，上限越小，一轮元数据下刷量越小，下刷越频繁. 因此，本文分别测试 log tree 内存占用上限为 10 MB, 20 MB, 30 MB 的情况下，dirty 树结点内存占用上限为 65 MB, 85 MB, 105 MB 时，NAT 版本和 IBT 版本首次写和覆盖写的吞吐率表现和元数据下刷的写放大. 下面分别在 SSD 和 HDD 上进行测试分析.

由于 NAT 版本和 IBT 版本的有效元数据量相同，因此本节通过元数据下刷量评价元数据下刷写放大.

5.2.1 基于 SSD 的 Fio 测试

Fio 测试中，首次写和覆盖写的吞吐率、元数据下刷量对比如图 11 所示.观察分析图 11，可得出 4 个结论：

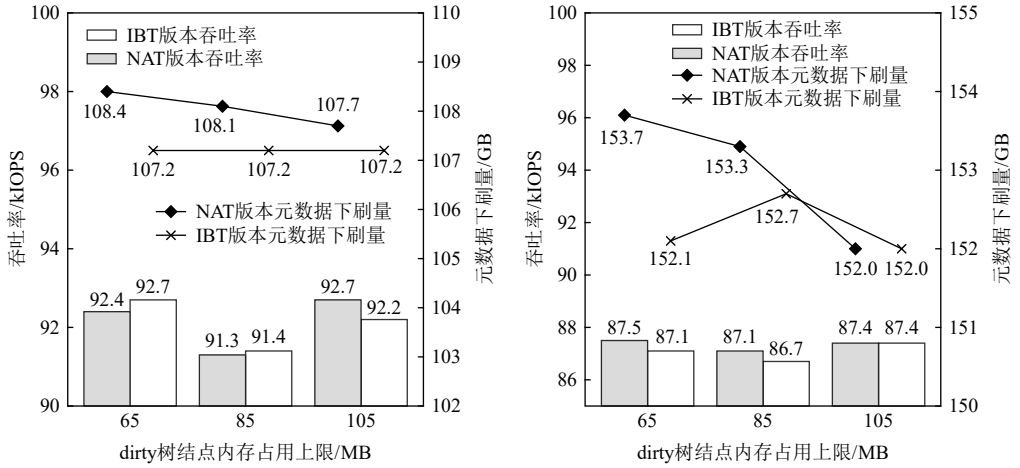
1）对于元数据下刷量，IBT 版本在大多数情况下均优于 NAT 版本，但差异很小；

2）log tree 内存占用上限越大，元数据下刷量越小；

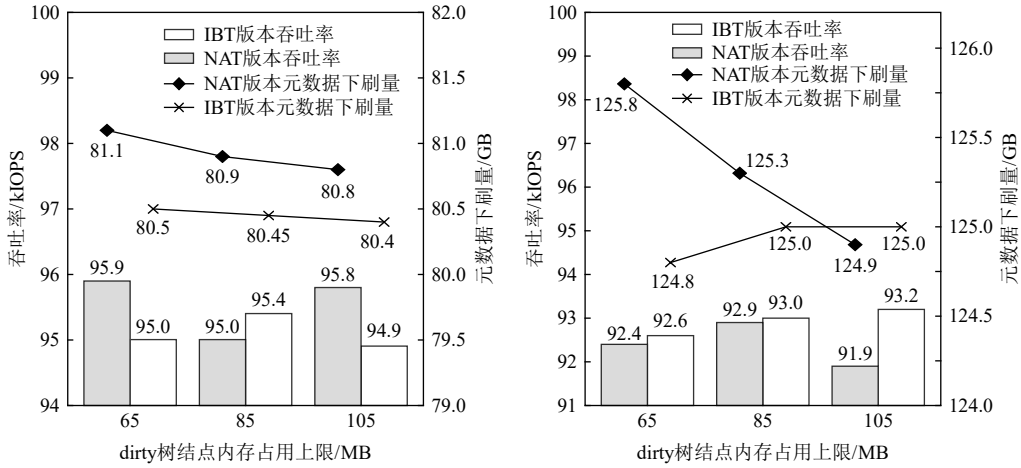
3）NAT 版本和 IBT 版本，在覆盖写测试中 log tree 内存占用上限越大，吞吐率越大；

4）log tree 大小相同时，dirty 树结点内存占用上限对元数据下刷量影响很小.

