

分类号_____

学号 D201880826

学校代码 10487

密级_____

华中科技大学 博士学位论文

(学术型 专业型)

日志结构文件系统性能优化研究

学位申请人：杨梨花

学科专业：计算机系统结构

指导教师：谭支鹏 教授

王芳 教授

答辩日期：2022年8月23日

答辩委员会

	姓名	职称	单位
主席	孟令奎	教授	武汉大学
委员	李战怀	教授	西北工业大学
	贾连兴	教授	国防科学技术大学
	胡燏翀	教授	华中科技大学
	凌贺飞	教授	华中科技大学

**A Dissertation Submitted in Partial Fulfilment of the Requirements
for the Degree of Doctor in Engineering**

**Research on Performance Optimization of Log-structured
File Systems**

Ph.D. Candidate : YANG Lihua

Major : Computer Architecture

Supervisor : Prof. TAN Zhipeng

Prof. WANG Fang

Huazhong University of Science and Technology

Wuhan 430074, P. R. China

August, 2022

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名: 杨梨花

日期: 2022 年 8 月 24 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

保 密口，在_____年解密后适用本授权书。
本论文属于
不保密。

(请在以上方框内打“√”)

学位论文作者签名: 杨梨花

日期: 2022 年 8 月 24 日

指导教师签名: 

日期: 2022 年 8 月 24 日

华中科技大学博士学位论文

摘要

基于 NAND 闪存 (NAND Flash) 的存储卡和固态硬盘因为其大容量、轻便、抗震等优异特性，在移动设备和数据中心被广泛使用。面向闪存特性设计的闪存友好型文件系统 (Flash Friendly File System, F2FS) 是一种典型的日志结构文件系统 (Log-structured File System, LFS)，它采用日志结构追加写机制和闪存芯片组织结构友好的数据布局，提升了随机写性能，获得了工业界和学术界的广泛关注和应用。文件系统因碎片问题导致性能下降，而 F2FS 所采用的异地更新机制同时加重了文件分配空间和空闲空间的碎片化程度，其高并发应用模式也会产生大量随机同步小写请求，进一步加剧碎片化，导致 I/O 请求响应变慢、设备运行卡顿、用户体验不佳。为解决上述碎片化导致的性能下降问题，本文从文件空间分配方法、多文件数据组织布局方法和无效空间回收机制三个方面开展了日志结构文件系统性能优化研究。

随着文件创建、更新和删除，文件碎片会快速积累，日志结构文件系统的异地更新机制加剧碎片化，移动应用产生的大量随机同步小写请求会加速文件系统老化。针对该问题，提出一种基于追溯文件的自适应预留空间策略 (Adaptive Reserved Space based on Traceback, ARST)。ARST 分析典型应用的行为特征，选择与碎片产生强相关的文件特征用于构造数据集，采用低成本、高准确率的机器学习算法在大量文件中精准选择会严重碎片化的文件，为这些文件预留空间，有效减少文件碎片的产生。为了最大化空间预留的效果，ARST 还选择与时间无关的碎片写特征，挖掘历史信息少的潜在碎片文件作为预留文件。为了更高效地使用存储空间，ARST 根据文件写特征决定合适的预留空间大小，并根据文件系统空间使用率决定是否保留文件预留空间。实验结果表明，相比于 F2FS 现有就地更新策略，ARST 显著减少了文件碎片，文件顺序读带宽提升高达 56.55%，应用执行时间减少了 33.78%~94.28%。

F2FS 将所有用户产生的常规文件数据组织为温数据类，共用一个日志 (log) 完成写操作，造成写特征差异明显的用户数据混合存储，其数据失效速率的差异化导致空闲空间碎片化程度增加。针对该问题，提出一种基于数据热度的多日志延迟写策略 (Multi-log delayed writing based on Hotness, M2H)。不同于传统基于语义信息和数

华中科技大学博士学位论文

据分析的静态分类, M2H 使用 K 均值聚类算法, 基于文件块更新距离感知数据热度变化, 动态地准确区分数据热度, 将数据分别写入相应热度的日志, 使热度相近的数据汇聚在同一块区域, 改善多文件间的数据组织布局, 并通过延迟写策略提升多日志区写操作效率。为了准确描述数据热度, M2H 基于负载读写特征选择文件块更新距离、最近使用距离和读次数定义热度, 文件块更新距离较小、最近使用距离较小和读次数较多的数据热度高。另外, M2H 基于段清理次数为基准的敏感性分析决定 K 值, 提升 K 均值聚类效果。实验结果表明, 与传统 F2FS 相比, M2H 有效减少了空闲空间碎片, 段清理次数减少了 50.61%~94.36%, 有效块迁移数量减少了 64.31%~99.98%。

当文件系统连续空闲空间耗尽时, 系统中存在大量的空闲空间碎片, 需要触发垃圾回收 (Garbage Collection, GC) 获得空闲段, 或使用线程日志写入方式, 覆盖写空闲空间碎片, 直接造成新写入文件的碎片化, 这两种方式都会导致文件系统读写性能下降。通过构造初始、重载和末端碎片化三种不同空闲空间碎片数量的状态, 分析了空闲空间碎片对能耗的影响、垃圾回收的能耗及无效空间回收效率。发现空闲空间碎片增加了系统任务 I/O 次数, 导致能耗明显增加, 使用 GC 整理空闲空间碎片时, 后台 GC 能耗较大, 且减少空闲空间碎片的效果有限。针对该问题, 提出空闲空间碎片感知的 GC 策略 (free space Fragmentation Aware GC, FAGC)。FAGC 基于数据分析重新评估了空闲空间碎片大小的阈值, 使用空闲空间碎片系数快速衡量空闲空间碎片化程度, 选择空闲空间碎片化严重的段作为清理对象, 增大每次 GC 的收益, 提高 GC 整理空闲空间碎片的效率, 有效降低能耗。实验结果表明, 与 F2FS 最新的 ATGC 策略相比, 在空间使用率较高和空闲空间碎片数量较多时, FAGC 减少 I/O 密集性负载的 GC 次数高达 74.51%, 减少有效块迁移数量高达 73.92%, 降低能耗高达 100.64 焦, 具有更高的无效空间回收能效。

关键词: 日志结构文件系统; 文件碎片; 空闲空间碎片; 垃圾回收; 能耗

Abstract

NAND Flash-based memory cards and solid state drives are widely used in mobile devices and data centers because of their large capacity, light weight, and shock resistance. The Flash Friendly File System (F2FS) designed for flash memory is a typical Log-structured File System (LFS), which adopts a log-structured write scheme and a friendly organization structure of flash memory chips that improve the random write performance, so F2FS has been widely concerned and applied in industry and academia. The performance of file system degrades due to the fragmentation problem and the out-of-place update scheme adopted by F2FS increases the fragmentation of the file and free space at the same time. The high concurrent application mode will generate a large number of random synchronous small requests, which further aggravates the fragmentation. Fragmentation results in slow responses to I/O requests, slow device operation, and poor user experience. In order to solve these problems of performance degradation caused by fragmentation, this paper conducts performance optimization research on log-structured file systems from three aspects: file space allocation method, multi-file data organization and layout method, and invalidation space recovery mechanism.

File fragments accumulate rapidly with creating, updating, and deleting files, and the out-of-place update mechanism of log-structured file systems exacerbates fragmentation. A large number of random synchronous small requests generated by mobile applications accelerate the aging of the file system. To address this problem, an adaptive reserved space based on traceback (ARST) scheme that uses machine learning algorithms is proposed. ARST analyzes the behavioral characteristics of typical applications, selects file characteristics that are strongly related to fragmentation to construct datasets, and uses low-cost, high-accuracy machine learning algorithms to accurately select files that will be severely fragmented among a large number of files, and provides these files with reserved space to reduce file fragmentation effectively. In order to maximize the effect of space reservation, ARST also selects time-independent write features and mines potential fragmented files with little historical information as reserved files. In order to use the storage space more efficiently, ARST determines the appropriate reserved space according to the

file write characteristics and decides whether to reserve the file reserved space according to the file system space usage. The experimental results show that compared with the existing in-place update strategy of F2FS, ARST reduces file fragmentation significantly, improves the sequential read bandwidth by up to 56.55%, and reduces the running time of applications by 33.78%~94.28%.

F2FS classifies all regular file data generated by users into warm data and uses a log to complete the write operations, resulting in mixed storage of user data with obvious differences in write characteristics and the difference in data failure rate leads to an increase in the degree of free space fragmentation. To address this problem, a multi-log delayed writing based on hotness (M2H) strategy is proposed. Different from the traditional static classification based on semantic information and data analysis, M2H uses the K-means clustering algorithm to perceive the change of data hotness based on the file block update distance, distinguishes the data hotness dynamically and accurately, and writes the data into the corresponding hotness logs respectively, so that the data with similar hotness are gathered in the same area which improves the data organization and layout among multiple files, and improves the write efficiency of multi-log area through delayed writing strategy. In order to describe the data hotness accurately, M2H selects the file block update distance, the most-recently-used distance, and the number of reads to define the hotness based on the workload reading and writing characteristics. Sensitivity analysis based on the number of segment cleanings to determine the K value makes the K-means clustering effect better. The experimental results show that, compared with traditional F2FS, M2H reduces free space fragmentation effectively, the number of segment cleanings is reduced by 50.61%~94.36%, and the number of migrated valid blocks is reduced by 64.31%~99.98%.

When the continuous free space of the file system is exhausted, there are a large number of free space fragments in the system. Garbage collection (GC) needs to be triggered to obtain free segments, or the threaded logging write scheme is used that overwriting free space fragments directly makes new-written files fragmented. Both methods will lead to a decrease in the read and write performance of the file system. By constructing three states of the different numbers of free space fragments: initial, full, and full fragmented, the impact of free space fragments on energy consumption, the energy consumption of garbage collection and its effect on recycling invalid blocks are analyzed. It is found that free space

华中科技大学博士学位论文

fragmentation increases the number of system tasks I/Os, resulting in a significant increase in energy consumption. When using GC to reduce free space fragments, the background GC consumes a lot of energy, and its effect on reducing free space fragments is limited. To solve this problem, a free space Fragmentation Aware GC (FAGC) strategy is proposed. FAGC re-evaluates and defines the threshold of free space fragmentation based on data analysis, uses the free space fragmentation factor to measure the degree of free space fragmentation quickly, selects the cleaning objects with serious free space fragmentation, increases the revenue of each GC, and improves the efficiency of GC to reduce free space fragmentation and reduces energy consumption effectively. The experimental results show that, compared with the latest ATGC strategy of F2FS, when the space utilization is high and the number of free space fragments is large, FAGC reduces the GC count of I/O-intensive workloads by as much as 74.51%, reduces the number of migrated valid blocks by as much as 73.92%, and reduces energy consumption by up to 100.64 J, so FAGC has a higher energy efficiency of recycling invalid space.

Key words: Log-structured File System, File Fragmentation, Free Space Fragmentation, Garbage Collection, Energy Consumption

华中科技大学博士学位论文

目 录

摘 要.....	I
Abstract.....	III
1 绪论	
1.1 研究背景.....	(1)
1.2 性能优化研究现状.....	(10)
1.3 研究内容与论文组织结构.....	(21)
2 自适应选择文件的空间预留策略	
2.1 减少文件碎片的需求和挑战.....	(25)
2.2 预留文件选择方法.....	(32)
2.3 空间预留策略.....	(42)
2.4 性能评估与分析.....	(44)
2.5 本章小结.....	(53)
3 基于数据热度的多日志延迟写策略	
3.1 温数据冷热分离的需求和挑战.....	(55)
3.2 基于 K 均值聚类的数据热度识别.....	(58)
3.3 多日志延迟写及段清理策略.....	(67)
3.4 性能评估与分析.....	(69)
3.5 本章小结.....	(79)
4 空闲空间碎片感知的垃圾回收策略	
4.1 碎片问题对能耗的影响.....	(80)
4.2 垃圾回收能耗分析及挑战.....	(85)
4.3 碎片感知垃圾回收策略.....	(90)
4.4 性能评估与分析.....	(96)

华 中 科 技 大 学 博 士 学 位 论 文

4.5 本章小结 (101)

5 总结与展望

5.1 本文的主要创新点 (102)

5.2 未来工作展望 (104)

致 谢 (106)

参考文献 (108)

附录 1 攻读博士学位期间取得的研究成果 (123)

附录 2 公开发表的学术成果与博士学位论文的关系 (125)

附录 3 攻读学位期间参加的科研项目 (126)

附录 4 中英文缩写对照表 (127)

1 绪论

1.1 研究背景

受益于 NAND 闪存 (NAND Flash) 轻便、容量大等优异特性，闪存存储器已被广泛应用于智能手机、平板、个人电脑和服务器等。随着闪存存储器逐步成为主要的存储器件之一，多流固态硬盘技术^{[3][4]}和分区命名空间固态硬盘（Zoned Namespace Solid State Drive, ZNS SSD）技术^{[5][6]}都得到了快速发展，针对闪存特性设计的闪存友好型文件系统（Flash Friendly File System, F2FS^[1]）逐渐流行。日志结构技术将小的随机写请求组合成一个较大的顺序写入请求，机械硬盘和固态硬盘可以有效地处理写请求，传统的日志结构文件系统性能依赖于连续的空闲空间，维护这些空间需要开销较大的垃圾回收操作。F2FS 是一种典型的日志结构文件系统（Log-structured File System, LFS）^[2]，采用日志结构方式写入和闪存物理单元友好的数据布局方法，提高了随机小写性能，通过前滚恢复和检查点操作实现低成本且高效的一致性保护，因此在移动设备场景应用广泛。基于 NAND 闪存的移动设备在近十年高速发展，常见的移动设备包括智能手机、平板、可穿戴设备等，其中智能手机的需求量最大、使用最为普遍，目前全球智能手机用户数量已超过 60 亿¹，F2FS 在移动设备领域使用广泛，其性能优化具有重要的经济收益，受到学术界和工业界的广泛关注。

闪存存储系统分层结构如图 1.1 所示，在这个分层结构的顶部，用户空间进程通过系统调用访问文件，如打开、读取、写入等等。SQLite 数据库是安卓（Android）系统默认的关系型数据库，大量应用如系统邮件和脸书，使用 SQLite 持久地管理文件数据，SQLite 的插入、删除、更新操作速度影响系统响应时间，C 标准函数库（C standard library, libc）是最基本的 C 语言函数库，提供 C 语言中使用的宏、类型的定义、字符串操作符、数学计算函数以及输入输出函数等。作为具体文件系统之上的抽象层，虚拟文件系统（Virtual File System, VFS）将文件操作命令发送到实际文件

¹ <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>

系统,如 F2FS,常用闪存文件系统还有日志闪存文件系统(Journaling Flash File System Version 2, JFFS2)^[7]和无序区块映像文件系统(Unsorted Block Image File System, UBIFS)^[8]。UBIFS 由诺基亚公司和匈牙利塞格德大学共同研发,使用 UBI 层管理闪存坏块,通过内存技术设备(Memory Technology Device, MTD)访问闪存芯片,解决了 JFFS 存在的内存消耗大、可扩展性差、损耗均衡能力差等问题^[9]。不同于 UBIFS, F2FS 不直接面向裸 NAND 闪存,而是和其他通用文件系统一样面向块设备层接口,通过闪存转换层(Flash Translation Layer, FTL)访问 NAND 闪存。

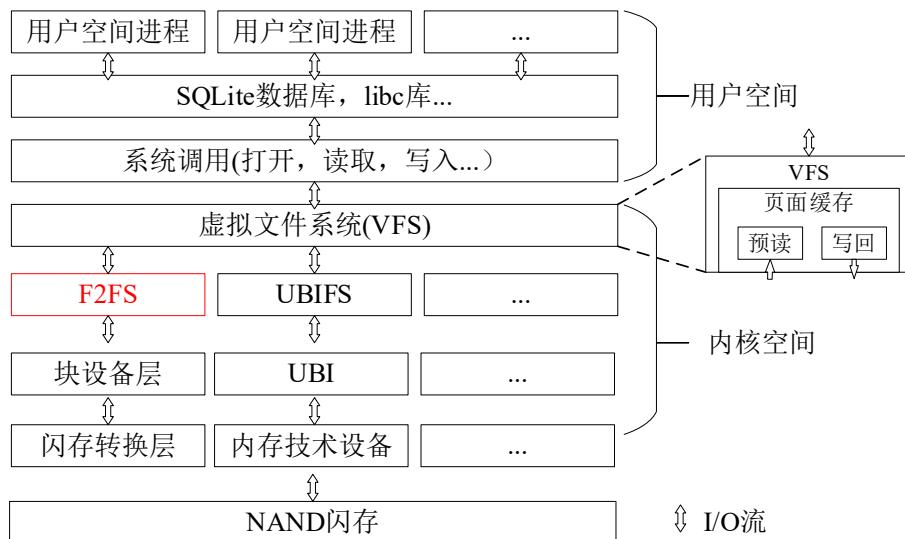


图 1.1 闪存存储系统的结构

基于闪存特性设计的数据布局和管理机制, F2FS^[1]已广泛使用。表 1.1 列举了使用 F2FS 文件系统的商用移动设备,从 2020 年 9 月发布的移动终端操作系统 Android 11 开始,安卓开源项目(Android Open Source Project, AOSP¹)的动态系统更新要求 /data 分区使用 F2FS 或第四代扩展文件系统(fourth extended file system, ext4^[10]),由于 F2FS 提供更出色的读写性能,建议使用 F2FS²。F2FS 使用日志结构写方式提高了随机写性能,使用检查点和前滚恢复机制保证数据一致性,在移动应用要求同步数据时能够快速响应,基于闪存物理单元友好的数据组织布局, F2FS 在数据中心 SSD

¹ <https://source.android.com/>

² <https://source.android.google.cn/devices/tech/ota/dynamic-system-updates?hl=zh-cn>

华中科技大学博士学位论文

服务器也广泛使用^[11]。

表 1.1 用 F2FS 文件系统的商用移动设备

生产年份	商用移动设备
2012-2015 年	摩托罗拉 MSM8960 JBBL 设备/ Droid/ G/ X 系列
2014-2016 年	谷歌 Nexus 9、摩托罗拉 E LTE/Z、一加 3T、华为 Honor 8/ V8/ P9/ Mate 9
2018-2019 年	谷歌 Pixel 3/3 XL、中兴 Axon 10 Pro、三星 Galaxy Note 10/Tab S6
2020 年起	使用 Android 11 的谷歌 Pixel 系列

F2FS 在提供良好 I/O 性能的同时也被碎片化^{[12][13]}问题困扰，随着频繁地创建、更新和删除文件，文件系统产生严重的碎片化问题，通常分为文件碎片和空闲空间碎片，文件碎片是单个文件离散分布的数据片段，空闲空间碎片由未回收的无效空间和离散分布的空闲空间构成。文件碎片分割 I/O 请求，使 I/O 请求变小，增加访问随机性，导致文件系统性能下降。重整文件碎片可以减少文件碎片，但需要复制文件数据，并将其写到新地址，会增加文件系统读写开销，缩短闪存寿命，为减轻上述隐患，只在必要的时候才重整文件碎片。不同热度的数据混合存储增加了空闲空间碎片数量，而空闲空间碎片的增多加剧文件系统段清理¹开销^{[14][15]}，数据热度描述了未来一段时间内数据被访问的可能性。目前 F2FS 使用垃圾回收（Garbage Collection, GC）整理空闲空间碎片，然而 GC 增加了读写开销和能量消耗，在保证文件系统性能良好的前提下，如何优化 GC 策略，提高 GC 减少空闲空间碎片的效率非常重要。综上所述，文件碎片造成文件系统性能下降、不同热度的数据混合存储增加空闲空间碎片、后台 GC 减少空闲空间碎片效果有限是日志结构文件系统性能优化的三个挑战。