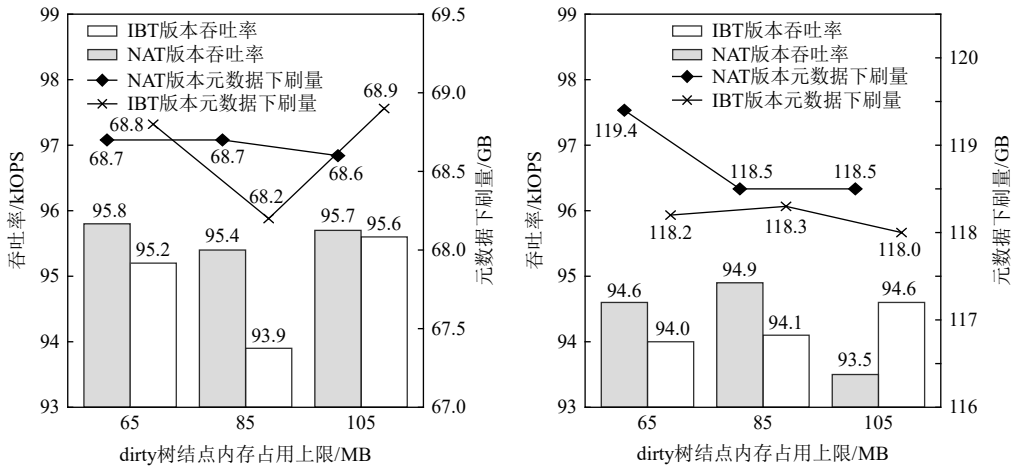
对于第 4 个结论，若在 Meta tree 合并过程中，dirty 树结点内存占用超过上限，需要下刷后再合并剩余映射. 由于 log tree 将映射关系按照逻辑地址排序，下刷后再合并剩余映射时，不会修改已下刷叶子结点，但可能会修改部分已下刷的中间结点. 而中间结点总量较少，因此 dirty 树结点的内存占用上限对元数据下刷量影响很小.



(a) log tree内存占用上限为10MB时的首次写（左）和覆盖写（右）



(b) log tree内存占用上限为20MB时的首次写（左）和覆盖写（右）



(c) log tree内存占用上限为30MB时的首次写（左）和覆盖写（右）

Fig. 11 Comparison of throughput and the total amount of flushed metadata based on SSD

图 11 基于 SSD 的吞吐率、元数据下刷量对比

由于 NAT 版本和 IBT 版本 B+ tree 插入更新复杂度基本相同，因此只需要考虑 Meta tree 合并下刷时的差异. 由于吞吐率表现基本相同，因此 NAT 版本和 IBT 版本的元数据下刷效率对性能的影响基本相同. 为此，本节主要通过分析元数据下刷量评价 NAT 版本和 IBT 版本的元数据写放大. 表 5 统计了图 11(a)中，dirty 树结点内存占用上限为 85 MB 时，2 轮测试期间的元数据下刷量.

Table 5 Total Amount of 2 Versions Flushed Metadata Under Fio Test Based on SSD

表 5 基于 SSD 的 Fio 测试中 2 个版本元数据下刷总量 GB

轮次	NAT 版本	IBT 版本
首次写	108.1	107.2
覆盖写	153.3	152.7

从表 5 可以看出，IBT 版本的元数据下刷量略低于 NAT 版本，但 NAT B+ tree 下刷时存在非顺序写. NAT 版本的元数据包括 NAT 块和 B+ tree 树结点，NAT 块下刷的物理设备访问模式为固定物理设备空间的非顺序写，而数据写为顺序写，NAT 块和数据可并发写. 2 轮测试中 NAT 块下刷情况如表 6 所示.