1.1.1 日志结构文件系统技术

20 世纪 90 年代初 Mendel Rosenblum 和 John K. Ousterhout 提出日志结构文件系统（Log-structured File System, LFS）^[2]的设计与实现，随着 NAND 闪存存储器成本

¹ 如不特指闪存转换层的垃圾回收，本文中的垃圾回收与段清理都是在文件系统层面回收无效块，Lee 等人在论文^[1]中表示为段清理（segment cleaning），在内核的 F2FS 源码中表示为 GC。

华中科技大学博士学位论文

降低、广泛使用，日志结构文件系统受到工业界和学术界广泛关注。日志结构文件系统以日志(log)方式追加写，使用段清理回收无效空间，NAND 闪存采用异地更新，以闪存页为单位进行写入，以闪存块为单位进行擦除，F2FS 专为闪存特性设计，采用日志结构写方式，以文件系统层 4 KB 大小的逻辑块为单位进行读写，以逻辑节为单位进行擦除，F2FS 的 I/O 性能比 ext4 更好^[1]，读写能耗更低^[16]。

日志结构文件系统技术在不断改善，Seltzer 等人分析了 Unix 快速文件系统和 LFS 的性能^[17]，发现当元数据操作成为瓶颈时，相比于 Unix 快速文件系统，LFS 性能更好，当创建 1 KB 及更小的文件或删除 64 KB 及更小的文件时，LFS 提供了一个数量级的性能改进。郑特龙提出一个基于双模相变存储器（Phase Change Memory，PCM）的日志结构文件系统，在保证数据一致性的同时提高基于 PCM 的文件系统吞吐量^[18]。非易失存储器加速的日志结构文件系统（NOVolatile memory Accelerated，NOVA）^[19]是一个混合易失和非易失的内存文件系统，采用传统的日志结构文件系统技术充分发挥非易失存储器（Non-Volatile Memory，NVM）的快速随机访问特性，为每个文件提供单独的日志，实现高性能同时保证一致性。Xu 等人进一步通过合并快照、校验码等技术提高 NOVA 的容错性和可靠性，提出 NOVA-Fortis^[20]兼顾容错性和高性能。Liao 等人提出一个多核友好的日志结构文件系统 MAX^[21]，使用一种新的读写锁实现并发控制，引入文件单元扩展对内存索引和缓存的访问，采用多个日志分区实现并发空间分配，解决文件系统内部的扩展性瓶颈并提升中央处理器（Central Processing Unit，CPU）并行性。

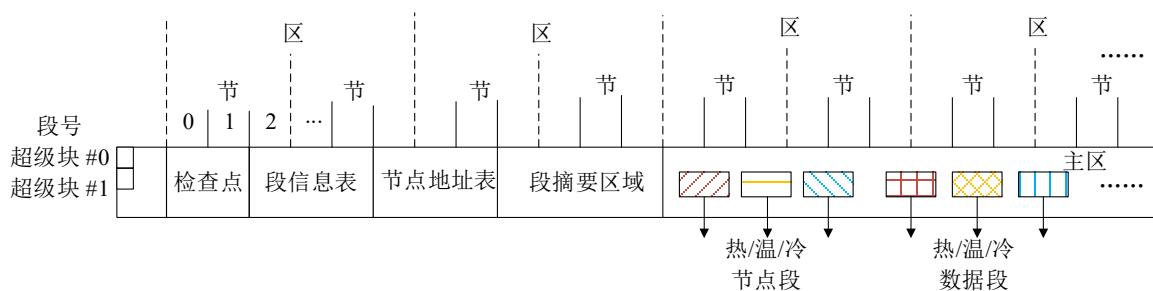


图 1.2 F2FS 的文件布局

F2FS 是典型的日志结构文件系统，其文件数据组织布局如图 1.2 所示，划分为区(zone)、节(section)、段(segment) 和逻辑块(logical block)，以方便用户在格

式化文件系统时将这些单元与闪存芯片的晶圆 (die)、分组 (plane)、闪存块 (flash block) 和闪存页 (flash page) 等物理单元对齐。逻辑块是最基本的读写单位，大小为 4 KB，512 个块组成一个段，大小为 2 MB，一个或多个段构成一个节，节是 F2FS 垃圾回收的单位，实际应用的 F2FS 设置一个节由一个段组成，一个或者多个节组成一个区，如此进行数据组织布局是为了尽量与闪存操作单元同步。F2FS 在逻辑上包含六个部分，分别是超级块 (Super Block, SB)、检查点 (Check Point Pack, CP)、段信息表 (Segment Information Table, SIT)、段摘要区 (Segment Summary Area, SSA)、节点地址表 (Node Address Table, NAT) 以及主区 (Main Area)。两个超级块存放文件系统静态元数据，互为备份，检查点记录文件系统动态元数据，用于保证文件系统一致性，段信息表记录文件系统段中逻辑块使用状态，段摘要区记录段中逻辑块所属状态，这两部分信息用于段清理，节点地址表用于缓解日志结构写方式的数据更新引入的元数据写放大问题，主区由 4 KB 大小的逻辑块组成，用于存储数据（文件数据或目录）和节点 (inode 或数据块索引)，主区的数据占整个系统的绝大多数。除主区采用日志结构写入的异地更新方式以外，其他与文件系统元数据相关的区域采用就地更新方式。

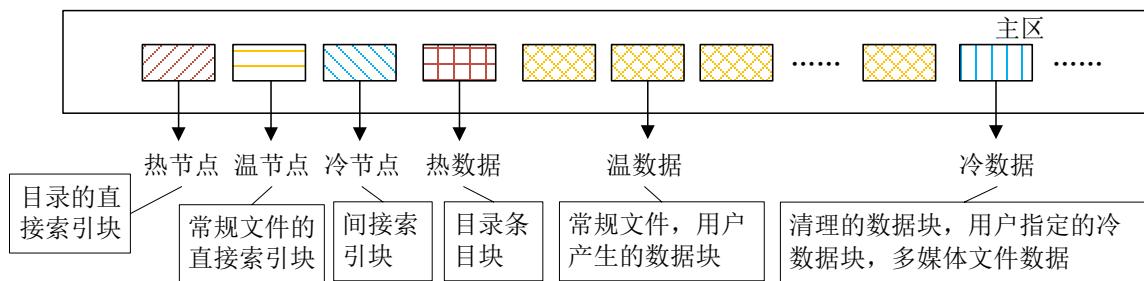


图 1.3 F2FS 主区数据的热度分类

分离冷热数据能实现有效数据的双峰分布^[14]，更新频率较低的冷数据不容易失效，更新频率较高的热数据更容易失效，更新后旧版本热数据失效，所在段迁移成本低，有较大的可能被选为清理对象，能减少 GC 迁移冷数据造成的写放大。主区文件块的热度根据文件系统语义，如数据功能（索引数据和文件数据），划分为六大类，分别是热、温、冷的节点和数据块，主区数据热度分类如图 1.3 所示，这六类文件块在文件系统中分别从 6 个 log 写入，根据系统语义 F2FS 将用户写常规文件产生的大

量数据块分类为温数据，可以看出温数据的数据量占比最大。

自 2012 年 12 月 20 日在 Linux 内核 3.8 版本中集成 F2FS 之后，F2FS 的性能优化研究受到广泛关注。文件碎片增加 I/O 请求数量，造成文件系统老化，导致设备响应用户请求变慢，针对该问题需要在不影响设备寿命的前提下减少文件碎片。日志结构文件系统性能取决于连续的空闲空间，连续空闲空间不足时需要执行段清理来获得空闲段，因此在保证文件系统性能不下降的前提下减少段清理次数和开销成为关键。当空闲空间碎片数量比较多时，为回收无效空间 F2FS 会频繁触发段清理，需要有效减少空闲空间碎片以减少段清理开销。研究 F2FS 的能耗后发现垃圾回收的能耗比较大，而后台垃圾回收减少空闲空间碎片的效果有限，触发后台 GC 回收无效空间时提高单位能耗的收益是关键。综上所述，针对日志结构文件系统的文件碎片和空闲空间碎片，优化日志结构文件系统性能是重要且富有挑战的研究工作。

1.1.2 文件系统性能优化研究

文件系统性能优化研究受到广泛关注，性能在本文中包括带宽、延迟和能耗，移动设备存储资源有限，与用户的高度互动特性要求低时延低能耗。大量工作研究智能手机上存储系统的瓶颈^{[22][23]}，研究移动存储系统 I/O 特点^[24]与应用行为特征^{[25][26]}，深入理解移动存储系统的能耗特征^[27]，保证移动存储系统的快速响应^{[28][29]}以及解决能耗问题^{[30][31]}。Jingpei Yang 等人提出 log-on-log 问题^[32]，虽然移动应用、文件系统和 NAND 闪存的数据都是异地更新的，但因为工作负载的随机性、未对齐的段大小和不协调的多日志垃圾回收，反而使读写效果更差。汉阳大学的 Jeong 等人提出 Androstep^[33]分析安卓移动设备存储系统的性能，Androstep 包括工作负载生成器 MobiBench 和工作负载分析器 MOST (Mobile Storage analyzer)。Hahn 等人发现页面缓存和闪存存储设备存在 I/O 优先级倒置现象，但现有减少 I/O 优先级倒置的技术不适用于智能手机，提出一种前台应用感知的 I/O 管理方案^[34]，通过抢占整个 I/O 栈中的后台请求以及防止前台应用的数据被从页面缓存中下刷来加速前台请求。Han 等人观察到移动设备目录项查找时一部分前缀是一样的，通过动态跳过常见路径前缀优化性能^[35]。Liang 等人发现安卓手机的内存管理问题^{[36][37]}，如回收空间太大、回收范围有限等，调整回收页面的内核线程 kswapd，在回收大小和整体性能间均衡。Mao

华中科技大学博士学位论文

等人基于应用间的重复数据比较少的特点提出一种应用感知的数据去重技术^{[38][39]}, Ji 等人基于移动应用的文件访问模式提出双模式压缩技术以减少总写入流量^[40], 任晶磊在页缓存中采用原子性事务机制, 有效降低移动应用的响应时间和能耗^[41]。

日志结构文件系统性能优化旨在提高吞吐量, 缩短响应时延, 降低能耗。移动消费电子设备由电池供电, 但电池的尺寸和容量有限, 管理好能耗至关重要, 能耗同样是大规模存储系统性能优化约束之一^[42], 现有的能耗研究工作主要从分析能耗和减少能耗两个角度展开研究。

一些研究工作分析移动设备各组件存储活动的能耗, 搭建模拟器评估系统的能耗。Mohan 等人基于差分分析的实验结果表明由随机 I/O 主导的工作负载中, 智能手机存储子系统会消耗大量能量 (36%), 与屏幕能耗相当, 比网络能耗多^[16], 还发现文件系统中随机 I/O 比顺序 I/O 消耗更多的能量, 对于大多数工作负载, F2FS 只消耗 ext4 一半的能量。Carroll 和 Heiser 分析了手机 Openmoko Neo Freerunner 的电量消耗, 开发了 Freerunner 的功率模型, 发现闪存芯片本身的能耗较低, 但执行闪存管理层的组件, 如 CPU 和随机存取存储器 (Random Access Memory, RAM) 的能耗是不可忽略的^[43]。Li 等人分析了移动设备存储硬件和软件所消耗的能量^[44], 建立了一个能量模型 EMOS (Energy Modeling for Storage) 估计存储活动所需能量, 一组存储密集型微基准测试显示, 在安卓手机和 Windows RT 平板电脑上, 存储软件比存储硬件多消耗 200 倍的能量。Yoon 等人提出了安卓能量衡量系统 AppScope^[45], 通过监视硬件组件请求的内核活动, 提供关于应用能耗准确而详细的信息。Olivier 等人提出了一种三阶段方法来估计闪存存储系统应用 I/O 的性能和能耗^[47], 在探索阶段识别了影响存储系统性能和能耗的主要因素, 在建模阶段用方程和算法对主要影响因素构建模型, 最后一个阶段提出名为 OpenFlash 的模拟器并实现了原型。

一些研究工作基于应用行为特征和 I/O 模式参数降低能耗, 由于后台应用不断消耗智能手机的电池电量, 如何平衡应用启动延迟和电池寿命成为问题, Chung 等人提出了一种应用管理框架 (application management framework) 终止不用的后台应用以节省能量, 并预启动有益的应用以减少应用启动延迟^[46]。Nguyen 等人研究缓存、设备驱动和块层中的存储参数如何影响智能机能耗, 设计并实现了 SmartStorage 系

统^[48]跟踪智能手机运行时的 I/O 模式，将当前 I/O 模式与从 8 个基准中记录的已知模式匹配最优参数，如调度算法和队列深度，然后动态配置存储参数以降低能耗。Huang 等人利用可穿戴设备上蓝牙和 Wi-Fi 混合的低能耗网络连接，设计基于电池支持 RAM 的可穿戴设备快速存储系统 WearDrive^[49]，将数据和计算从可穿戴设备传递到手机上，以低能耗成本在手机上执行大型高能耗任务，同时使用电池支持的 RAM 执行小型高能效任务以减少可穿戴设备的能耗。

一些研究工作使用新型高能效存储器件降低能耗，Zhong 等人提出 DR.Swap^[50]，采用节能非易失存储器作为交换区，利用 NVM 的字节寻址能力，允许读请求从交换区直接读取，保证只读页的零拷贝，降低智能手机能耗。Yan 和 Fu 观察到智能手机应用中超过 40% 的二级（Level 2, L2）缓存访问是操作系统内核访问，频繁的内核访问导致 L2 缓存中用户块与内核块严重干扰，提出将 L2 缓存划分为两个分别只能由用户代码和内核代码访问的单独小段，同时发现用户段与内核段的访问行为完全不同，提出在用户段使用短保留（short-retention）自旋转转移扭矩随机存取器（Spin-Transfer Torque Random Access Memory, STT-RAM），在内核段使用长保留 STT-RAM 的多保留 STT-RAM 使用方案^[51]，以节省缓存能耗。

综上所述，现有工作研究存储系统能耗的模型和特点，针对能耗问题从基于读写特征和利用新型器件两个角度降低能耗展开了研究。需要进一步了解日志结构文件系统的存储软件活动能耗特征，理解文件系统碎片化和段清理对能耗的影响，提出减少能耗的有效策略。

1.1.3 文件系统碎片化

文件碎片和空闲空间碎片造成文件系统老化，是文件系统性能提升的瓶颈。日志结构文件系统的异地更新机制更是加重了文件碎片化和空闲空间碎片化程度，加剧了日志结构文件系统的段清理开销。如图 1.4 所示，在一个初始化的文件系统上，文件 A、B 的数据连续存放，随后创建文件 D，更新文件 A，追加写文件 D，更新文件 B，删除文件 C，文件系统中的数据分布变得离散，碎片化的文件系统状态存在文件 A、B、D 的文件碎片，还存在空闲空间碎片。文件碎片是单个文件离散分布的数据片段，文件碎片增加读取文件的 I/O 次数，导致访问随机性增大，直接造成读性能下

华中科技大学博士学位论文

降。空闲空间碎片由未回收的无效空间和离散分布的空闲空间构成，由于 F2FS 日志结构追加写，其空闲空间碎片主要由无效数据块组成，离散分布的空闲空间存在于 F2FS 所写 6 个日志头的位置。

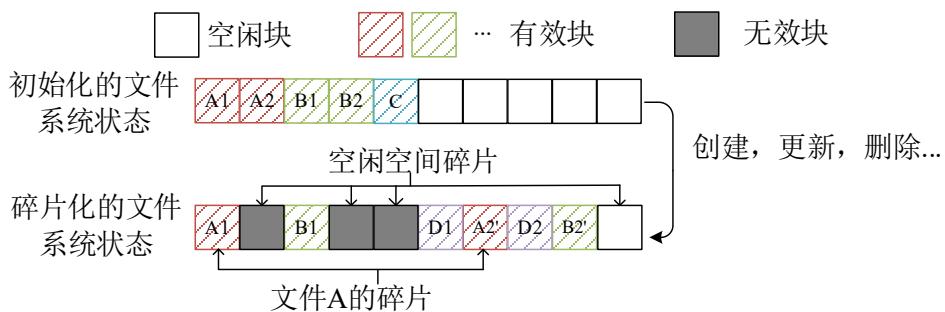


图 1.4 日志结构文件系统中文件碎片和空闲空间碎片的产生示意图

当空闲空间不足时触发段清理，整理空闲空间碎片获得整个空闲段，此时存在大量空闲空间碎片，F2FS 也支持以线程日志（threaded logging）方式写“洞¹”（hole），不同于追加日志写方式，线程日志写方式不会触发段清理，但可能产生随机写，导致写性能下降，进而造成新的文件碎片，降低顺序读性能。日志结构文件系统一般通过日志以追加写形式在新地址写新数据，这样异地更新数据导致文件碎片增加，旧版本数据失效造成空闲空间碎片增多。文件碎片和空闲空间碎片二者之间互相影响，文件更新后，存放该文件数据的文件碎片可能变成存放无效数据的空闲空间碎片，线程日志写空闲空间碎片可能导致新的文件碎片。

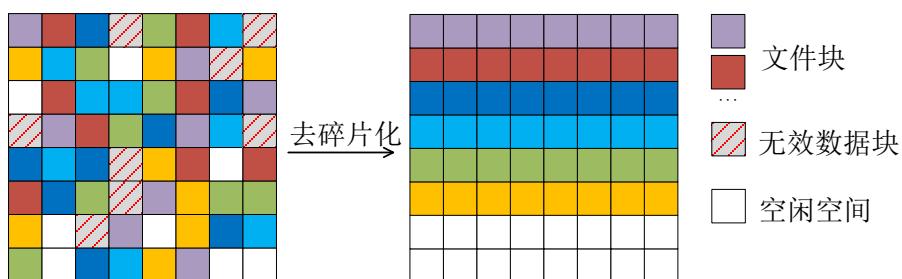


图 1.5 文件系统去碎片化示意图

减少文件碎片和空闲空间碎片对文件系统性能提升非常重要，如图 1.5 所示，去

¹ “洞”（hole）指离散分布的空闲空间碎片。

碎片化 (defragmentation) 包括减少不同文件的文件碎片和空闲空间碎片数量。图 1.5 中去碎片化过程复制碎片文件数据重整到新的连续地址，以有效减少文件碎片，但需要复制数据，F2FS 依赖段清理整理空闲空间碎片并回收无效空间。

1.2 性能优化研究现状

1.2.1 碎片的产生及影响研究

文件系统碎片化包括文件系统中的文件碎片和空闲空间碎片，基于不同老化工具的碎片产生与影响的研究工作如表 1.2 所示，使用不同方法制作老化工具，通过复现各种文件系统的碎片，验证文件系统因碎片而老化，研究碎片的产生原因，量化碎片对性能的影响，发现碎片快速产生、显著降低文件系统性能。老化工具分为三类，合成负载生成器、脚本执行实际应用和 trace 重播，Kadekodi 等人提出的 Geriatrix^[13] 属于合成负载生成器，Smith 和 Seltzer 的老化工具^[53]以及 Liang 等人使用的 Iozone 也是合成负载生成器，Ji 等人提出的 AutoAge^[56] 和 Conway 等人提出的 Git-Benchmark^[12] 属于脚本执行实际应用的老化工具，Jeong 等人使用 MobiBench¹ 抓取应用 trace 并重播^[24] 属于 trace 重播一类。

表 1.2 文件系统碎片产生与影响的研究现状

老化工具类型	工具名称	主要特点
合成负载生成器	Geriatix	提出文件系统老化工具 Geriatix ^[13]
	老化负载	利用文件系统快照产生碎片 ^[53]
	Iozone	研究 F2FS 中文件碎片对性能的影响 ^[54]
脚本执行实际应用	AutoAge	产生、衡量和减少文件碎片的方法 ^{[55][56]}
	Git-Benchmark	证实大多数文件系统都会老化 ^[12]
trace 重播	MobiBench	使用 MobiBench 抓取应用 trace 并重播 ^[24]

Kadekodi 等人提出一个配置文件驱动的文件系统老化工具 Geriatix^[13]，用于产生目标级别的文件碎片和空闲空间碎片，老化过程分为快速老化和平稳老化两个阶

¹ <https://github.com/ESOS-Lab/Mobibench>

华中科技大学博士学位论文

段。Geriatrrix 使用系统分区大小、空间使用率、文件大小分布、目录深度分布、相对年龄分布等老化配置参数，提供一个包含 8 个文件系统老化配置文件的存储库用于老化文件系统。

Smith 和 Seltzer 利用文件系统快照收集信息，如 inode 号、文件类型、文件大小等，构造老化负载，通过重放与实际文件系统几个月甚至几年时间里所经历的工作负载相似的老化负载，测试文件系统产生碎片后的性能，并评估了 UNIX 快速文件系统（Unix Fast File System, FFS）用该方法生成的负载老化后的性能^[53]。

香港城市大学的 Liang 等人评估移动设备上 F2FS 性能^[54]，使用 Iozone¹模拟测试文件碎片对性能的影响，发现随着写操作数量增加，F2FS 逻辑层碎片化变严重，影响预读命中率和 I/O 调度的合并操作，顺序读性能随着碎片数量增加而降低。

Ji 等人在实际移动平台上用碎片化程度（Degree of Fragmentation, DoF）量化了 ext4 碎片化严重程度^[55]，发现碎片导致频繁的块 I/O 请求和离散的块 I/O 模式，导致移动设备文件系统性能下降，还发现数据库文件是碎片化程度最严重的文件之一。Ji 等人又深入研究了移动设备上 ext4 产生、衡量和减少文件碎片的方法^[56]，设计了基于脚本的文件系统老化工具 AutoAge，证实了文件碎片造成用户可察觉的延迟，并评估了现有减少文件碎片的方法。

Conway 等人设计了 Git-Benchmark^[12]，使用 Git²连续同步 Linux 内核源代码和一个邮件服务器负载来模拟老化文件系统，用扫描时延和布局得分（layout score）衡量文件系统老化程度，发现老化过程速度很快，导致文件系统性能大幅下降，分析发现不论是基于机械硬盘（Hard Disk Drive, HDD），还是固态硬盘，大部分文件系统都会积累碎片，导致其性能下降，如广泛使用的 ext4^[57]、XFS^[58]、Btrfs^[59]、ZFS^[60] 和 F2FS 都会老化，Conway 所在团队研究的 Btrfs^{[61][62][63]}在实际负载测试中能有效缓解文件系统老化。Conway 等人还发现文件系统空间写满程度（fullness）与负载使用（usage）对文件系统性能的影响程度不一样^[64]，与文件系统空间写满程度相比，

¹ <https://www.iozone.org/>

² <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

文件系统性能与负载使用、碎片的产生关系更密切，并建议关注直接影响写性能、间接影响读性能的空闲空间碎片。

Jeong 等人提出 MobiBench 生成安卓系统工作负载^[24]，并使用 MobiGen¹记录和回放给定应用的用户行为生成的系统调用 trace，通过回放系统调用 trace，MobiGen 可以在没有实际人为干预的情况下重现人工驱动的 I/O 活动，配合使用 MobiBench 和 MobiGen 能老化文件系统。

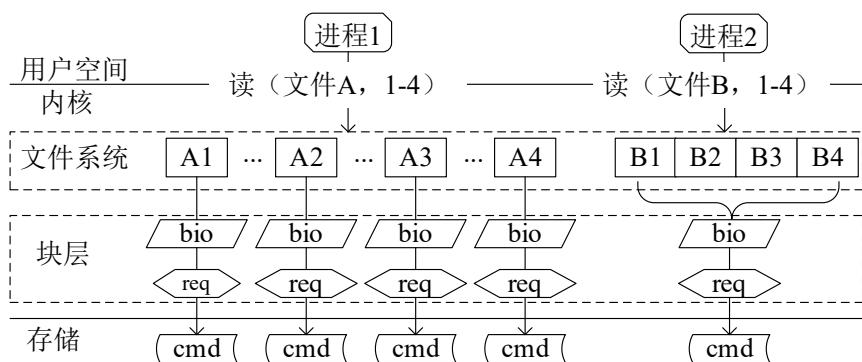


图 1.6 文件碎片分割 I/O 请求示意图^[66]

现有研究工作研究碎片的产生原因以及验证碎片对性能的影响，因为 HDD 磁盘寻道机械臂旋转的时间开销，人们很容易接受 HDD 文件系统碎片会造成性能下降，但对于 SSD 文件系统，认为 SSD 随机读写性能优于 HDD，碎片的影响不是很大，而去碎片化需要复制碎片数据，因为 SSD 的写前擦除特性，增加的读写操作会缩短 SSD 寿命。但事实并非如此，表 1.2 中现有研究工作表明 SSD 文件系统中碎片会快速积累，造成文件系统反应缓慢，导致用户体验下降。如图 1.6 所示，bio 表示通用块（block）层的一个 I/O 结构，req 表示一个请求（request）结构，cmd 表示发送给闪存存储器的 I/O 命令（command），文件 A 是碎片化的，文件 B 是连续的，进程 1 和进程 2 分别顺序读文件 A 和 B 的 4 个文件块，同样顺序读 4 个文件块，进程 1 需要 4 个 I/O 请求，进程 2 仅需要一个 I/O 请求。这是因为文件 A 的 4 个文件碎片在文件系统层分割 I/O 请求，导致在块层和存储层需要多个 I/O 请求读文件 A 的数据，

¹ <https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen>

华中科技大学博士学位论文

文件碎片增加 I/O 次数，使 I/O 请求变小，增加访问随机性。移动设备的计算存储资源和电池电量有限，移动设备和用户高度交互，用户对应用响应延迟敏感，碎片造成的设备卡顿直接导致用户不良体验，碎片的负面影响不容轻视。碎片产生与文件系统空间使用率、数据组织布局和应用 I/O 行为特征有关，碎片通过影响创建、插入、合并、排序、调度等操作，导致完成应用读写请求的 I/O 次数增加，读写请求地址的寻找次数增加，时间开销增大。此外文件碎片的影响还取决于底层存储介质^[52]，SSD 碎片影响其内部并行程度，碎片化程度增加导致读并行程度下降^[65]。空闲空间碎片导致新写文件碎片化地写入，直接影响其写性能，造成新的文件碎片，影响该新写文件的读性能。

当前日志结构文件系统性能优化研究按照研究方向可以分为三类，第一类是针对性能问题，基于应用行为特征和 I/O 访问模式解决问题，第二类是基于软硬件技术发展，使用新型存储协议或存储器件提升文件系统性能，第三类是基于应用、文件系统和底层存储硬件特征垂直整合优化。第一类研究针对实际问题，第二类研究因在目前的移动设备或服务器没有实际产品或难以获得及推广应用，大多数研究是基于模拟平台展开，实际场景中的性能优化有待验证，第三类研究需要修改与优化上下层整体，需要应用层与器件层支持修改。本文的研究工作属于第一类研究，在发现具体问题后，衡量其研究意义，经过详细分析问题后提出设计方法并评估实际场景中的效果与讨论，发现的具体问题是文件碎片积累导致文件系统性能下降、不同热度的数据混合存储导致空闲空间碎片数量及段清理开销增加、后台段清理能耗较大且减少空闲空间碎片的效果有限，接下来分析解决这三个具体问题的相关研究工作现状。

1.2.2 减少文件碎片的研究

缓解文件碎片化的方法分为两类，一类是文件碎片产生前预防碎片产生，如基于文件大小增长的预分配空间大小调整方法^[67]、延迟分配^[57]，一类是文件碎片发生后重整碎片，如复制碎片化的数据并迁移到新地址^[68]。

针对 F2FS 中文件碎片造成的顺序读性能下降，Li 等人提出多级阈值同步写方法^[69]，多级阈值同步写方法包括碎片化状态判断模块和写入模式选择模块，前者根据无效数据块数量获得碎片化程度，后者根据碎片化程度设置就地更新阈值，当同步

华中科技大学博士学位论文

写请求大小超过阈值时使用追加写，否则就地写入，以改善顺序读性能。

Ahn 等人提出一个去碎片化的文件系统 (De-fragmented File System, DFS) [70]，通过动态重新定位聚拢内存中缓存的碎片文件数据块，也聚拢同一目录下的关联小文件，使得小的碎片文件连续存储在磁盘上，以减少文件碎片和提高小文件读性能。

为了在段清理过程中减少文件碎片，Park 等人在内存中设计一个有效块队列 (Valid Block Queue, VBQueue)，在迁移要清理的段上有效数据时将所有有效块复制到 VBQueue，然后根据 inode 号重新排序有效块，使得属于同一个文件的有效数据被迁移到新段时连续[71]。该方法在 SD 卡上更为有效，而在 SSD 上的效果不佳。

Park 和 Eom 在不增加 I/O 开销的前提下提出一个抗老化日志结构文件系统 (Anti-Aging Log-structured File System, AALFS) [72]，AALFS 在段清理时基于有效块 inode 号和文件偏移量排序以减少文件碎片。Park 和 Eom 又提出一个不限文件系统的去碎片化工具 FragPicker^[66]，他们发现与机械硬盘不同，碎片导致现代存储设备性能下降的原因是请求分割，单个 I/O 请求被分割成多个请求。FragPicker 分析应用 I/O 活动，使用不受限于特定文件系统的函数且没有修改内核，只重整对 I/O 性能至关重要的碎片数据，以实现与完全重整每个碎片文件的传统去碎片化工具相媲美的性能改进水平，最小化因整理碎片引入的 I/O 数量。

Hahn 等人提出一个面向移动设备 ext4 文件系统的碎片整理工具 janusd^[65]，包括分别整理逻辑碎片和物理碎片的 janusdL 和 janusdP，文件系统层逻辑碎片增加了系统软件堆栈中的 I/O 开销，存储介质层物理碎片降低了闪存中的 I/O 并行性。JanusdL 利用闪存存储内部逻辑地址到物理地址的映射表实现无数据拷贝的逻辑碎片重整，JanusdP 重整物理层的文件碎片，但需要复制数据，只有在必要时才调用。通过自适应选择 janusdL 和 janusdP，基于定制 SSD 的实验结果表明，janusd 可以实现与 e4defrag¹相同水平的 I/O 性能提升，而不会降低闪存使用寿命。

F2FS 提供 5 种局部数据就地更新 (In-Place Update, IPU) 策略 ²，分别是

¹ <https://manpages.ubuntu.com/manpages/trusty/man8/e4defrag.8.html>

² https://android.googlesource.com/kernel/msm/+/android-7.1.0_r0.2/Documentation/filesystems/f2fs.txt

华中科技大学博士学位论文

F2FS_IPU_FORCE, F2FS_IPU_SSR, F2FS_IPU_UTIL, F2FS_IPU_SSROUTIL 和 F2FS_IPU_FSYNC 策略, 使用 ipu_policy 参数选择其中的一种就地更新策略, 默认使用 F2FS_IPU_FSYNC 策略, 当一个同步请求的数据大小小于指定阈值时就地更新。

综上所述, 表 1.3 总结了减少文件碎片的研究现状, 移动设备和服务器的应用读写特征不一样, 现有研究工作基于文件碎片如何影响 I/O 性能的分析, 从预防碎片产生和重整碎片两个角度提出减少文件碎片的策略, 但是已有减少文件碎片的研究工作还缺乏面向移动设备日志结构文件系统文件碎片和空闲空间碎片的优化技术。

表 1.3 减少文件碎片的策略

研究角度	平台	文件系统	主要特点
预防碎片产生	服务器	XFS	预分配空间大小调整方法 ^[67]
	不受限	ext4	将块分配从写操作时间延迟到页刷新时间 ^[57]
	服务器	F2FS	多级阈值同步写方法 ^[69]
		DFS	通过动态地重新定位和聚拢数据块减少碎片 ^[70]
重整碎片	服务器	F2FS	段清理过程中基于 inode 号给有效块排序 ^[71]
		不受限	只对非常影响 I/O 性能的部分数据重整碎片 ^[66]
	移动设备	ext4	在 FTL 实现无需复制数据的重整过程 ^[65]

1.2.3 区分数据热度的研究

数据热度描述数据更新或被访问的频繁程度, 不同热度的数据混合存储而更热的数据更早失效导致空闲空间碎片增加, 数据越热, 访问频率越高, 越容易异地更新, 旧版本热数据变成无效数据, 当无效数据较多、空闲空间不足时, 需要清理无效数据回收空间^[73], 而段清理成本高。由于实际 I/O 访问数据的时间和空间局部性, 数据读写间的热度倾斜普遍存在而显著, 一周内从四台不同机器收集的磁盘数据显示了不均匀的访问频率分布, 前 20% 最常访问的块占总访问量的较大百分比(45-66%)^[74]。在 TPC-C¹实际场景的测试中, 2000 个最频繁更新页面(数据库中 128 KB 页面总数的 1.6%) 的写入次数占总写入次数的 29%^[75], 即 1.6% 的频繁写入数据产生了 29%

¹ <http://www.tpc.org/tpcc/default5.asp>

的写入次数。

(1) 不同热度数据混合存储对空闲空间碎片的影响

F2FS 异地更新数据同时加剧文件碎片和空闲空间碎片化，不同热度的数据混合存储加重了空闲空间碎片化程度。如图 1.7 所示，假设一行表示一个段，文件 A 中数据热度大于文件 B 中数据热度，大于文件 C 中数据热度，在 F2FS 中执行以下操作：写 A1, B1B2, C1，追加写 A2, B3, C2C3，异地更新 B3（旧版 B3 失效），追加写 A3A4, C4，异地更新 A1A2A3A4，F2FS 定期触发后台 GC，基于收益最大化原则，后台 GC 倾向于选择有效块数量少且年龄大的段作为清理对象。在图 1.7 (a) 中，后台 GC 选择第二个段进行清理，将该段的有效块 C2、C3 复制到新段，第二个段变成整段连续的空闲空间。连续的空闲空间视为一个空闲空间碎片，整段连续的空闲空间是一个空闲段，如图 1.7 (a) 所示，按照传统 F2FS 的热度分类规则，文件 A、B、C 的数据被分类为温数据，因此写入同一个段，产生 3 个空闲空间碎片，如图 1.7 (c) 所示，在细粒度区分不同热度的数据划分规则中，文件 A、B、C 的数据热度不同，依据各自热度分别写入不同的段，只产生 1 个空闲空间碎片，所以不同热度的数据混合存储增加了空闲空间碎片数量。

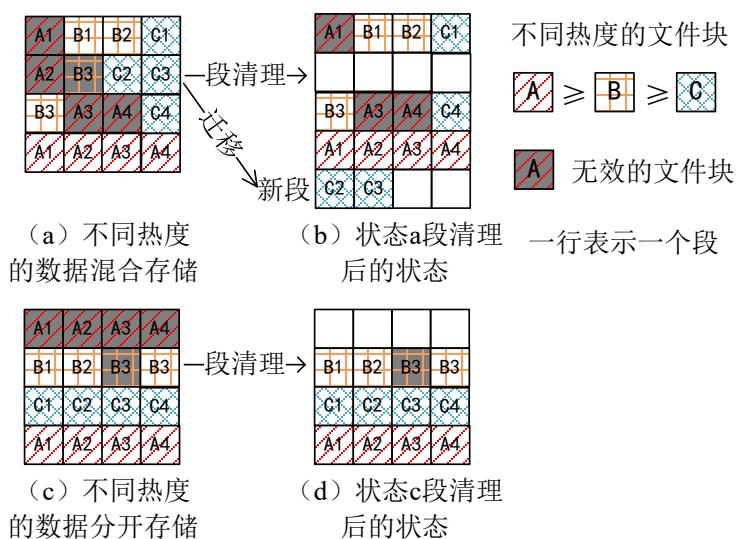


图 1.7 不同热度的数据混合存储导致空闲空间碎片增加示意图

(2) 不同热度数据混合存储对段清理的影响

不同热度的数据混合存储增加段清理开销，相比于图 1.7 (c) 文件 A 更新后整

华中科技大学博士学位论文

段失效，无需迁移有效数据块就能获得图 1.7 (d) 第一行表示的空闲段，图 1.7 (a) 获得空闲段需要迁移两个有效块，段清理后得到如图 1.7 (b) 第二行表示的空闲段，并在第五行表示的新段上写两个数据块 C2C3。区分数据热度存储的图 1.7 (c) 段清理后到图 1.7 (d) 无需迁移有效数据块，而不同热度数据混合存储的图 1.7 (a) 段清理后到图 1.7 (b) 需要迁移两个有效数据块，这说明不同热度的数据混合存储加剧段清理开销，图 1.7 也说明了 F2FS 异地更新数据导致文件碎片数量增加。

区分数据热度并分开管理不同热度的数据，可以减少不必要的读写操作，延长闪存使用寿命。一些工作分离文件系统中的冷热数据，以减少空闲空间碎片和垃圾回收开销。对于常用闪存文件系统之一的 UBIFS，刘景超设计基于多个哈希函数的哈希表来统计冷热数据，将冷数据和热数据分开存放，以减少垃圾回收次数^[76]。Kang 和 Eom 提出一种冷热数据分离方法^[77]，利用页缓存中隐藏信息动态识别文件系统中块的热度，以有效减少数据复制操作。

Min 等人基于 Linux 日志结构文件系统提出一个新的 SSD 文件系统(File System for SSDs, SFS)^[14]，SFS 将所有文件系统级的随机写操作转换为 SSD 级的顺序写操作以获得最大带宽，统计块写次数和频率，迭代量化所有段的热度，获得分组标准以分离冷热数据段，使得段使用率形成明显双峰分布以减少段清理开销，但其热度识别过程不能适应 I/O 模式热度变化。为减少 F2FS 的段清理开销，Li 等人提出高检测频率后台段清理机制^[69]，根据空闲空间大小动态调整后台段清理的频率。

一些工作分离磁盘或 SSD 中冷热数据以减少垃圾回收开销，对于区分数据热度，减少文件系统空闲空间碎片同样具有指导作用。Wang 等人提出混合日志结构(Hybrid Log-structured, Hylog)^[78]磁盘布局，采用日志结构方式写热页提高写性能，采用覆盖方式写冷页降低清理成本，将一个页写入磁盘之前，HyLog 使用一个基于写代价模型的分离算法确定该页是否是热页。

Xie 等人提出了一种自适应分离感知闪存转换层(Adaptive Separation-Aware Flash Translation Layer, ASA-FTL)^[79]，该层使用采样实现轻量级分离标准识别，设计选择性缓存机制节省 SSD 中有限的 RAM 资源，使用数据聚类以低成本准确识别和分离冷热数据，有效减少闪存垃圾回收开销，提升性能。

华中科技大学博士学位论文

Li 等人提出了一种基于热度感知机器学习（Hotness-Aware Machine Learning, HAML）的 SSD 管理方法 HAML-SSD^[80]，以减少 SSD 垃圾回收开销，根据更新频率和平均更新时间间隔定义热度，使用二维的 K 均值（K-means）聚类算法动态聚类热度相似的数据并存储。

Chiang 等人提出动态数据聚类（Dynamic dAta Clustering, DAC）方法^[81]，使用访问频率动态分类数据，在数据更新和段清理期间进行聚类，以减少复制数据量和擦除操作数量，不需要复杂计算来确定数据是热还是冷的，设计了一个自适应清理管理器动态调整清理策略，以响应数据访问行为的变化。