Table 6 Statistics of Flushed NAT Block Under Fio Test Based on SSD

表 6 基于 SSD 的 Fio 测试中 NAT 块下刷统计

轮次	NAT 块下刷量/GB	比例/%
首次写	1.66	1.5
覆盖写	3.8	2.5

覆盖写测试中 NAT 块的写入量和比例都有所增加. 主要原因是 B+ tree 树结点总量增多，导致 NAT 块总数增多，随机修改 NAT 块的范围增大. 但由于 NAT 块写相对于顺序写的比例较小，对用户数据写性能影响较小，所以 NAT 版本和 IBT 版本的吞吐率差异很小. 然而，随机写会导致闪存块碎片增多，降低闪存性能和寿命，这在短期测试中不能体现出来.

NAT B+ tree 下刷需要下刷叶子结点和 NAT 块，IBT B+ tree 对于状态为 dirty 的叶子结点，其查询路径上所有的祖先结点一定为 dirty，而 IBT 版本的元数据下刷量小于 NAT 版本. 因此，IBT 版本虽然理论上存在一定的写放大，但由于 NAT 块的写开销，IBT 版本实际表现与 NAT 版本相当或优于 NAT 版本.

5.2.2 基于 HDD 的 Fio 测试

Fio 测试经过首次写和覆盖写 2 轮测试后，吞吐率、元数据下刷量对比如图 12 所示.观察分析图 12，可得出 5 个结论：

1）对于元数据下刷量，IBT 版本在大多数情况下均优于 NAT 版本；

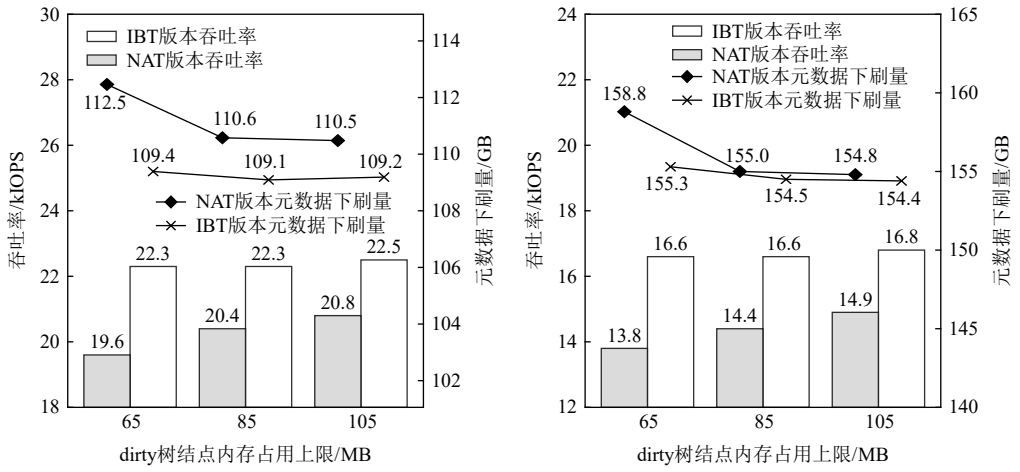
2）对于吞吐率，相对于 NAT 版本，IBT 版本首次写提升 4.7%~13.8%，覆盖写提升 10.7%~20.3%，内存占用越少提升越大；