Kim 等人提出 PCStream^[82]，使用 K-means 聚类算法根据支持的流的数量对具有相似数据生命周期的程序上下文进行分组，以分离冷热数据，减少多流 SSD 的垃圾回收开销，一个程序上下文表示一个程序的执行路径，能有效表示主要的 I/O 活动。

表 1.4 区分数据热度的技术

区分方法	方案名称	主要贡献
静态分类	UBIFS 优化 ^[76]	使用多哈希函数的哈希表统计冷热数据
	hint ^[77]	利用页缓存中隐藏信息动态识别文件系统中块的热度
	SFS ^[14]	计数块的写次数和频率，获得分组标准分离冷热数据段
	Hylog ^[78]	基于写代价模型的分离算法区分冷热页
动态分类	ASA-FTL ^[79]	在 Flash 转换层使用数据聚类分离冷热数据
	HAML-SSD ^[80]	基于更新频率和平均更新时间间隔，用聚类算法区分热度
	DAC ^[81]	根据数据的访问频率对数据进行动态分类
	PCStream ^[82]	基于数据生命周期使用聚类算法对程序上下文进行分组

区分数据热度的工作如表 1.4 所示，不同热度的数据混合存储加剧了空闲空间碎片化程度，分离热数据与冷数据进行数据重组可以减少清理开销，现有研究工作从准确定义热度、精准区分冷热数据和妥当放置冷热数据等方面展开研究，分别使用静态分类和动态分类方法分离冷热数据。实验分析发现日志结构文件系统温数据体量较大且不同热度的数据混合，迫切需要一个精准区分温数据热度的方法，然而现有区分文件系统数据热度的方法不够精准或无法随 I/O 模式变化动态地识别数据热度。

1.2.4 减少段清理的研究

传统的日志结构文件系统性能依赖于连续空闲空间，维护连续空闲空间需要昂贵的清理操作回收无效块，迁移有效块时引入大量读写操作，而空闲空间碎片增加清理开销。日志结构文件系统有两种管理空闲空间的方法：线程（threading）写入和复制（copying）数据^[2]。线程写入方法把文件系统中的无效块当作空闲块，直接覆盖写。复制数据方法先选择一个段作为清理对象，识别该段中的有效块，并将有效块复制到新段，回收清理对象作为空闲段用于后续数据写入，即段清理方法。F2FS 一般使用段清理方法管理空闲空间，段清理回收离散的无效块为接下来的日志结构写数据提供空闲段。F2FS 以节为单位执行清理过程，实际应用的 F2FS 默认设置一个节由一个段组成，清理过程称为段清理（Segment Cleaning, SC）。F2FS 中有两种段清理方式，前台段清理（Foreground SC, FSC）和后台段清理（Background SC, BSC），只有当写请求没有足够空闲空间时才触发前台段清理，定期唤醒内核线程触发后台段清理，在选择清理对象时，前台段清理使用贪心算法，后台段清理使用成本-收益（Cost-Benefit, CB）算法。触发前台段清理时会阻塞日志结构文件系统写请求，增加 I/O 响应延迟。Seltzer 等人分析了日志结构文件系统的清理开销，在事务处理环境中，当磁盘写到 48% 时，清理开销导致日志结构文件系统性能降低 34% 以上^[17]。

段清理过程包括三步，选择清理对象、标识及迁移有效数据、等待检查点同步回收被清理的段，现有研究工作通过优化脏段中有效数据的分布，优化段清理选择的对象、触发时机和频率，以减少段清理触发次数，减少段清理开销。

Zhang 等人观察到写流量较大时，闪存设备内部并行性没有得到充分利用，提出了一种并行性感知的文件系统 ParaFS^[15]。ParaFS 基于定制 FTL 的闪存设备，利用闪存通道级并行性在维持冷热数据分离的同时实现二维数据分配，协调文件系统级和 FTL 级的垃圾回收过程，优化读写和擦除请求的调度，提高了写密集型负载的性能。

Gwak 等人提出一种 GC 记录技术以减少日志结构文件系统的段清理开销^[83]，使用日志只记录与 GC 进程相关的文件系统修改，不用引入高成本检查点操作保证文件系统一致性。Gwak 和 Shin 进一步分析检查点开销，实现段清理日志（Segment Cleaning Journaling, SCJ）^[84]技术，在一个特殊的日志区域记录段清理的块迁移信息，

华中科技大学博士学位论文

做段清理时不用触发检查点操作，避免引入不必要的写，考虑到等待检查点释放的无效块数量比较多时影响文件系统性能，Gwak 和 Shin 在元数据更新较多和检查点时间间隔较久时做检查点操作。

Park 等人观察到 Android 智能手机的挂起模式与触发后台段清理操作间的冲突，提出挂起感知段清理技术 (Suspend-aware SC)，在手机屏幕关闭后启动后台段清理，在手机达到实际挂起状态时停止后台段清理^[85]。针对 F2FS 段清理影响性能的问题，Li 等人提出高检测频率的后台段清理(High Frequency BSC)方法^[69]，称为 MWHFB，该方法根据空闲空间大小动态调整后台段清理的频率，能提高段清理性能，降低后台段清理对闪存寿命的影响。与 Suspend-aware SC 技术不同的是，MWHFB 可以根据空闲空间大小动态调整后台段清理频率。

Wu 等人提出强化学习辅助的后台段清理 (Reinforcement Learning assisted Background segment Cleaning, RLBC)^[86]方法，通过学习 I/O 工作负载行为和逻辑地址空间状态，基于强化学习自适应地决定何时触发后台段清理，达到平衡系统性能和存储器寿命的目的。为了减少移动场景的内存开销，Wu 等人进一步提出了基于剪枝的多目标深度强化学习后台段清理方法 (deep MultiObjective reinforcement learning-based Background segment Cleaning, MOBC)^[87]，MOBC 考虑移动场景中所有相关元素，自适应地做出清理决策，减少了块迁移数量和段清理触发次数，同时提出基于稀疏多径多层次感知器的结构化剪枝方法进行稀疏化处理，减少了时间和空间开销。

此外，Yu 等人提出最新的 F2FS 垃圾回收策略 ATGC (Age-Threshold based Garbage Collection)¹，被集成到 Linux 5.10 及以上版本的内核。ATGC 基于年龄阈值进行垃圾回收，根据定义的年龄阈值筛选较年长的清理对象，ATGC 是除前台垃圾回收和后台垃圾回收以外 F2FS 最新使用的垃圾回收策略，对于测试用例的垃圾回收次数从 162 次下降到 75 次²，明显提高了文件系统垃圾回收的效率。

¹ <https://lwn.net/Articles/828027/>

² <https://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs.git/commit/?h=dev&id=9aa9b90842087b6d388b6ad7fae9918df4e5691c>

华中科技大学博士学位论文

综上所述，日志结构文件系统优化段清理的工作如表 1.5 所示，现有工作从管理数据分布和调整段清理时机、频率、对象方面展开研究，日志结构文件系统段清理读写开销大，分析发现一次后台段清理的能耗较大而减少空闲空间碎片的效果有限，需要基于日志结构文件系统数据分布提出高能效回收无效空间的方法。

表 1.5 日志结构文件系统优化段清理技术

设计角度	方案名称	主要特点
管理数据分布	ParaFS ^[15]	基于闪存通道并行性设计二维数据分配
	SCJ ^{[83][84]}	使用日志记录段清理时文件系统的修改
	Suspend-aware SC ^[85]	挂起感知的后台段清理技术
调整段清理时机、频率、对象	MWHFB ^[69]	高检测频率的后台段清理策略
	RLBC ^[86]	基于强化学习自适应决定何时触发后台段清理
	MOBC ^[87]	基于剪枝的多目标深度强化学习后台段清理方法

1.3 研究内容与论文组织结构

移动设备爆炸性增长和 SSD 大规模存储系统高速发展为日志结构文件系统性能优化带来巨大挑战，日志结构文件系统由于自身的异地更新和段清理机制，面临着文件碎片造成系统卡顿、空闲空间碎片增加段清理频率、段清理增加读写开销等严重问题。本文针对文件碎片和空闲空间碎片，基于数据读写特征合理使用机器学习算法来改善数据组织布局，进而提升 I/O 性能，缩短应用执行时间，并降低能耗，所提出的日志结构文件系统性能优化技术适用于移动场景和服务器场景，并且可以方便地从 F2FS 移植到其他日志结构文件系统。所提出的策略与负载特征和 I/O 模式相关，移植到不同场景时，可以通过分析各自场景的负载特征后重新训练模型或修改参数来最大化性能收益。

本文研究内容的组织结构如图 1.8 所示，主要研究内容在第 2 章至第 4 章介绍。针对文件碎片积累降低顺序读性能、增加应用启动时间的问题，第 2 章基于应用行为特征提出一种基于机器学习算法的自适应预留空间策略；针对不同热度的温数据混合存储增加空闲空间碎片、加剧段清理开销的问题，第 3 章提出使用 K-means 聚

华中科技大学博士学位论文

类算法区分数据热度，并基于细粒度数据热度实现多日志延迟写策略；针对一次后台垃圾回收能耗较大而后台垃圾回收减少空闲空间碎片效果有限的问题，第4章提出空闲空间碎片感知的垃圾回收策略。第2章从文件空间分配角度研究空间预留技术以延缓碎片产生，第3章从多文件数据组织布局角度研究细粒度区分数据热度以减少空闲空间碎片产生，第4章从无效空间高能效回收角度研究空闲空间碎片感知的垃圾回收以提高单位能耗减少空闲空间碎片的效率。本文围绕着减少文件碎片和空闲空间碎片优化日志结构文件系统性能展开研究，主要研究内容如下：

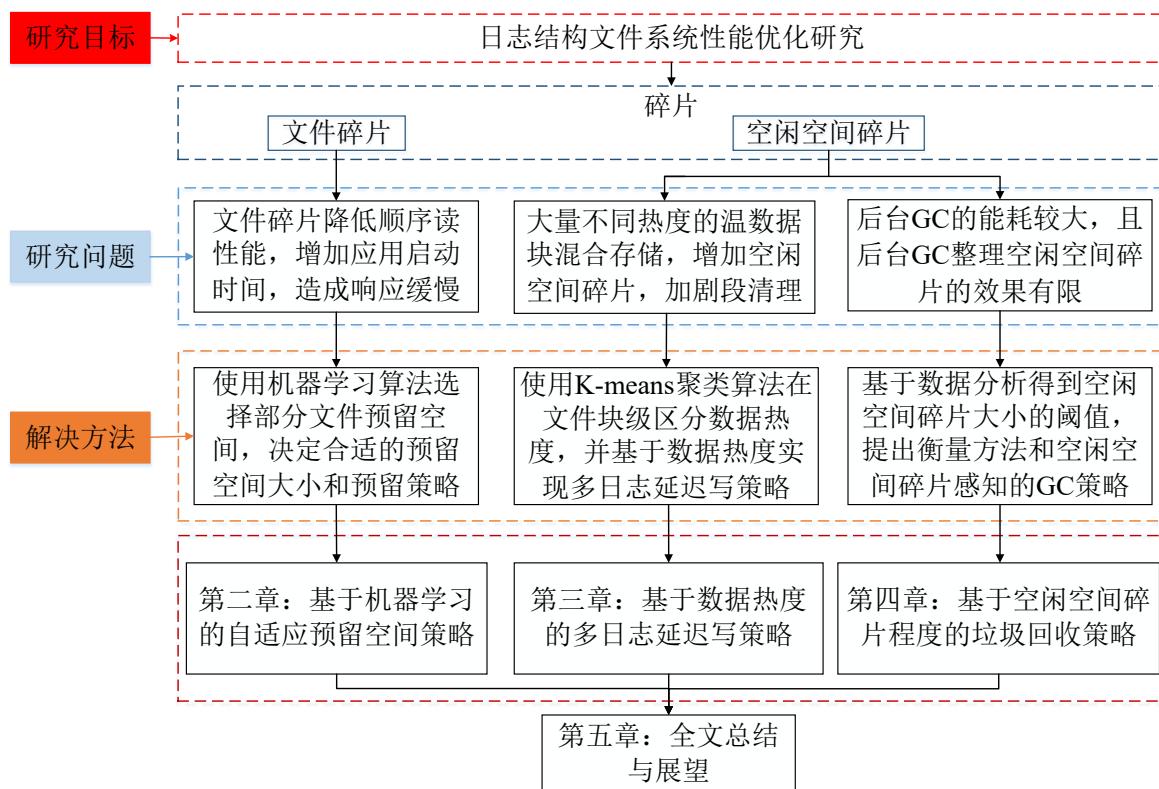


图 1.8 论文研究内容和组织结构

(1) 基于机器学习的自适应预留空间策略 (Adaptive Reserved Space based on Traceback, ARST)。文件碎片和空闲空间碎片是引起文件系统老化、造成文件系统性能下降的重要原因。因为移动设备与用户高度交互，要求 I/O 响应延迟越短越好，而碎片造成的 I/O 响应延迟不可忽略。预留空间是直观减少文件碎片的方法，但是为所有文件预留空间会浪费空间，提出基于追溯的自适应预留空间策略，挑选部分文件为其预留空间，这部分文件称为预留文件。该策略选择与碎片产生强相关的文件特征

华中科技大学博士学位论文

构造数据集，对于小型数据集，追溯到写文件描述符（file descriptor, fd）对应的写文件之后，根据文件读写特征在分配空间时决定是否预留；对于普通的大型数据集，基于追溯写 fd 获得的文件构造数据集，使用决策树算法选择预留文件；对于历史信息较少的大型数据集，使用随机森林或 AdaBoost 算法选择预留文件。预留文件在预留空间内就地更新，普通文件遵循传统 F2FS 的异地更新，为了不浪费预留空间，文件系统空间使用率小于 70% 时，所有预留文件就地更新；空间使用率在 70% 到 80% 之间时，已分配空间的预留文件就地更新，新写预留文件不再预分配空间；空间使用率大于 80% 时，所有文件都异地更新。ARST 策略在第 2 章进行介绍。

(2) 基于动态识别的细粒度数据热度管理 F2FS 温数据，提出多日志延迟写策略（Multi-log delayed writing based on Hotness, M2H）。F2FS 将用户写的常规文件数据分类为温数据，温数据占比至少 80%，实际应用场景中温数据的文件块间热度差异大，热度各异的温数据通过一个 log 写入，相对较热的数据较早失效导致空闲空间碎片增加，而较冷的数据仍然有效导致段清理有效块迁移数量增加。M2H 改善多文件数据组织布局，基于工作负载读写特征，选择性使用文件块更新距离、最近使用距离和读次数定义数据热度，基于段清理次数的敏感性实验决定 K-means 算法的 K 值，使用 K-means 聚类算法准确识别热度，基于聚类的细粒度数据热度划分设计了多日志写入，并延迟提交多日志以避免写性能下降，使用段上有效块的平均热度优化段清理对象的选择，分别管理被清理段（victim segment）上的有效块和无效块以减少检查点操作，减少段清理开销，提高 F2FS 整体性能。M2H 策略在第 3 章进行介绍。

(3) 基于数据分析获得空闲空间碎片大小的阈值，提出衡量方法和空闲空间碎片感知的垃圾回收策略（free space Fragmentation Aware GC, FAGC）。移动设备提供有限的电量供应，存储系统软件堆栈的能耗占系统总能耗比例较高，日志结构文件系统被广泛用作移动设备文件系统，通过构造不同场景深入研究基于 F2FS 的移动设备能耗，基于能耗的分析提出 FAGC 策略以提高单位能耗减少空闲空间碎片的效果。实验分析发现一次后台垃圾回收的能耗比较大，而后台垃圾回收减少空闲空间碎片的效果有限，重新评估多大的空闲空间是空闲空间碎片，基于数据分析得到空闲空间碎片的阈值，提出空闲空间碎片系数来衡量空闲空间碎片化程度，并提出选择空闲空

华 中 科 技 大 学 博 士 学 位 论 文

间碎片严重的段做段清理对象和使用线程日志写方式迁移有效块的 FAGC 策略，以减少垃圾回收次数和提高每次后台 GC 减少空闲空间碎片的效率，实现无效空间的高能效回收。FAGC 策略在第 4 章进行介绍。

2 自适应选择文件的空间预留策略

随着文件系统不断创建、删除、更新文件，会产生文件碎片，累积的碎片造成文件系统老化，I/O 响应时间增加。由于移动应用大量的随机同步小写请求以及日志结构文件系统异地更新的特征，F2FS 碎片化问题突出，导致文件系统性能下降明显。用户与移动设备的高度交互要求 I/O 请求的快速响应，移动设备有限的计算存储资源使得文件碎片的负面影响更加明显，文件系统碎片化问题亟待解决。减少文件碎片的方法通常分为两种，一种是碎片严重后通过复制数据重整，另一种是碎片产生前延缓碎片产生。

分析碎片的产生原因、碎片对性能的影响及现有的去碎片化方法，发现现有研究缺乏面向移动设备日志结构文件系统去碎片化的有效解决方法。文件空间预留能使碎片文件数据变得连续，有效减少文件碎片，但移动应用会产生大量文件，为所有文件预留空间会浪费移动设备本就不富余的存储空间，在大量文件中选择合适的文件为其预留空间是难点，决定合适的预留空间大小以及合理地打开和关闭预留功能是不浪费空间的关键。通过追溯文件描述符得到写入文件，选择文件读写特征构造数据集，提出基于机器学习算法选择预留文件的自适应预留空间策略（Adaptive Reserved Space based on Traceback，ARST），研究了文件碎片对系统性能的影响，分析了应用 I/O 行为特征，详细阐述了 ARST 的设计原理和实现，在实际平台上实现了 ARST 策略，进行了系统的测试并给出实验评估结果与分析结论。

2.1 减少文件碎片的需求和挑战

在文件系统使用过程中，随着文件的创建、更新和删除，碎片数量快速增加，使得设备响应迟缓，导致时间敏感的用户使用体验下降，造成智能手机使用过程中的系统卡顿。碎片通常分为两类，一类是文件碎片，文件碎片是单个文件离散分布的数据片段，另一类是空闲空间碎片，未回收的无效数据块都属于空闲空间碎片，连续的文件数据或无效数据视为一个文件碎片或空闲空间碎片，未回收的无效块在 F2FS 段清理时会被回收。文件碎片不仅降低文件系统的性能，还会降低闪存读写性能，缩短闪

存设备的寿命。对文件系统而言，碎片的数量越少越好，但在使用过程中碎片不断增加，其负面影响不可轻视，是亟需解决的重要问题。

减少碎片的方法通常分为两类，一类是碎片产生后，在碎片严重影响文件系统性能时，重整碎片以减少碎片数量。这种方法需要寻找关联性，将离散的单个小文件或文件组的数据重新组织到一起，此过程需要复制数据，复制数据会增加读写数量，缩短存储器件寿命。另一类是在碎片产生前聚拢文件数据，避免单个文件的数据被分割为多个文件碎片，包括延迟文件数据提交、为文件划分元组以及为文件预留空间使得文件数据存放在预留空间内，达到减少碎片的目的，这类方法的难点在于需要精准确定对象，否则会延长文件系统响应时间，浪费存储空间。

2.1.1 文件碎片对性能的影响

分析现有减少碎片的研究工作发现缺少优化移动设备日志结构文件系统的文件碎片与空闲空间碎片的方法，在合成负载和实际应用负载上衡量文件碎片对日志结构文件系统性能的影响，并发现文件碎片影响系统性能的规律。

移动应用每 10 天左右更新一次¹，在频繁的更新过程中产生的碎片数量是庞大的。研究过程中基于华为 Mate10 Pro²使用 ftrace³抓取各个应用的 trace，基于 HiKey960 development platform⁴（在下文简称 Hikey 960）代码的开源性和编译的便捷性，快速完成 ARST 策略的实现与评估，使用 Hikey960 测试移动存储系统碎片的影响。研究在传统异地更新 F2FS 和局部数据就地更新（In-Place Update，IPU）的 F2FS 中碎片是否快速积累，并衡量碎片对移动设备文件系统性能的影响是否严重。测试 IPU 性能时采用 F2FS 默认使用的 F2FS_IPU_FSYNC 策略，碎片化程度比较严重的文件是 SQLite 数据库文件^[55]，当用户在前台使用一个应用时，应用在后台以多线程模式执行任务，如脸书创建 18 个并发线程，并行地写入不同的 SQLite 文件中^[56]，所以使

¹ <https://www.nowsecure.com/blog/2015/06/08/understanding-android-s-application-update-cycles/>

² <https://consumer.huawei.com/en/phones/mate10-pro/>

³ <https://www.kernel.org/doc/Documentation/trace/ftrace.txt>

⁴ <https://www.96boards.org/product/hikey960/>

用数据库文件构造合成负载。

(1) 合成负载

当碎片数量增加时，文件顺序读性能显著下降，在传统 F2FS 和 IPU 中使用 fio¹ 构造连续文件和碎片文件，比较它们的碎片数量和文件顺序读性能。一个典型的安卓系统工作场景是多个文件写请求交叉伴随着同步命令，用 fio jobfile 构造交叉写入请求，其中连续文件的场景是目标文件顺序写，其他 7 个文件交叉随机写，碎片文件的场景是目标文件与其他 7 个文件交叉随机写，其中文件大小为 128 MB，块大小为 4 KB，所有写请求都同步写入数据，再测试目标文件的顺序读性能。分析发现在 Hikey960 和华为 Mate10 Pro 上，fio 构造的连续文件顺序读性能优于碎片文件，在传统 F2FS 和 IPU 中，连续文件的碎片数量是 1，而碎片文件的碎片数量在 32764 到 32768 之间。

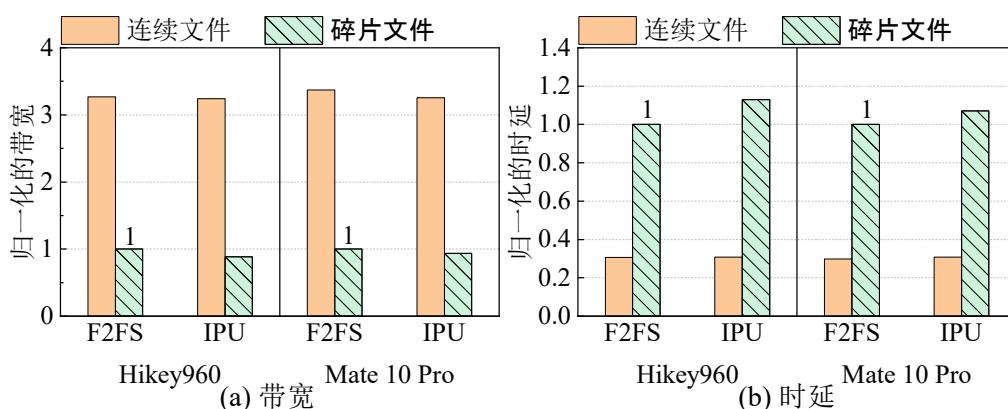


图 2.1 连续文件与碎片文件的顺序读性能比较

以 Hikey960 与华为 Mate 10 Pro 上传统 F2FS 顺序读碎片文件的带宽和时延为基准，图 2.1 展示了连续文件与碎片文件的顺序读性能，可以看到传统 F2FS 中碎片文件的顺序读带宽明显下降，在 Hikey960 上降幅为 69.39%，在 Mate10 Pro 下降了 70.35%，顺序读时延相应地增加了 2.27 倍和 2.36 倍。图 2.1 中 IPU 表示使用 F2FS_IPU_FSYNC 就地更新策略的 F2FS，从图 2.1 中看到 IPU 的实验结果与传统 F2FS 相近，在 Hikey960 和 Mate10 Pro 上 IPU 的碎片文件顺序读性能分别下降 72.75%

¹ https://fio.readthedocs.io/en/latest/fio_doc.html

和 71.19%。仅同时随机写 8 个文件就会积累大量文件碎片，顺序读性能也显著下降，说明碎片积累对系统性能影响明显，实验结果表明碎片对智能手机和 Hikey960 开发板的性能影响一致。

分析文件碎片数量对文件系统性能的影响以及 IPU 能否有效减少文件碎片，采用 fio 随机写文件，通过调整同步写大小，构造多个碎片数量不同的 128 MB 大小的文件。测试了同样大小，不同碎片数量文件的顺序读性能，如图 2.2 所示，当碎片数量越多时，文件顺序读时延越长。见图 2.2 中红色 A 点，当碎片数量少于 128 时，碎片对顺序读时延的影响略小，见图 2.2 中蓝色 B 点，当碎片数量大于 2048 时，碎片对顺序读时延的影响较严重。可以看到传统 F2FS 和 IPU 之间的顺序读时延没有太大差异，IPU 稍微减少了单个文件的碎片。当文件系统比较满时，较大的连续空闲空间很少，新写入文件被迫写成多个文件碎片，导致文件顺序读性能比较差。

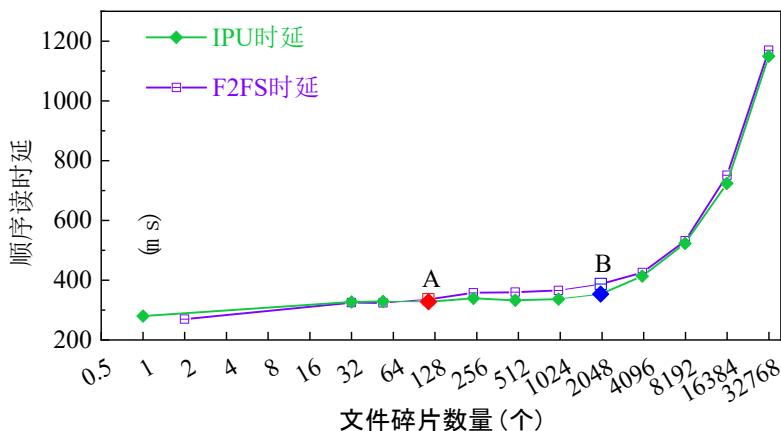


图 2.2 同样大小的文件，碎片数量不同时的顺序读性能

测试 IPU 策略中文件碎片大小均为 4 KB，不同大小的文件顺序读性能，实验结果如图 2.3 所示，发现当碎片数量越多时，文件顺序读时延越长。考虑到 64% 的 I/O 操作请求大小小于 4 KB^[24]，设置图 2.3 中 IPU 碎片文件的碎片大小为 4 KB，连续文件是模拟理想场景，连续写一个文件，该文件仅包含一个文件碎片。用图 2.3 上面的横轴坐标表示文件大小，对应的纵坐标表示连续文件的顺序读时延，在图 2.3 中用红色圆点表示；用下面的横轴坐标表示碎片文件的碎片数量，对应的纵坐标表示碎片文件的顺序读时延，在图 2.3 中用黑色方点表示，横轴坐标相近的连续文件和碎片文件

的大小相同。在文件系统可用空闲空间非常不足时, F2FS 采用线程式的日志结构写方式, 大文件的文件碎片数量越多, 其读 I/O 请求数量越多。如图 2.3 中蓝色 A 点所示, 当文件碎片数量为 512 时, 同样大小的碎片文件与连续文件之间的顺序读时延差值变大, 到蓝色 B 点附近, 当文件碎片数量为 4096 时, 同样大小的碎片文件与连续文件之间的顺序读时延差距非常明显, IPU 无法有效缓解碎片带来的负面影响。当文件系统写得较满时, 会出现严重的空闲空间碎片, 大文件被分割写导致大量的文件碎片。在多线程并发同步写的场景下, IPU 几乎不能减少文件碎片, 仅可以稍微减少单个文件碎片, 所以 IPU 减少文件碎片的效果有限, 需要其他更加有效的方法。

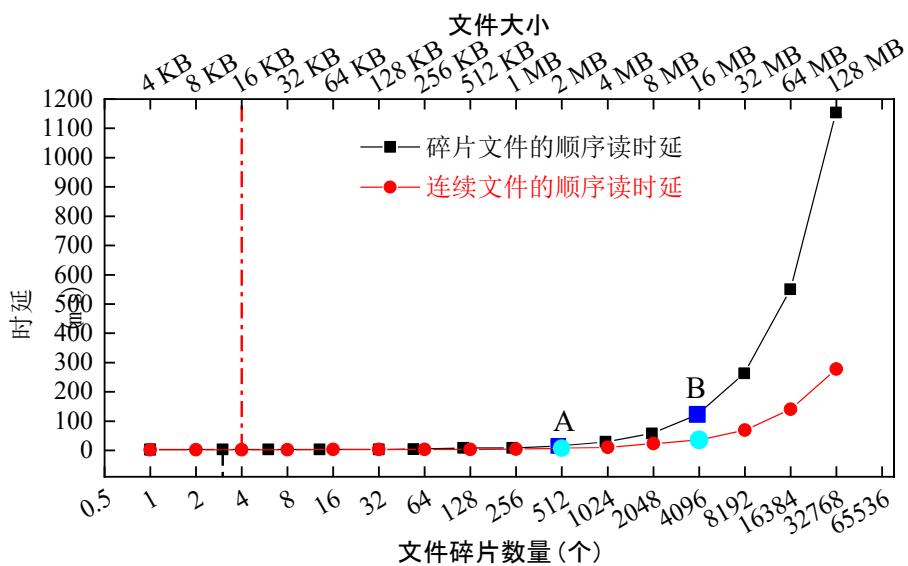


图 2.3 包含不同数量的 4 KB 碎片的文件顺序读性能

(2) 实际应用负载

在文件系统空间使用率一致的前提下, 分析实际应用场景不同程度的文件碎片对应用启动时间的影响, 使用安卓调试桥 (Android Debug Bridge¹, adb) 调用活动管理器 (Activity Manager², am) 命令启动应用并测量应用启动时间, 启动应用时没有任何预加载数据。分别选择社交、搜索、游戏、购物和地图类的典型应用微信、百度、

¹ <https://developer.android.com/studio/command-line/adb>

² <https://developer.android.com/reference/android/app/ActivityManager>

华中科技大学博士学位论文

王者荣耀、淘宝和百度地图，以初始化文件系统中微信、百度、王者荣耀、淘宝和百度地图等应用的启动时间为基准，一个场景是通过腾讯视频下载一部 972 MB 大小的电影后测试各个应用启动时间，该场景下系统文件碎片化程度较轻。另一个场景是通过百度网盘下载大量图片且所有图片大小之和与上一部电影相同，测试五个应用的启动时间，这些图片存放在 6 个文件夹，分别包含 2222、2110、2440、1000、922 和 1075 张图片，该场景下系统文件碎片化程度较重。通过 `df -h -k` 命令监控下载大量图片时文件系统使用的存储空间大小，使最终使用存储空间的大小总和与下载一部电影时系统使用的存储空间大小一致。

表 2.1 文件碎片和空闲空间碎片共用的碎片分级和大小

碎片大小分级	碎片 X 的大小（单位：KB）
第一级	$0 < X \leq 4$
第二级	$4 < X \leq 8$
第三级	$8 < X \leq 16$
第四级	$16 < X \leq 32$
第五级	$32 < X \leq 64$
第六级	$X \leq 128$
第七级	$X > 128$

观察到文件系统中有大量 4 KB 大小的文件碎片和空闲空间碎片，FS 的第一级碎片大小从 4 KB 开始，而大于 128 KB 的碎片数量较少，所以将碎片分为七级，拓展 Ji 等人提出的碎片大小（Fragment Size, FS）^[56]用于统计文件碎片和空闲空间碎片数量。不同层次的碎片大小如表 2.1 所示，其中分类的级别数可以根据碎片数量分布的实际情况调整。F2FS 读写数据以逻辑块为单位，一个块大小为 4 KB，表 2.1 中一个碎片的大小是 4 KB 的整数倍，如第三级碎片包含 12 KB 和 16 KB 的碎片。从 /proc 中通过 `segment_bits` 获取有效块的信息，下载一部电影与下载大量图片场景的文件碎片数量如表 2.2 所示。下载一部电影和下载大量图片两种场景的文件系统空间使用率一样，但下载大量图片的场景中文件碎片数量更多，系统中文件碎片化程度更加严重，下载一部电影的场景下系统文件碎片化程度较轻微。

华中科技大学博士学位论文

表 2.2 不同场景下系统包含的文件碎片数量(个)

碎片大小分级	第一级	第二级	第三级	第四级	第五级	第六级	第七级
下载一部电影	2175	1169	433	253	242	135	216
下载大量图片	37852	11569	10616	5215	1716	1076	1024

在初始化系统和构造的两种场景下典型应用的启动时间如图 2.4 所示, 分别表示在初始化状态的 F2FS、下载一部电影后文件碎片化程度较轻的 F2FS 和下载大量图片后文件碎片化程度较重的 F2FS 中启动典型应用花费的时间, 启动时间指使用 adb 命令唤醒应用, 加载应用数据, 到应用图标显示的时间。在华为 Mate10 Pro 上比较不同文件碎片化程度的场景下应用启动时间, 碎片化程度较重的文件系统中应用启动时间平均增加 11.58%。碎片化越严重, I/O 请求被分割的次数越多, 导致 I/O 请求数量增加, 严重降低顺序读性能, 直接影响应用请求响应速度。应用每次启动时, 加载的数据稍有变化, 在应用的更新过程中, 移除旧版本安装包, 写入新版本安装包, 应用更新后, 应用安装包数据会变化, 随着碎片数量增加, 应用安装包文件数据布局变得离散, 应用启动需要加载的文件数据也因此变得离散, 导致启动时间增加, 用户使用智能手机一段时间后会因此感受到卡顿。

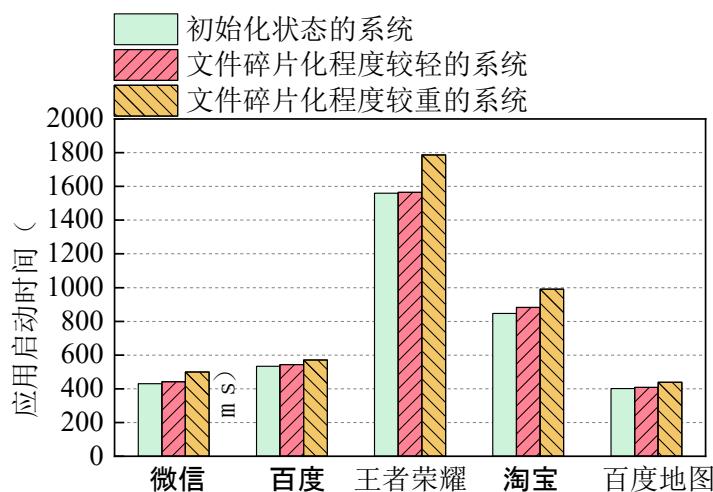


图 2.4 不同文件碎片化程度的系统中各个应用启动时间

2.1.2 减少文件碎片的挑战

碎片的存在使得本能顺序读写的数据变成随机分布, 随机读写比顺序读写消耗

更多能量^[16]，但智能手机的电量是有限的。上述实验证明 F2FS 的文件碎片是一个严重的问题，虽然移动存储系统的性能受到多方面因素的影响，但碎片化是导致 I/O 吞吐量下降和响应时间变长的关键因素。由于 FTL 和闪存存储对用户不可见，所以本章不考虑存储设备的物理碎片，仅对文件系统中的逻辑碎片进行优化，且文件系统的性能下降主要来自逻辑碎片^[65]。

现有研究常用复制数据的方法重整文件碎片使文件数据连续，然而该方法需要复制数据，导致读写开销增加，闪存寿命缩短，需要一种无需复制数据的策略保证文件数据连续性，有效减少文件碎片的方法。

实验结果表明连续文件的 I/O 性能优于碎片文件，应尽可能令文件数据连续，预留空间方法提升数据连续性，主动将同一个文件数据聚拢在相同区域，可以减少产生文件碎片的可能性，预留空间的不理想状态是没有使用完预留空间，造成存储空间浪费和空闲空间碎片。移动设备上有大量应用交叉写不同文件，如果为所有文件都预留空间，则会浪费预留空间，观察到有些文件比其他文件更加频繁地读写，需要有选择性地为会不断增长、不断更新、产生旧版本数据的文件预留空间，如何高效地选择预留文件成为关键。需要深入挖掘文件碎片产生相关的读写特征，基于移动应用的读写行为特征提出有效的文件预留空间减少碎片方法，难点在于找到会严重碎片化的预留文件、合适的预留空间大小和开启关闭预留功能的时机，达到以较低成本提高移动设备日志结构文件系统性能的目的。

2.2 预留文件选择方法

2.2.1 架构设计

图 2.5 通过一个具体的选择文件预留空间示例，展示了传统 F2FS、所有文件预留空间方法与选择性预留空间策略的区别。图 2.5 中文件数据的写操作顺序依次是写 A1A2，B1B2，C，更新 A2，追加写 B3，A3，更新 B2，写 D1D2。如图 2.5 所示，左侧的传统 F2FS 异地更新数据，在第③步，文件 A 和 B 存在文件碎片，在第④步，当要写 D1D2 时，因为空闲空间不足需要触发文件系统级的垃圾回收，回收无效数据块，获得足够多的空闲空间。中间部分展示所有文件预留空间方法，在第④步时，

华中科技大学博士学位论文

因为剩余空闲空间不足，可用空闲空间被文件 C 占用，需要释放文件 C 未使用的预留空间，重新将其分配给文件 D。考虑到文件 A、B 追加写和更新文件数据，ARST 策略选择为文件 A、B 预留空间，而文件 C 使用传统日志结构方式写，在第③步文件 A、B 的数据更新后，在预留空间内文件 A、B 的数据仍是连续的，在第④步写文件 D 的数据时空闲空间足够，不需要其他操作。比较发现在第④步，传统 F2FS 和所有文件预留空间方法完成文件 D 写入花费的时间比 ARST 策略多，在传统 F2FS 中接着顺序读碎片化的文件 A 和 B 的数据，因为存在文件碎片会导致顺序读延迟增加。所有文件预留空间方法容易浪费空间，导致空闲空间碎片化，从图 2.5 示例中发现仅写一次之后不再更新与追加写的文件 C 不是合适的预留文件，为了在减少文件碎片和不浪费空闲空间并维护空闲空间连续之间较好地权衡，ARST 需要选择合适的预留文件。

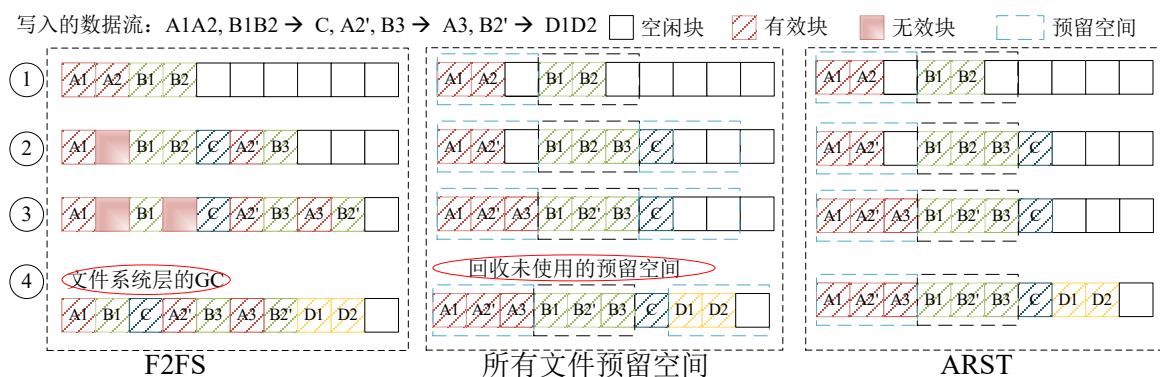


图 2.5 传统 F2FS 和不同预留空间策略的数据写方法示意图

ARST 基于文件读写特征使用机器学习算法选择部分文件预留空间，ARST 的架构设计如图 2.6 所示，左半部分描述了一个分层的安卓 I/O 系统结构，ARST 只在文件系统层做出修改，不需要修改应用层和 SQLite 数据库层，闪存层也无需修改。右半部分展示在文件系统层实现 ARST 的主要工作流程，并用一个地址预分配示例展示了实现预留功能需要修改的部分，其中 0 表示可用地址，-1 表示该地址已经被占用，其他具体数值表示实际的逻辑地址，ARST 的写数据流程基于 IPU 实现空间预留的过程做出相应修改。与图 2.5 中描述的一致，图 2.6 中 ARST 保证分配给预留文件的地址是连续的。