3）IBT 版本吞吐率表现几乎不受 dirty 树结点内存占用上限影响，而 NAT 版本受影响较大，特别是图 12(b)；

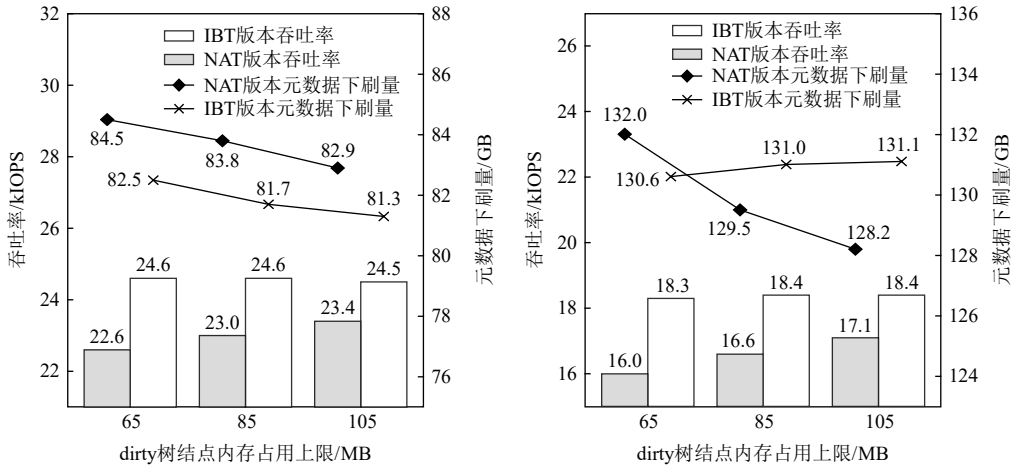
4）log tree 内存占用上限越大，吞吐率越大；

5）log tree 内存占用上限相同，大多数情况下，dirty 树结点内存占用上限越大，元数据下刷量越小.

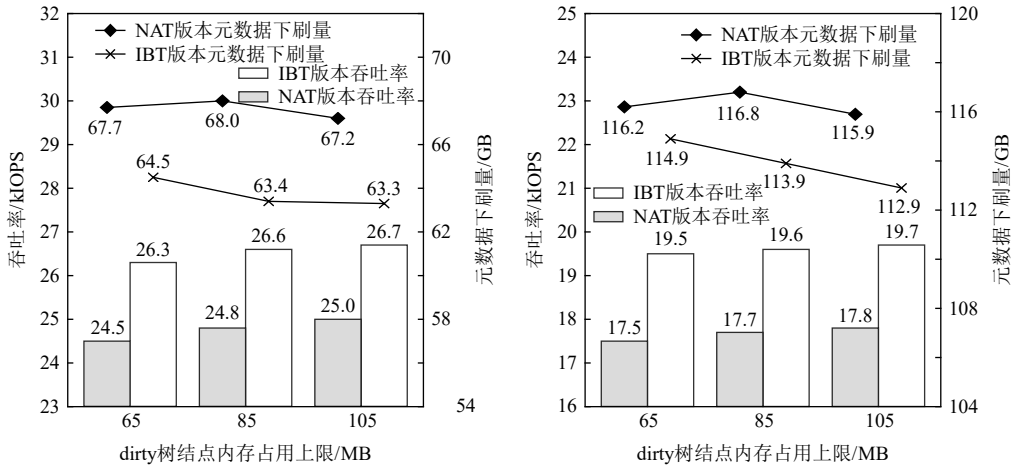
以图 12(a)dirty 树结点内存占用上限为 85MB 为例，分析元数据下刷与吞吐率的关系. 相对于 NAT 版



(a) log tree内存占用上限为10MB时的首次写（左）和覆盖写（右）



(b) log tree内存占用上限为20MB时的首次写（左）和覆盖写（右）



(c) log tree内存占用上限为30MB时的首次写（左）和覆盖写（右）

Fig. 12 Comparison of throughput and the total amount of flushed metadata based on HDD

图 12 基于 HDD 的吞吐率、元数据下刷量对比

本, IBT 版本 2 轮测试吞吐率提升分别为 9.3% 和 15.3%. 相较于 5.2.1 节中基于 SSD 的测试, 提升比例更高. 同 5.2.1 节, 需要关注 Meta tree 合并下刷对性能造成的影响. 2 轮测试期间元数据下刷量如表 7 所示.

IBT 版本比 NAT 版本元数据下刷量略小, 但由于 NAT 块下刷不是顺序写, 且 HDD 的吞吐率受 I/O 连续度的影响较 SSD 更大, 少量的非顺序写也会造成磁头的抖动, 因此导致吞吐率提升比例更大. 表 8

Table 7 Total Amount of 2 Versions Flushed Metadata Under Fio Test Based on HDD

表 7 基于 HDD 的 Fio 测试中 2 个版本元数据下刷总量 GB

轮次	NAT 版本	IBT 版本
首次写	110.6	109.1
覆盖写	155.0	154.5

Table 8 Statistics of Flushed NAT Block Under Fio Test Based on HDD

表 8 基于 HDD 的 Fio 测试中 NAT 块下刷统计

轮次	NAT 块下刷量/GB	比例/%
首次写	1.8	1.6
覆盖写	4.0	2.5

中展示了 2 轮测试中 NAT 块下刷情况.

如表 8 所示, 覆盖写测试中 NAT 块的写入量和比例都增加了, 从而导致固定位置非顺序写请求增多. IBT B+ tree 叶子结点需要下刷, 其父亲结点必须下刷, 对于局部性比较好的树操作, 多个被修改的叶子结点可能有相同的祖先结点. 而 NAT 块的设计不确保 1 个块涵盖的多个树结点在一轮元数据下刷时同时下刷, 所以局部性较好的树操作不一定能减少 NAT 块的下刷量.

综上, IBT B+ tree 的设计能够更好地利用空间局部性降低元数据的下刷量, 避免随机访问提升元数据的下刷效率. NAT 版本, B+ tree 树结点总量越多, 更新 NAT 块越多, 写放大越严重, 增加 HDD 寻道时间从而降低系统吞吐率. 因此, IBT 版本在元数据写放大和下刷效率方面的表现均优于 NAT 版本.

5.3 Filebench 测试

为避免空读, 测试查询、插入、更新、删除操作的综合性能, 使用 Filebench 进行测试, 该组测试主要选取 varmail, fileserver 这 2 个负载. 为充分测试对比 IBT B+ tree 和 NAT B+ tree 的综合性能, 同时适配当前服务器环境, 对 Filebench 中标准 wml 脚本的测试时间、线程数、文件大小、文件数量均作了修改. 这些负载均首先分配一个文件集合, 2 类负载的行为如下:

1) varmail, 模拟邮件服务器的 I/O 负载, 该负载多线程执行相同任务. 顺序执行删除、创建—追加—同步、读—追加—同步、读操作.

2) fileserver, 模拟文件服务器上的 I/O 负载, 该负载多线程执行相同任务. 顺序创建、写整个文件、追加写、读整个文件、删除文件、查看文件状态.

测试方法为: 分别基于 HDD 和 SSD 创建 4TB 的 Monty-Dev 设备, 格式化为 ext4 文件系统, 使用 discard 选项挂载文件系统以测试 B+ tree 的删除操作, 每类负载执行 1 h, 每个测试执行 3 次, 结果取平均值.

5.3.1 基于 SSD 的 Filebench 测试

Filebench 测试结果, IBT 版本和 NAT 版本基本相同, 差异较小. 主要原因有: 文件系统并不会将写请求直接转发至底层设备, 设备利用率低; 文件系统下发的写请求基本为顺序写, 2 个版本差异较小. 因此, 本节及 5.3.2 节主要通过元数据下刷量评价 IBT 版本和 NAT 版本. 由于 Filebench 测试按照指定时长进行测试, 不同测试中 Monty-Dev 设备收到的 I/O 请求数量存在差异, 因此, 本节及 5.3.2 节使用元数据下刷比例指标评价元数据的写放大情况:

$$\text{元数据下刷比例}=\frac{\text{元数据下刷量}}{\text{写数据总量}}\times 100\%. \quad (1)$$

通过式(1)的计算方法, 计算结果如图 13 所示.