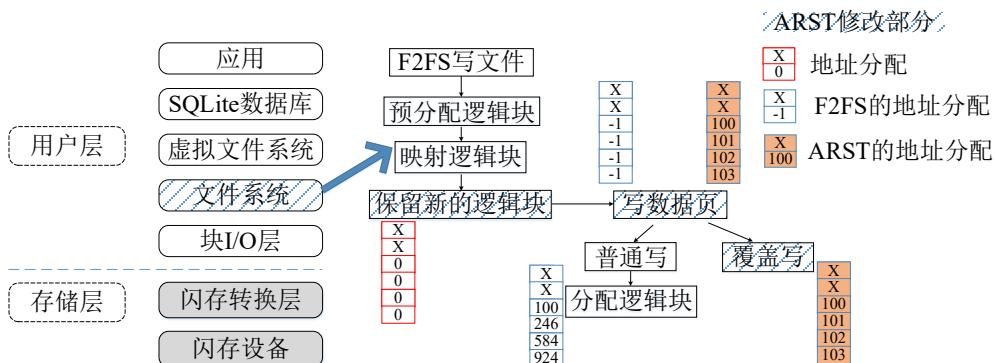


图 2.6 ARST 架构设计与地址预分配示意图

2.2.2 应用读写行为特征分析

多次追加写入和更新的文件，如图 2.5 中的文件 A 和 B，是合适的预留文件，而一次写入的文件，如文件 C，则不是合适的预留文件，因为仅写一次的文件没有产生碎片。收集应用 trace 并分析其行为特征以深入挖掘哪些文件可以作为预留文件，采用 ftrace 收集了使用半小时智能手机的应用 trace，该过程主要模拟用户使用微信、微博及一些系统应用，该 trace 包括 822532 次写操作，148423 次读操作。在该数据集中大多数.db、.db-journal 和.db-wal 文件被写入了很多次，与 SQLite 数据库文件属于最严重的碎片文件结论一致^[55]。但在该数据集中，无法根据文件类型直接判断文件是否要预留，因为观察到有些.png 文件被写入一次，而有些.png 文件被多次写入，尽管与直觉相悖，通常认为.png 文件是仅写一次的。随着用户使用各式各样的应用，产生大量文件数据，手动选择预留文件是不可行的，大规模数据集之间既有相似之处，也存在不同之处，机器学习方法能实现对不同数据集的高效分析并适应动态负载，启发使用一种低成本、高准确率的机器学习算法高效地选择预留文件。

进一步分析不同应用数据集的 I/O 行为特征，Jeong 等人使用 MobiBench 抓取了脸书和推特 trace，并提供了数据集的细节^[24]，使用 MobiBench 抓取微信、百度和王者荣耀 trace 来获得典型应用的数据集。类似于脸书和推特，微信是一个社交应用；使用百度浏览新闻，搜索关键字，点击推荐新闻，通过链接跳转到百度贴吧，返回浏览的历史网页等；进行王者荣耀游戏的日常操作包括启动应用后与弹出窗口交互，阅读游戏地图，开始和退出一场王者荣耀游戏等。收集的应用 trace 已经公开在 github

华中科技大学博士学位论文

网页¹上，各个应用数据集的文件读写特征分析如表 2.3 所示。因为 MobiBench 使用 strace²收集数据，在该 trace 中使用文件描述符描述文件读写特征，可以直接获取打开文件返回的文件描述符，打开文件和写操作的文件描述符，但是不能通过一条读写操作 trace 直接获取写入文件。将文件分为多次写入和一次写入，并分析应用数据集的特征，脸书数据集中完成 14429 次写操作，推特数据集中完成 6029 次写操作，根据写操作的文件描述符统计，脸书和推特数据集前 19.8% 和前 25% 的文件是多次写入的，数据显示这些文件的写入次数超过 100 次。微信、百度和数据集分别完成 22378 次、81011 次和 93109 次写入，根据写入文件统计写入次数超过 100 次的文件，它们各自前 4.5%、0.9% 和 6.9% 的文件是多次写入的，因为微信、百度和王者荣耀数据集比较大，这些数据集中写操作的文件描述符和写入文件之间的映射关系混乱，所以根据写入文件统计文件写入次数。

表 2.3 典型应用数据集的读写特征

应用	时长 (s)	大小 (MB)	读操作数 (千次)	写操作数 (千次)	打开文件 返回的文 件描述符 (个)	打开文 件数量 (个)	写入的 文件描 述符 (个)	追溯写 入的文 件数量 (个)
脸书	117.016	0.896	9.918	14.429	99	11	91	4
推特	117.34	0.495	6.144	6.029	71	7	60	5
微信	230.766	4.869	63.255	22.378	377	1946	205	178
百度	304.656	11.282	221.432	81.011	493	4873	264	1399
王者 荣耀	375.158	20.037	533.395	93.109	525	5857	171	72

通过 strace 收集的数据集采用 fd 描述文件特征是直接和便利的，但不精准，因为 fd 和文件之间是多对多的映射关系，在不同时间打开同一个文件可能返回不同的 fd，多个线程打开不同文件可能返回相同的 fd。脸书和推特 trace 中打开文件返回的文件描述符数量与写操作的文件描述符数量没有太大区别，且脸书和推特 trace 打开

¹ <https://github.com/lihua-yang/Wechat-Baidu-App-and-Game-traces-captured-by-Mobibench>

² <https://github.com/strace/strace>

的文件数量有限，可以找到 fd 和文件之间精准的映射关系，但随着数据量增加，微信、百度和王者荣耀 trace 中 fd 和文件的映射关系比较复杂。在收集的数据集中，除百度以外的其他应用写操作的文件描述符数量比写文件数量多，原因在于百度浏览了很多新闻，读写了很多文件。因为预留的 fd 可能会映射到预留文件和普通文件，根据预留的 fd 查找预留文件时，可能会找到不必预留的普通文件，所以 ARST 使用写入文件而不是 fd 描述文件特征。基于打开文件返回的 fd 收集文件特征并构造数据集的方法称为 ARS，该方法没有追溯（Traceback）写入文件。ARST 先追溯写操作的文件对象，使用追踪到的包含文件路径的文件名描述文件特征，基于追溯到的文件描述文件特征是精准的，能使机器学习方法更加高效。

设计 ARST 有三个关键问题，(1) 选择合适的预留文件，如图 2.5 中 ARST 选择文件 A、B 作为预留文件，(2) 确定合适的预留空间大小，在图 2.5 中 ARST 为文件 A、B 分别预留三个数据块，(3) 在空间不足时 F2FS 性能下降，ARST 需要在合适的时机打开和关闭预留功能以避免浪费空间。有一些文件的历史信息当前比较少，但这些文件可能继续追加写，并在未来多次更新与读取，准确识别上述文件，并为其预留空间，可以最大化发挥预留空间的功能，减少文件碎片。

2.2.3 自适应选择预留文件

文件系统中没有机器学习算法能直接使用的数据集，过滤文件特征构建机器学习数据集具有挑战，原因在于文件系统中有大量特征可以用于对文件进行分类，如文件类型、大小、访问时间等，需要在其中选择与产生文件碎片强相关的特征，由于一些文件的创建时间不长，与时间有关的历史信息有限，选择了部分与时间无关的特征，共选择了以下 6 个文件特征。

- 1) 文件类型：大多数数据库文件碎片化程度严重，碎片产生与文件类型有关。
- 2) 文件大小：频繁追加写入的文件大小持续增长，产生大量碎片，当存在严重的空闲空间碎片时，较大的文件更有可能碎片化写入。
- 3) 文件写时间间隔：文件写时间间隔用数据集中第一次写文件和最后一次写该文件之间的时间间隔表示。通常文件写时间间隔越大，该文件数据越有可能被其他文件写入的数据分割。

- 4) 文件写请求次数: 随着写请求次数增加, 文件会变得碎片化。
- 5) 文件写频率: 文件写频率用写请求次数与文件写时间间隔的商表示。一个文件被写入的频率越高, 它越有可能产生碎片。
- 6) 写并发程度: 用于衡量写请求的并发程度, 定义为发出写请求时(非常短的时间内)并发写文件的数量, 统计写该文件时上下0.1秒内的写文件数量。若当时只写入该文件一个文件, 则写并发程度为0, 写并发程度越大, 文件写请求之间的并发冲突越大, 产生的文件碎片越多。

单独使用一种文件特征构造数据集不精准, 如一个大文件可能只写一次, 文件大小、文件写时间间隔和文件写请求次数是与碎片产生强相关的特征, 需要进一步评估文件类型、写频率和写并发程度三个属性作为文件特征的必要性, 而不是简单地在构造数据集时增加某个属性。比较了四个数据集的分类准确率, 分别是只删除文件类型、只删除文件写频率、只删除写并发程度和包含所有属性的数据集, 分析发现包含所有属性的数据集分类准确率最佳, 因此使用所有文件特征构造数据集。

在追溯到写入文件之后, 统计该数据集中写请求大小之和以得到文件大小, 分别计算文件的写时间间隔和写请求次数, 并根据定义获取写频率和写并发程度, 一个文件可能是多次写入的, 所以写并发程度取该文件每个写请求并发程度的平均值。对数据特征进行预处理, 将文件类型特征转换为离散数值, 设计了一个映射字典将其转换为数值, 先处理系统文件夹和应用文件夹, 再分别映射文件名后缀, 其他未包含在上述类别中的文件类型作为最后一个分类, 其他特征是无需处理的数值。在分析文件碎片和I/O行为后, 用预留标志标记文件, 并随机取75%的数据作为训练集。

基于历史信息决定某个文件是否是预留文件是一个二分类问题, 最常用的分类算法有逻辑回归¹(Logistic Regression, LR)、人工神经网络^[88](Artificial Neural Network, ANN)、K最近邻算法²(K-Near Neighbor, KNN)、支持向量机³(Support

¹ https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

² <https://scikit-learn.org/stable/modules/neighbors.html#classification>

³ <https://scikit-learn.org/stable/modules/svm.html#svm-classification>

华中科技大学博士学位论文

Vector Machines, SVM)、决策树算法¹ (Decision Tree, DT)、AdaBoost²算法和随机森林算法³ (Random Forest, RF)。决策树算法采用分类与回归树 (Classification And Regression Tree, CART)，重复机器学习过程 15 次，取平均值作为最终结果。用真正例 (True Positives, TP)，真负例 (True Negatives, TN)，假正例 (False Positives, FP) 和假负例 (False Negatives, FN) 计算机器学习算法中的精确率 (Precision)、召回率 (Recall) 和准确率 (Accuracy)，用式 (2.1) 计算精确率，用式 (2.2) 计算召回率，用式 (2.3) 计算准确率。

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (2.1)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (2.2)$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (2.3)$$

追溯到写入文件后，脸书和推特分别写了 4 个和 5 个文件，是小型数据集，可以在获取文件特征后直接分析决定是否预留，不需要机器学习过程。微信、百度和王者荣耀数据集是大型数据集，对这三个数据集比较了七种通用分类算法的分类效果，表 2.4 展示了不同应用场景下不同分类算法的分类效果，可以看到 KNN、SVM、决策树、AdaBoost 和随机森林算法取得了良好的分类准确率。

需要从高准确率的分类算法中选择计算成本最低的一种，图 2.7 展示了上述取得良好分类准确率的算法计算时间开销，可以看到决策树算法的计算时间最短。虽然 AdaBoost 和随机森林的三个应用平均准确率分别比决策树算法略高 0.05% 和 2.26%，但 AdaBoost 和随机森林的计算开销比较大，分别是决策树算法的 4.47 倍和 3.98 倍，原因在于 AdaBoost 训练了 10 个分类器，随机森林包含 10 棵决策树。决策树的训练可以在 10 毫秒内完成，分析和预测开销是可以接受的，决策树算法易于理解，计算成本低，分类精度高，Wang 等人也使用决策树算法管理腾讯图片的缓存^[89]，因此对不同数据集使用决策树算法来获得预留文件可行。ARST 离线更新分类器获得预留文

¹ <https://scikit-learn.org/stable/modules/tree.html>

² <https://scikit-learn.org/stable/modules/ensemble.html#adaboost>

³ <https://scikit-learn.org/stable/modules/ensemble.html#forest>

华中科技大学博士学位论文

表 2.4 不同应用场景下常用的机器学习分类算法精确率、召回率和准确率性能

应用	算法	精确率	召回率	准确率
微信	逻辑回归	0.8633	0.750667	0.751111
	人工神经网络	0.858	0.529333	0.530370
	K 最近邻	0.948	0.892	0.891852
	支持向量机	0.7613	0.873333	0.872593
	决策树	0.9133	0.905333	0.905185
	AdaBoost	0.9413	0.932667	0.933333
	随机森林	0.9607	0.943333	0.942222
百度	逻辑回归	0.9727	0.962667	0.962667
	人工神经网络	0.9633	0.822	0.821333
	K 最近邻	0.9847	0.974667	0.976
	支持向量机	0.94	0.968667	0.969524
	决策树	0.99	0.988667	0.988571
	AdaBoost	0.9873	0.987333	0.987619
	随机森林	0.9887	0.986	0.985037
王者荣耀	逻辑回归	0.5673	0.374	0.374074
	人工神经网络	0.6607	0.392667	0.392593
	K 最近邻	0.906	0.778667	0.777778
	支持向量机	0.6913	0.828	0.828148
	决策树	0.8647	0.843333	0.844444
	AdaBoost	0.8373	0.817333	0.818519
	随机森林	0.9133	0.872	0.874074
平均	逻辑回归	0.8011	0.695778	0.695951
	人工神经网络	0.8273	0.581333	0.581432
	K 最近邻	0.9462	0.881778	0.881877
	支持向量机	0.7976	0.89	0.890088
	决策树	0.9227	0.912444	0.912734
	AdaBoost	0.922	0.912444	0.913157
	随机森林	0.9542	0.933778	0.933778

件，考虑到应用更新频率和用户行为特征，在用户很少使用智能手机的时间训练分类器，如每隔 10 天的凌晨四点。由于决策树算法能低成本高准确率地选择预留文件，还发现有些预留文件在更新前后都是预留文件，更新预留文件的开销是可以接受的。

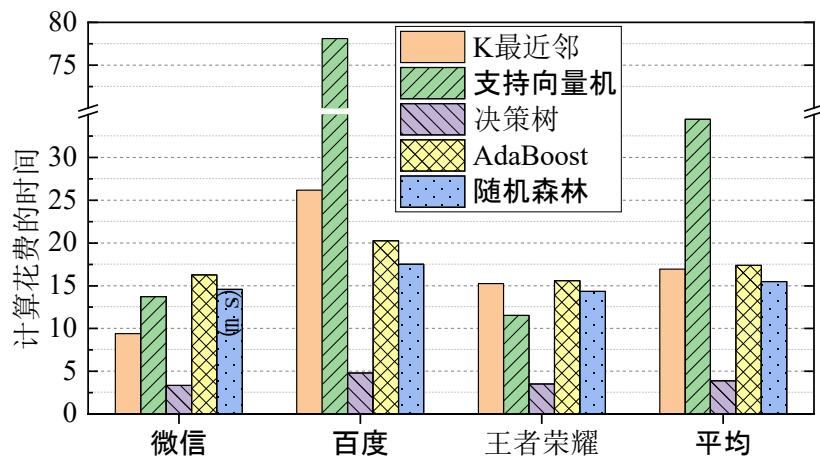


图 2.7 不同应用场景下高准确率分类算法的计算开销

很难决定一个历史信息很少的文件是否需要被预留，原因在于文件的相关信息量过少以致无法准确判断，但找到潜在可能成为预留文件的文件是重要的，可以最大化发挥预留空间的价值。文件类型、写频率和写并发程度是三个时间依赖性小的文件特征，可以辅助准确地分类历史信息少的文件。截断数据集为更短时间的应用数据集，如 304656 毫秒的百度数据集截断前 200000 毫秒的数据集，预处理文件特征，构建一个新的数据集，在文件历史信息少的场景下，观察一个在完整数据集中被分类为预留文件的文件在该截断数据集中是否也被分类为预留文件。如果在历史信息有限的场景下，该文件也被分类为预留文件，则 ARST 的机器学习模型能准确分类历史信息少的文件。截断了来自微信、百度和王者荣耀数据集的三个更短运行时间的数据集，预测 30 个在完整数据集中被分类为预留文件的文件在历史信息少时是否也被分类为预留文件。实验结果发现 AdaBoost 和随机森林算法能准确分类，文件可以准确地被提前预测为预留文件，而 KNN、SVM 和决策树算法的分类效果较差。因此当需要挖掘具有很少历史信息的文件作为预留对象，并可以接受计算时间开销时，使用 AdaBoost 或随机森林算法，对于其他一般场景的大型数据集，使用决策树算法选择预留文件。

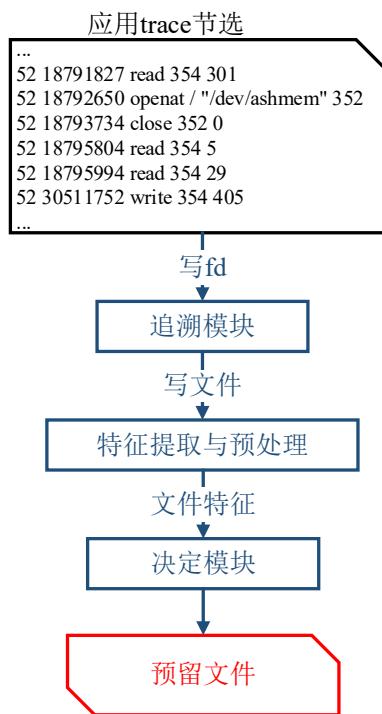


图 2.8 ARST 选择预留文件模块示意图

图 2.8 展示了 ARST 选择预留文件的模块示意图，当需要决定历史信息少的文件是否预留时，ARST 决定模块使用 AdaBoost 或随机森林算法，对于其他大规模的数据集，决定模块使用决策树算法。对于一般的小型数据集，ARST 在追溯得到少量的写入文件后，根据这些文件预处理的读写特征选择预留文件，不必使用机器学习算法，如表 2.3 所示，在追溯写入文件后，提供的脸书和推特数据集分别获得 4 个和 5 个写入文件，经过特征特取与预处理后，对数量较少的这些文件可以基于文件特征直接决定是否预留，无需使用机器学习算法。当系统刚刚初始化使用或提供的数据集较少时，考虑到文件数量比较少，决定模块无需使用机器学习算法。对于无需挖掘历史信息少的潜在预留文件的大型数据集，基于收集的不同应用 trace，同一类应用的文件特征相似，可以使用同一个决策树模型，如同属社交应用的脸书和微信，对于不同类型的的应用，如社交应用、地图应用和视频应用，考虑到它们的文件特征不同，为了保证分类准确率，训练得到的决策树分叉判断条件可能稍微有区别。决策树算法的计算开销比较小，更新模型的时间开销可以接受。当文件系统多线程打开多个文件时，可能存在多个文件返回相同 fd 的情况，ARS 根据预留 fd 映射预留文件时，可能选择