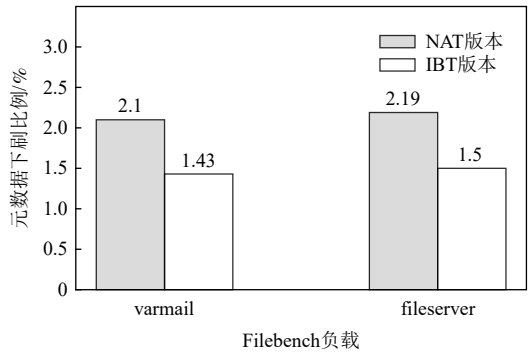


Fig. 13 Comparison of Filebench test based on SSD

图 13 基于 SSD 的 Filebench 测试对比

相对于 NAT 版本, IBT 版本在 varmail 负载和 fileserver 负载测试中, 元数据下刷比例分别减少 0.67% 和 0.69%. 经统计, 若元数据写比例仅统计树结点下刷量, 则 2 类负载测试中 IBT 版本均大于 NAT 版本, 该现象符合 IBT 版本和 NAT 版本设计的差异. 因此, 导致 NAT 版本元数据下刷比例高的主要原因是 NAT 块下刷的开销大. 接下来, 具体分析 NAT 版本元数据下刷量的组成.

如表 9 所示, NAT 块下刷量占元数据下刷比例较表 6 所示的更高. 较 Fio 负载, Filebench 负载支持 discard I/O 请求, 系列删除操作完成后, 一些树结点需要被释放, 释放树结点需要将 NAT 块中的记录更新为无效映射. 经统计, varmail 和 fileserver 负载测试中 discard I/O 占写 I/O 请求的比例分别为 81.5% 和 76%. 因此, 删除操作会严重影响 NAT 版本的 NAT 块下刷量, 从而影响元数据下刷比例, 导致 NAT 版本写放大更加严重.

Table 9 Statistics of Flushed NAT Block for Different Loads Under Filebench Test Based on SSD

表 9 基于 SSD 的 Filebench 测试不同负载 NAT 块下刷统计

负载	NAT 块下刷量/GB	比例/%
varmail	6.9	34.3
fileserver	8.2	36.0

5.3.2 基于 HDD 的 Filebench 测试

评价方法与 SSD 上测试相同，元数据下刷比例对比结果如图 14 所示.

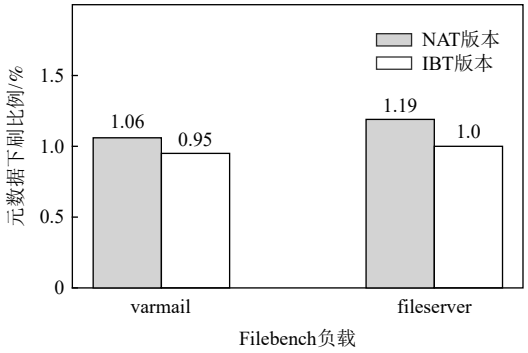


Fig. 14 Comparison of Filebench test based on HDD

图 14 基于 HDD 的 Filebench 测试对比

相对于 NAT 版本，IBT 版本在 varmail 负载和 fileserver 负载测试中，元数据下刷比例分别减少 0.11% 和 0.19%. HDD 上 IBT 版本降低比例较 SSD 上测试低的主要原因是：Filebench 测试预先写入的数据量相同，HDD 上测试 varmail 负载和 fileserver 负载分别比 SSD 上测试执行的操作数少约 92% 和 86%. 根据 5.3.1 节分析，删除操作是造成元数据下刷量差异的主要原因. 经统计，varmail 和 fileserver 负载测试中 discard I/O 占写 I/O 请求的比例分别为 26% 和 32.2%，均低于 SSD 上的测试结果. 造成 discard I/O 比例低的主要原因是：discard I/O 操作元数据处理开销较大，HDD 上处理较 SSD 慢，在测试结束后，ext4 文件系统仍然在发送 discard I/O. 因此，造成 HDD 上测试 IBT 版本元数据下刷比例降低的主要因素也是删除操作，造成降低比例较 SSD 测试低的原因是删除操作比例较低.

NAT 块下刷量相对于元数据下刷量的比例如表 10 所示. discard I/O 占比的差异与 NAT 块占比为正相关. 综上，在元数据写放大方面，相对于 NAT 版本，IBT 版本在删除操作中的表现也更优.

综上，进行了 Fio 测试和 Filebench 测试，IBT 版本相对于 NAT 版本：在 HDD 上的吞吐率和元数据下刷写放大方面，较 SSD 上的优化效果更显著；在处理大规模随机更新和删除操作时，开销更小.

Table 10 Statistics of Flushed NAT Block for Different Loads Under Filebench Test Based on HDD

表 10 基于 HDD 的 Filebench 测试不同负载 NAT 块下刷统计

负载	NAT 块下刷量/GB	占比/%
varmail	0.4	17.3
fileserver	0.63	20.7

6 总 结

本文主要研究了使用 B+ tree 管理日志结构块存储系统的地址映射面临的 wandering B+ tree 问题. B+ tree 具有扇出大、综合效率高的优点，被广泛应用于各类存储系统中. 对于日志结构块存储系统，使用 B+ tree 管理逻辑地址和物理地址的映射需要解决：支持树结点下刷方式为异地写，并避免递归更新 B+ tree 树结构. 本文从树结构和树操作设计的角度出发，提出了 IBT B+ tree，将树结点物理地址嵌入树结构的设计，引入 dirty 链表设计，定义 IBT B+ tree 插入、删除操作，并提出了非递归更新算法，既解决了 wandering B+ tree 问题，又不引入额外的数据结构和固定物理设备空间. 设计实现块设备存储系统 Monty-Dev，将 IBT B+ tree 与 NAT B+ tree 进行对比，实验结果表明在不同的负载下 IBT B+ tree 的写放大表现与 NAT B+ tree 相当或更优，下刷效率更高. IBT B+ tree 可应用于日志结构块设备系统，如 ASD 系统；也可应用于日志结构文件系统管理文件的地址映射，如 NILFS2 文件系统.

作者贡献声明：杨勇鹏负责论文撰写、部分实验方案设计、实验执行和论文修订；蒋德钧负责部分实验方案设计和论文修订.

参 考 文 献

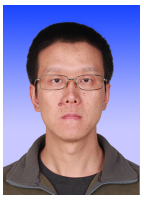
[1] Reinsel D, Gantz J, Rydning J. The digitization of the world – From edge to core [EB/OL]. 2018[2022-03-01].https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[2] Western Digital. Ultrastar DC HC550 [EB/OL]. 2020[2022-03-01].https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-hc550-hdd#0F38353

[3] Western Digital. Ultrastar DC HC650 [EB/OL]. 2020[2022-03-01].https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/data-center-