到一个不需要预留的普通文件，如提供的脸书数据集中，ARS 根据预留 fd 构造数据集时选择了 7 个预留文件，而 ARST 根据追溯的写入文件构造数据集时只选择了 2 个预留文件，相比于 ARST，ARS 中有 5 个普通文件被预留空间。ARST 追溯到写 fd 的文件后基于写入文件处理文件特征，虽然在构造数据集时追溯写 fd 花费时间，但通过追溯获得的预留文件是精准的。

2.3 空间预留策略

2.3.1 合适的预留空间大小

寻找合适的预留空间大小也是一个挑战，如果预留空间太大，会浪费文件系统存储空间，如果预留空间太小，就不能存放所有的文件数据，仍然存在文件碎片。考虑到 F2FS 以 2 MB 大小的段为单位执行 GC，将预留空间大小与 GC 单位对齐使得预留文件所处的段比较不可能被选择为清理对象。假设预留空间大小为 128 KB，其所处段被选作清理对象的概率大于预留空间大小为 2 MB 的段，原因在于预留空间大小为 2 MB 时整个段被标记为有效块，迁移开销很大导致不太可能被文件系统选作清理对象。此外预留空间中的数据属于同一个文件，可能同时失效，会减少 GC 开销，尽量减小 GC 开销也是 ARST 的目标，所以决定初始预留空间大小为 2 MB，当一个预留文件的大小超过 2 MB 时，则预分配另一个段。

2.3.2 预留功能的开启与关闭管理

当文件系统空闲空间不足时，停止预留策略，以避免重复预留空间和回收空间并触发 GC。根据文件系统空间使用率，ARST 动态地预留文件，其关键是选择一个文件系统空间使用率阈值来控制预留策略。一方面，F2FS_IPU_UTIL 就地更新策略默认的文件系统空间使用率阈值是 70%。另一方面，执行一系列微基准测试进行预留策略阈值的敏感性分析。在微基准测试中，缓慢地填满文件系统，并比较 GC 次数来确定预留策略的阈值。在测试预留策略阈值时，在四种不同的文件系统填充程度下运行 I/O 密集型工作负载，系统低空间使用率（2%）指系统刚刚初始化，经过微基准测试负载大量创建文件和删除部分文件操作后分别到达中空间使用率（64%）、高空

间使用率（84%）和满空间使用率（96%），分别在这四种系统状态插入执行一个 I/O 密集型负载，由于预留策略阈值不同，I/O 密集型工作负载的文件在四种文件系统填充程度的场景下是否预留不同，文件系统在运行完微基准测试负载和 I/O 密集型工作负载后是几乎填满（98%）的。因为空闲空间不足时会触发更多的 GC，所以选择整个过程的 GC 次数作为衡量标准，并重复试验三次，取平均值作为结果。

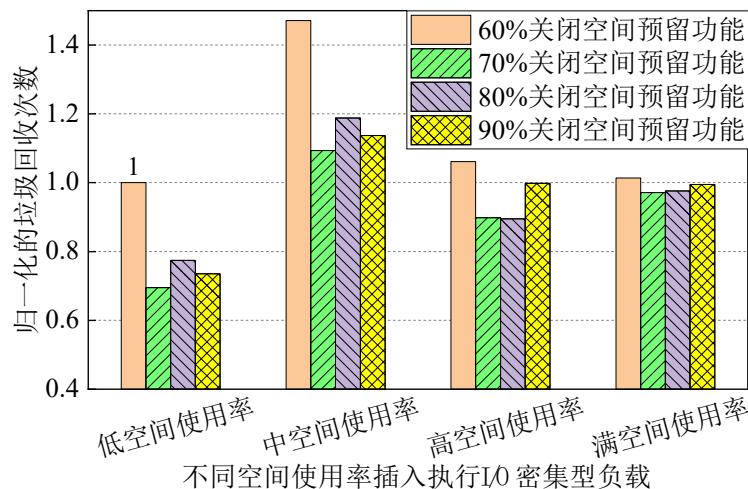


图 2.9 60%、70%、80% 和 90% 空间使用率关闭预留功能的垃圾回收次数，以 60% 空间使用率关闭预留功能的策略在低空间使用率系统状态的垃圾回收次数为基准

图 2.9 展示了将 60%、70%、80% 和 90% 文件系统空间使用率作为打开和关闭预留功能的阈值时，文件系统从启动开始到完成所有 I/O 任务整个过程的 GC 次数。从图 2.9 中看出，70% 作为预留策略阈值时，触发的 GC 次数最少，60% 阈值会触发最多次数的 GC，因为当文件系统空间使用率超过 60% 时，不再为预留文件预分配空间，导致文件碎片和空闲空间碎片数量增加，当预留阈值为 90% 时，由于空闲空间不足，GC 次数增加。观察到 GC 次数与文件系统的写满程度没有直接关系，而是与文件系统完成任务的使用情况有关，当文件系统空间使用率从低增加到中时，系统空间使用率的增加幅度（62%）是最大的，此时系统执行的任务数量最多，也是最有可能与插入执行的 I/O 密集型负载产生 I/O 冲突的，因此中空间使用率时插入 I/O 密集型负载在四种场景中 GC 次数最多，与已有工作^[64]的结论一致。

最后当文件系统空间使用率低于 70% 时，为所有预留文件预分配空间；当文件系统空间使用率在 70% 到 80% 之间时，已经预留空间的预留文件就地更新，但停止

华中科技大学博士学位论文

为新写入预留文件预分配空间；当空间使用率大于 80%时，停止预留功能并回收未使用的预留空间。通过修改预留标志并释放预留空间，可以容易地回收未使用的预留空间。为了防止文件系统性能发生颠簸，使用阶梯式的阈值，当预留功能根据文件系统空间使用率由关闭变为可用时，ARST 会在打开文件时进行调整。

2.4 性能评估与分析

基于传统 F2FS 使用 657 行 C 语言代码实现了 ARST 策略的原型，其他未预留的常规文件读写流程保持不变，在 f2fs_inode_info（/fs/f2fs/f2fs.h）结构和 f2fs_inode（/include/linux/f2fs_fs.h）结构中增加文件预分配标志、预分配工作时的文件开始偏移量、以及文件大小。通过对应用数据集的分析发现，一些文件是适合预留的，在 ARST 策略中，一些预留文件在 I/O 模式变化后仍然是预留文件。

2.4.1 实验环境

基于 Hikey960 开发板与华为 Mate10 Pro 智能手机实现 ARS 和 ARST 策略，Hikey960 和华为 Mate10 Pro 的软硬件参数如表 2.5 所示，基于 AOSP 软件架构的 Hikey960 适用于基于安卓系统的个人电脑、机器人、无人机和其他智能设备的研发。ARST 策略主要修改 Linux 内核中 F2FS 工作流程有关部分，实现源码由于落实到项目中暂时无法开源。

表 2.5 实验平台的参数

实验设备	Hikey960	Huawei Mate10 Pro
SoC	Kirin 960	HUAWEI Kirin 970
CPU	4 Cortex A73 + 4 Cortex A53 Big.Little CPU architecture	4 Cortex A73 2.36 GHz + 4 Cortex A53 1.8 GHz
RAM	3 GB LPDDR4 SDRAM	6 GB RAM
存储	32 GB UFS Flash Storage	128 GB ROM
AOSP	9.0	9.0
Linux 内核	4.9.76	4.9.111

为了准确理解 ARST 的性能收益，在 Hikey960 和华为 Mate10 Pro 上进行了全面的实验。实验结果的展示中分别使用 F2FS 表示传统 F2FS 性能，IPU 表示使用

F2FS_IPU_FSYNC 就地更新策略的 F2FS 性能，当同步的数据大小小于一个特定阈值时，该数据将就地更新，ARS 表示基于 fd 构建的数据集使用决策树算法选择预留文件的 F2FS 性能，ARS+IPU 表示 ARS 和 IPU 策略的集合，ARST 表示使用了 ARST 策略的 F2FS 性能，ARST+IPU 表示 ARST 策略与 IPU 共同工作的性能。测试中的实验重复多次，如 10 次，取平均值作为实验结果，遵循 Ji 等人提出的四种系统空间使用率状态用于展示实验结果，分别是低（初始化状态）、中（空间使用率小于 80%）、高（在 80% 到 95% 之间）和满（高于 95%）^[56]，本章实验中具体的空间使用率分别是低 2%、中 64%、高 84% 和满 96%。

从四个方面评估 ARST 性能，(1) 使用微基准负载的实验结果证明 ARST 策略有效减少文件碎片和空闲空间碎片，衡量 I/O 性能和垃圾回收开销，(2) 测量 SQLite 数据库的插入、更新、删除性能，(3) 重播了 MobiBench 在实际使用场景下收集五个典型应用的系统调用 trace，测试运行时间，(4) 评估了 ARST 的空间和时间开销。

2.4.2 文件碎片对顺序读的影响

测试顺序读连续文件与碎片文件的性能以衡量 ARST 减少文件碎片的效果，设置与图 2.1 中一样的归一化标准，用 ARST 策略减少碎片后，ARST 和 ARST+IPU 中单个文件的顺序读性能如图 2.10 所示，此时文件系统空间使用率是 2%。

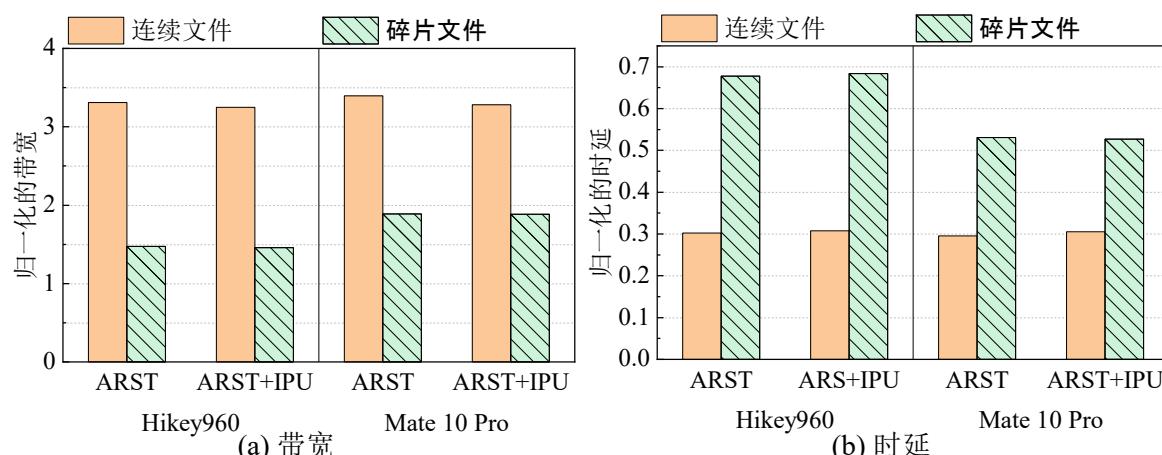


图 2.10 减少文件碎片后的文件顺序读性能，采用图 2.1 中的基准，以 Hikey960 与华为 Mate 10 Pro 上传统 F2FS 顺序读碎片文件的带宽和时延为基准

ARST 的结果与 ARST+IPU 类似，说明 ARST 兼容 F2FS 现有就地更新策略。与

华中科技大学博士学位论文

Hikey960 和 Mate10 Pro 上传统 F2FS 的碎片文件相比，ARST 碎片文件的带宽分别提高了 46.93% 和 88.97%，在 Hikey960 和 Mate10 Pro 上，ARST 碎片文件的顺序读时延也分别缩短了 32.27% 和 47.5%，与图 2.1 中的顺序读性能相比，ARST 减小了连续文件和碎片文件之间的性能差距。由于硬件的不同，ARST 在 Hikey960 和 Mate10 Pro 上的性能提升幅度不同，但性能提升趋势相同，因此在后续分析中不展示 Mate10 Pro 的性能。*fio* 构造的微基准负载和 SQLite 负载中 ARS 和 ARST 的预留文件一致，在这两个负载的实验分析中只展示 ARST 的结果，而重播应用 trace 测试执行时间时，ARS 和 ARST 的预留文件不一致，则分别展示 ARS 和 ARST 的应用执行时间。

通过 FS_IOC_FIEMAP 计算文件区段数量来衡量文件碎片数量，区段 (extent) 是一段逻辑地址连续的数据，用于表示一个文件碎片，区段数量越多，文件数据分布越离散。实验结果表明，低文件系统空间使用率时，传统 F2FS 中碎片文件有 32768 个区段，而 ARST 中只有 1 个区段，高文件系统空间使用率时，传统 F2FS 和 IPU 中碎片文件的区段数为 32768，ARST 和 ARST+IPU 的文件区段数分别为 3 和 5。虽然生成碎片文件的过程是相同的，但传统 F2FS 逐个写入文件数据，它们是交叉写入且异地更新的，而 ARST 则在预留的空间内写入数据。当写入的同步数据满足一定大时，IPU 才会执行，IPU 减少碎片的效果有限，移动存储系统中存在大量频繁的随机同步写，ARST 改变了文件布局，而 IPU 几乎没有改变。此外 Hikey960 上连续文件的顺序读带宽为 392 MB/s，而 ARST 中优化后的碎片文件的顺序读带宽为 177 MB/s，在华为 Mate10 Pro 上连续文件和优化后的碎片文件的顺序读带宽分别为 489 MB/s 和 274 MB/s。排除了其他原因，发现性能下降是因为闪存器内部存在物理碎片，导致访问闪存内文件时没有充分发挥闪存的 I/O 并行性。虽然 F2FS 和闪存都是异地更新的，但是 F2FS 的逻辑地址因为 FTL 的存在没有与闪存的物理地址对齐，FTL 加剧了碎片的负面影响，本章不涉及移动设备中闪存 FTL 的讨论。与传统 F2FS 相比，ARST 明显减少了文件碎片，受益于碎片文件的碎片数量明显减少，ARST 的 I/O 开销比传统 F2FS 要小。除了减少文件碎片和提高顺序读性能以外，还需衡量 ARST 是否提高了其他 I/O 性能和减少了 GC 开销。在大量的创建、更新、删除和截断操作后，在文件系统空间使用率高时，存在严重的空闲空间碎片，而大型的非 SQLite 文件也

存在文件碎片，空间使用率过高会使得所有类型文件产生碎片^[56]，所以还探讨 ARST 在高空间使用率时是否能有效减少碎片。设计一个碎片化负载，(1)为了以合适大小的文件填满文件系统，控制文件系统分区大小和初始填充文件大小之间的合理比例，将初始文件大小设置为 $2 \times 10^{-18} \times$ 分区大小并向上取整；(2)不断写该大小的文件到文件系统空间使用率超过 90%；(3)间隔删除文件；(4)设置文件大小为上一次填充文件大小的一半。重复步骤(2)至(4)，直到创建删除的文件大小为 4 KB，使用该碎片化负载能使文件系统快速充满文件碎片。在智能手机的实际使用中，存在频繁更新应用、增加文本信息以及删除过时的图片和视频等，使用该碎片化负载模拟快速老化。

2.4.3 空闲空间碎片数量

在高文件系统空间使用率采用不同大小的碎片数量衡量空闲空间碎片化程度，空闲空间碎片大小的分级如表 2.1 所示，一个连续的空闲空间片段记为一个空闲空间碎片，测试了每个策略中不同大小级别的空闲空间碎片数量，图 2.11 展示了不同大小空闲空间碎片的数量。在所有策略中，第一级空闲空间碎片数量所占的比例最大，从 63% 到 77% 不等，在所有策略中，ARST 产生的第一级和空闲空间碎片总数量最少，ARST 的空闲空间碎片总数量比传统 F2FS 减少 19.57%。

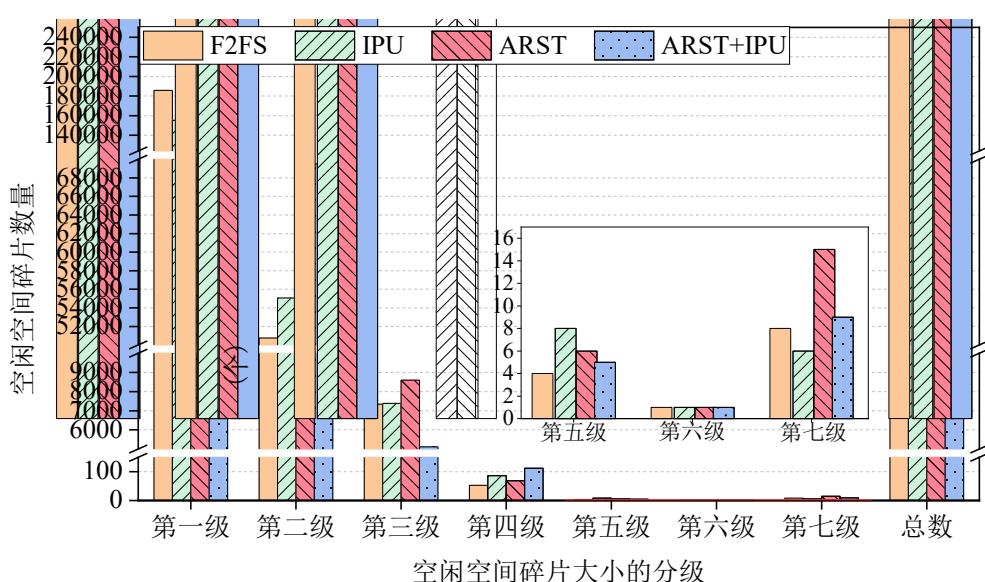


图 2.11 不同大小的空闲空间碎片数量

在文件系统空间使用率较高时，空闲空间被分割成大量的小块。从图 2.11 中可以看出，ARST 减少了单个的 4 KB 大小的碎片数量，增加了连续的大空闲空间碎片数量。ARST 中连续的空闲空间碎片不仅数量更多，而且更大，可以满足将来的大文件写请求而不会产生文件碎片，ARST 有效地减少了空闲空间碎片。

2.4.4 文件系统读写性能

与传统 F2FS 相比，ARST 顺序读写和随机读写性能如图 2.12 所示。通过碎片化负载对文件系统进行快速老化后，文件系统的空间使用率达到 84%，处于高文件系统空间使用率。在运行碎片化负载时创建预留文件，以确保在空间使用率达到 70% 之前确定文件布局，因此是在文件系统空间使用率小于 10% 时创建文件，在高文件系统空间使用率时测试文件 I/O 性能。对于顺序读、顺序写、随机读和随机写，在低文件系统空间使用率时，ARST 比传统 F2FS 分别提升 56.55%、169.76%、27.13% 和 34.28%，在高文件系统空间使用率时分别提高 47.65%、179.62%、20.8% 和 58.12%。在传统 F2FS 和 IPU 中，碎片文件的碎片化程度严重，在 ARST 中，无论是低文件系统空间使用率还是高文件系统空间使用率，文件碎片都明显减少了，因此 ARST 的 I/O 开销比传统 F2FS 和 IPU 小。文件碎片会降低 I/O 性能，主要原因在于碎片导致更多的 I/O 请求，影响 I/O 调度的合并操作，造成更多的 I/O 堆栈开销。此外安卓应用高度同步、多线程的写行为，导致文件系统碎片化严重，ARST 有效减少文件碎片，带来显著性能提升。