drives/ultrastar-dc-hc600-series/data-sheet-ultrastar-dc-hc650.pdf
[4] Nimbus Data. ExaDrive DC series [EB/OL]. 2020[2022-03-01].https://nimbusdata.com/docs/ExaDrive-DC-Datasheet.pdf
[5] Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system[J]. ACM Transactions on Computer Systems, 1992, 10(1): 26−52
[6] 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
[7] Chen Feng, Koufaty D A, Zhang Xiaodong. Understanding intrinsic characteristics and system implications of flash memory based solid state drives[J]. ACM SIGMETRICS Performance Evaluation Review, 2009, 37(1): 181−92
[8] Bouganim L, Jónsson B, Bonnet P. uFLIP: Understanding flash IO patterns [J]. arXiv preprint, arXiv: 09091780, 2009
[9] Min C, Kim K, Cho H, et al. SFS: Random write considered harmful in solid state drives [C/OL]//Proc of the 10th USENIX Conf on File and Storage Technologies (FAST '12). Berkeley, CA: USENIX Association, 2012: 139−154
[10] Woodhouse D. JFFS: The journalling flash file system [C/OL]//Proc of the Ottawa Linux Symp. 2001[2022-02-22].https://www.kernel.org/doc/mirror/ols2001/jffs2.pdf
[11] Lee C, Sim D, Hwang J, et al. F2FS: A new file system for flash storage [C]//Poc of the 13th USENIX Conf on File and Storage Technologies (FAST'15). Berkeley, CA: USENIX Association, 2015: 273−286
[12] Davenport C. The Pixel 3 uses Samsung's super-fast F2FS file system [EB/OL]2018[2022-03-25].https://www.androidpolice.com/2018/10/10/pixel-3-uses-samsungs-super-fast-f2fs-file-system/
[13] Bjørling M, Aghayev A, Holmberg H, et al. ZNS: Avoiding the block interface tax for flash-based SSDs [C]//Proc of 2021 USENIX Annual Technical Conf (USENIX ATC '21). Berkeley, CA: USENIX Association, 2021: 689−703
[14] Han K, Gwak H, Shin D, et al. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction [C]//Proc of the 15th USENIX Symp on Operating Systems Design and Implementation (OSDI '21). Berkeley, CA: USENIX Association, 2021: 147−162
[15] Na Wenwu, Meng Xiaoxuan, Si Chengxiang, et al. A novel network RAID architecture with out-of-band virtualization and redundant management [C] //Proc of the 14th Int Conf on Parallel and Distributed Systems (ICPADS 2008). Piscataway, NJ: IEEE, 2008: 105−112
[16] Ke Jian, Zhu Xudong, Na Wenwu, et al. Dynamic address mapping virtualization storage system[J]. Computer Engineering, 2009, 35(16): 17−19,22 (in Chinese)
(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)
[17] Na Wenwu, Meng Xiaoxuan, Ke Jian, et al. BW-VSDS: A network virtual storage system with large capacity, graceful scalability, high performance and high availability[J]. Journal of Computer Research and Development, 2009, 46(s2): 88−95 (in Chinese)
(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)
[18] Bityutskiy A B. JFFS3 design issues [J/OL]. Memory Technology Device (MTD) Subsystem for Linux, 2005[2022-01-06]. http://linux-mtd.infradead.org/tech/JFFS3design.pdf
[19] Linux Foundation. Linux kernel[EB/OL]. [2021-11-01].https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/
[20] Rodeh O. B-trees, shadowing, and clones[J]. ACM Transactions on Storage, 2008, 3(4): 1−27
[21] Rodeh O. B-trees, shadowing, and range-operations[R/OL]. IBM Research, 2006[2022-03-15].https://dominoweb.draco.res.ibm.com/reports/h-0248.pdf
[22] Bayer R, Schkolnick M. Concurrency of operations on B-trees[J]. Acta Informatica, 1977, 9(1): 1−21
[23] Kim J. f2fs-tools[EB/OL]. [2021-10-15].https://github.com/jaegeuk/f2fs-tools
[24] Yang Yongpeng. Research and implementation of memory metadata structure of SSD cache system[D]. Beijing: University of Chinese Academy of Sciences, 2018(in Chinese)
(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)
[25] Axboe J. Flexible I/O tester [EB/OL]. [2022-03-16].https://github.com/axboe/fio
[26] github.Filebench[EB/OL].[2022-03-16].https://github.com/filebench/filebench

Yang Yongpeng, born in 1993. PhD candidate. His main research interests include block storage system, file system.
杨勇鹏, 1993 年生. 博士研究生. 主要研究方向为块存储系统和文件系统.



Jiang Dejun, born in 1982. PhD, associate professor, PhD supervisor. Member of CCF, ACM, IEEE. His main research interests include storage architecture, storage system, distributed system.
蒋德钧, 1982 年生. 博士, 副研究员, 博士生导师. CCF, ACM, IEEE 会员. 主要研究方向为存储体系结构、存储系统、分布式系统.