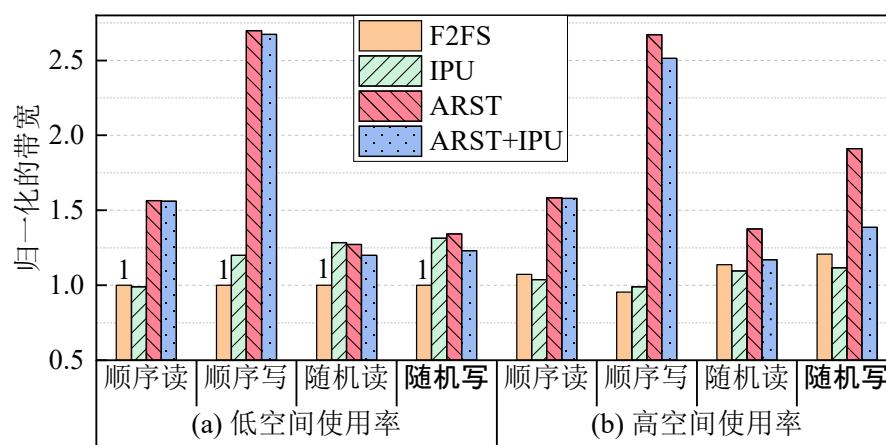


图 2.12 文件的顺序读写和随机读写性能，以低空间使用率下 F2FS 读写性能为基准

2.4.5 垃圾回收开销

F2FS 的垃圾回收对性能有负面影响，空闲空间碎片会放大 GC 的负面影响，因为更多的空闲空间碎片会增加 GC 次数。通过 *cat stat* 监控运行碎片化负载时的 GC 开销，归一化的垃圾回收开销如图 2.13 所示，衡量 GC 次数、迁移的有效块数量、读操作次数和写操作次数，ARST 比传统 F2FS 分别减少了 23.15%、7.76%、78% 和 38.36%，与 IPU 相比，ARST 分别减少了 13.29%、0.13%、67.43% 和 40.02%。根据减少的 GC 次数与每次 GC 的能耗，ARST 比传统 F2FS 减少消耗 13.4~53.64 焦能量，比 IPU 减少消耗 6.83~27.35 焦能量，能耗减少的幅度与减少的前台 GC 和后台 GC 次数的分布有关。文件碎片和空闲空间碎片互相影响，ARST 有效地减少了文件碎片，如图 2.11 所示减少了空闲空间碎片，从图 2.12 和图 2.13 看到 ARST 提升了 I/O 性能，减少了 GC 开销。

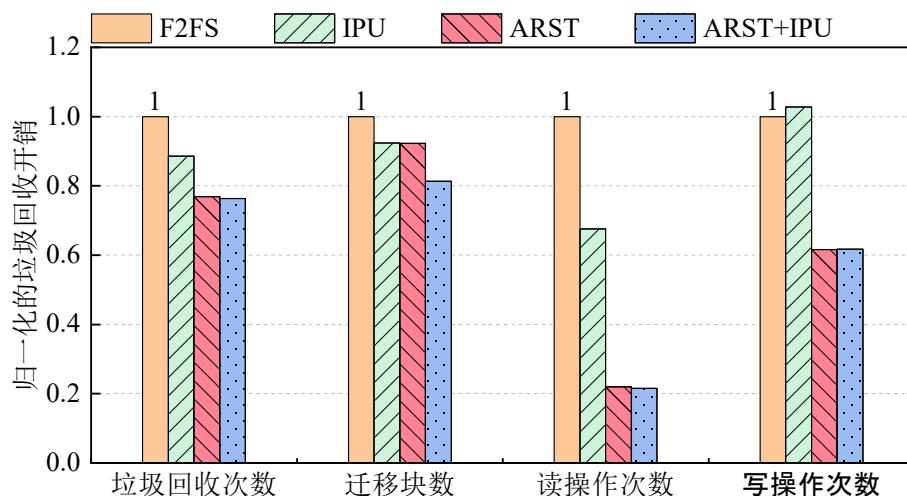


图 2.13 垃圾回收开销

2.4.6 数据库应用场景性能

SQLite 被广泛用于管理安卓应用的持久化数据，使用 MobiBench 对 SQLite 数据库执行多线程并发地操作，在默认的 TRUNCATE 日志模式下，测试 1000 条记录和 2 个线程的三种类型事务（插入、更新、删除）。在中文件系统空间使用率场景，碎片化负载将文件系统填满至 70%，使得预留文件能够就地更新，使用 ftrace 收集数据 trace，构造可用于机器学习建模使用的数据集，使用决策树算法确定预留文件。SQLite

的插入、更新和删除操作性能如图 2.14 所示，用每秒完成的事务数（Transactions Per Second, TPS）衡量，基于低文件系统空间使用率场景的传统 F2FS 性能归一化。对于插入、更新和删除，低文件系统空间使用率时 ARST 每秒完成事务数分别是传统 F2FS 的 1.24 倍、1.26 倍和 1.26 倍，中文件系统空间使用率时 ARST 分别是传统 F2FS 的 1.42 倍、1.44 倍和 1.44 倍。

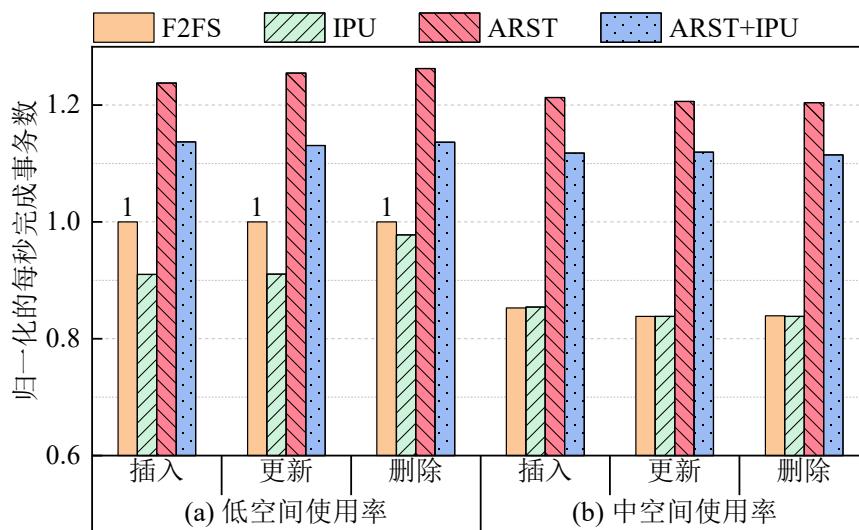


图 2.14 SQLite 应用场景的插入、更新和删除操作性能，以低空间使用率 F2FS 的性能为基准

文件碎片非常影响 SQLite 的操作速度，原因在于 SQLite 数据库中包含大量的同步写操作，SQLite 性能与同步写操作的性能正相关，因为 ARST 有效减少了文件碎片，连续的写入操作提高了同步写性能。与直觉相反，IPU 的性能比传统 F2FS 略差，ARST+IPU 的性能也比 ARST 略差，原因在于 IPU 策略无法优化同步写操作。

2.4.7 应用执行时间

对五个应用 trace 重复 SQLite 应用场景的机器学习建模过程并获得预留文件，脸书和推特 trace 由 Jeong 等人^[24]直接提供，此外使用 MobiBench 抓取了微信、百度和王者荣耀 trace，在六种策略中重放五个应用的 trace 并比较运行时间。图 2.15 展示了不同空间使用率重播各个应用 trace 的执行时间，与 IPU 策略相比，在低文件系统空间使用率、中文件系统空间使用率，ARST 的应用执行时间分别减少了 50.76%、46.9%（脸书），43.2%、33.78%（推特），93.13%、92.19%（微信），91.34%、91%（百度）和 94.28%、93.91%（王者荣耀）。在低空间使用率、中空间使用率，相比于 ARS，

ARST 减少运行时间的幅度分别是 14.86%、15.52%（脸书），17.28%、7.89%（推特），49.06%、39.34%（微信），55.67%、50.9%（百度），81.48%、78.4%（王者荣耀）。

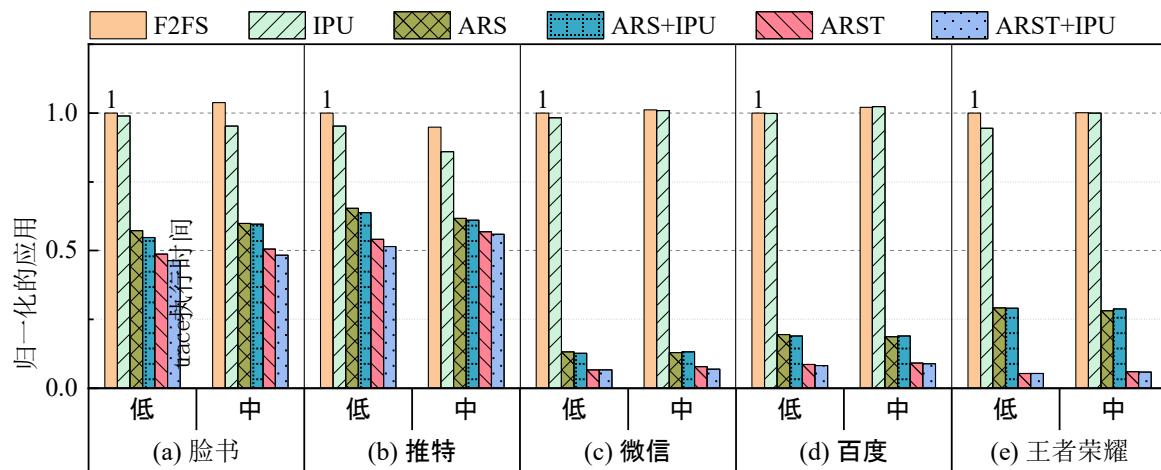


图 2.15 重播应用 trace 的执行时间，低表示低空间使用率，中表示中空间使用率，以低空间使用率 F2FS 花费的时间为基准

因为 ARST 改变了文件数据组织布局，预留文件的数据聚拢在同一块区域，所以相比于 F2FS 和 IPU，ARST 明显缩短了应用的执行时间。数据量越大，ARST 对 trace 运行时间的减少幅度就越大，因为 ARST 能更多地减少文件碎片。由于 ARS 中普通文件可能被选为预留文件，相比于 ARST，ARS 分配与回收预留空间的次数更多，有可能浪费空间。相比于脸书和推特数据 trace，微信、百度、王者荣耀数据 trace 的文件数量比较多，fd 与文件的映射更加混乱，ARS 根据 fd 得到的预留文件不精准，ARST 获得的预留文件更加精准，提升性能的幅度更大，综上所述在所有的应用数据 trace 上，ARST 相比 ARS 有更大的性能提升。

2.4.8 空间和时间开销

除了衡量性能提升以外，还评估了 ARST 策略的空间和时间开销。由于预留文件的大小一般不断增长，几乎没有预留空间浪费，在不理想的状态下，最后一个预留段可能会浪费空间，但是 ARST 会及时回收未使用的空间。预留对象只是一部分文件，预留文件的数据是就地更新的，ARST 仿照 IPU 的工作流程实现预留文件的就地更新，在管理索引节点时有一个小变化，就是在创建预留文件时提前分配地址。

ARST 适配传统 F2FS 的前滚恢复和检查点工作机制，设置直接同步 I/O 的文件能正

常工作说明 ARST 保证文件系统的一致性。

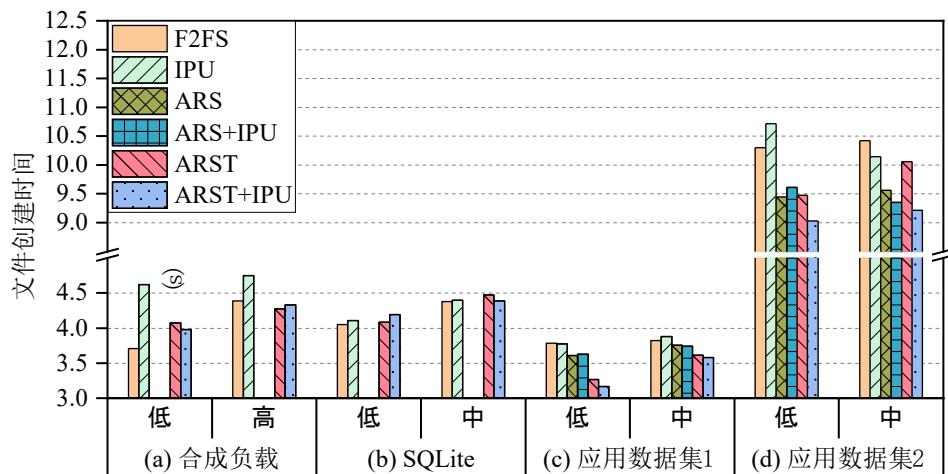


图 2.16 创建文件的时间开销，低/中/高分别表示系统低/中/高空间使用率

ARST 在创建文件时决定该文件是否为预留文件，根据每个数据集中普通文件与预留文件的写入比例创建不同数量的两种文件，比较了不同策略的文件平均创建时间。脸书和推特是 MobiBench 提供的数据集，微信、百度和王者荣耀是在华为 Mate10 Pro 上收集的数据集，因此将应用负载合集分为两种，脸书和推特为应用数据集 1，微信、百度和王者荣耀为应用数据集 2。在合成负载 (fio)、SQLite 应用场景、应用数据集 1 和 2 中，普通文件与预留文件的比例分别为 10、10、8 和 24，即在应用数据集 2 中创建 24 个普通文件和 1 个预留文件，共创建 25 个文件，其他负载中依次类推。创建文件的时间开销如图 2.16 所示，由于 ARS 和 ARST 的预留文件在 fio 构建的合成负载和 SQLite 负载中是相同的，两种策略的时间开销相同，对于这两种负载，没有在图 2.16 和图 2.17 中展示 ARS 和 ARS+IPU 的时间开销。与传统 F2FS 相比，ARST 需要在创建文件时判断该文件是否是预留文件，在合成工作负载和 SQLite 负载中分别多花费 3.6% 和 1.5% 的文件创建时间。在应用数据集 1 和 2 中 ARST 分别比传统 F2FS 少花费 9.6% 和 5.7% 的创建时间，主要原因在于减少文件碎片而缩短的运行时间大于文件创建时间的增量。

尽管应用数据集 1 和 2 中预留文件的数量不同，在创建文件时需要比较的预留文件名称数量不同，但预留文件的创建时间都减少了，说明增加预留文件的数量不会导致文件创建时间的大幅增加。与 ARS 相比，由于 ARST 的预留文件数量更少，

ARST 的文件创建时间也更少，中文件系统空间使用率时，在应用数据集 2 内，ARST 文件创建时间稍高，这与文件系统中的空闲空间碎片数量和使用情况有关，因为此时文件系统比较满，可能触发 GC 获得空闲空间，还面临创建 25 个新文件的写压力。

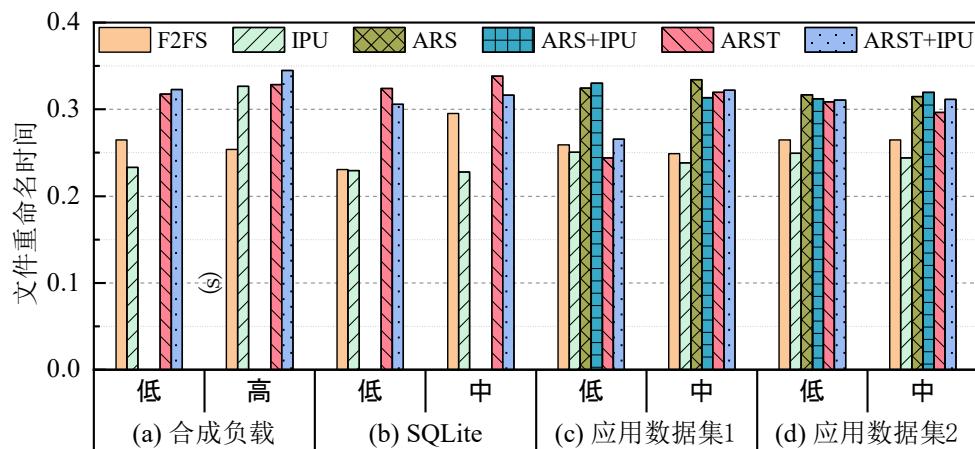


图 2.17 重命名文件的时间开销，低/中/高分别表示系统低/中/高空间使用率

将预留文件重命名为普通文件是对预留功能的主动关闭，通过实验测试比较了不同策略重命名文件花费的时间。将普通文件重命名为预留文件的流程与创建预留文件相同，不重复进行测试。图 2.17 展示了将预留文件重命名为普通文件花费的时间，与传统 F2FS 相比，ARST 仅稍微增加花费的时间，创建一个文件需要与预留文件列表中的文件名称进行比较，判断是否是预留文件，而重命名一个预留文件只需修改预留标志，ARST 的修改没有引入其他计算开销和冗余操作。ARST 文件创建时间相比于智能手机使用时长是微乎其微的，ARST 的时间开销可以接受，而文件碎片导致增加的运行时间比较大，用户能够感受到反应延迟。

综上所述，无论文件系统空间使用率是低、中还是高，ARST 都能有效减少文件碎片和空闲空间碎片，提升 F2FS 性能。ARST 策略能够容易地移植到其他日志结构文件系统，相比于其他策略，ARST 不用复制数据，能减少 GC 并有选择地预留空间。

2.5 本章小结

移动设备上传统 F2FS 和使用 F2FS_IPU_FSYNC 就地更新策略的 F2FS 都会产生大量文件碎片，碎片分割 I/O 请求，导致文件系统性能下降。提出了一种基于追溯

华中科技大学博士学位论文

文件的自适应预留空间策略 ARST，利用机器学习算法选择预留文件，预留文件在预留空间内就地更新实现有效减少文件碎片的效果。ARST 选择时间依赖性较小的文件特征，使用 AdaBoost 或随机森林算法预测历史信息较少的文件是否是预留文件，在其他情况下对于大型数据集使用决策树算法选择预留文件，对于小型数据集不使用机器学习算法，在追溯到写文件并分析文件特征后，直接决定文件是否预留。本章的主要贡献包括：

- (1) 指出智能手机上 F2FS 去碎片化工作的不足，证明移动设备上传统 F2FS 和使用现有就地更新策略的 F2FS 因为文件碎片导致反应缓慢。
- (2) 提出一种名为 ARST 的自适应预留空间策略以缓解文件碎片化，基于应用 trace 的 I/O 特征分析，合理选择文件特征和预处理数据后获得数据集，利用机器学习算法来高效地选择预留文件就地更新，而不改变普通文件的读写过程。
- (3) 提供智能手机上 F2FS 中抓取的微信、百度和王者荣耀数据 trace。在 trace 中追踪写入的文件后获得精准的预留文件，并基于追溯 fd 减少使用机器学习的文件对象。通过恰当地选择文件特征，使得能够使用 AdaBoost 和随机森林算法预测哪些历史信息较少的文件是预留文件。
- (4) 在实际平台上进行实验，实验结果表明 ARST 以可接受的开销有效减少了文件碎片和空闲空间碎片，提高 F2FS 读写性能，降低 GC 开销，在 SQLite 负载和实际应用负载中，ARST 的性能优于传统 F2FS 和使用 F2FS_IPU_FSYNC 就地更新策略的 F2FS。与 ARS 相比，ARST 能更准确地选择预留文件，缩短了重放实际应用 trace 的时间，加快了应用的运行速度，减少了空间浪费。

3 基于数据热度的多日志延迟写策略

F2FS 异地更新数据，需要通过段清理回收无效块，获得空闲空间用于写新数据。然而空闲空间碎片会增加清理开销，段清理过程复制有效数据，导致读写数据量增加，需要在保证系统正常工作的前提下减少段清理的开销。对于日志结构文件系统的数据组织布局，理想情况是热度相近的文件数据存储在一个段上，热度相近的数据更新时间也相近，整段或大量数据同时失效，该段无需迁移或迁移少量有效块就能获得一个新的空闲段。在 F2FS 的冷、温、热数据和节点中，通过数据分析发现，温数据占比最大，至少占总数据量的 80%，而温数据中不同文件块的访问热度相差较大，所有温数据被写入一个日志，造成温数据段中不同块的失效时间不同，导致空闲空间碎片增加，为回收这些空闲空间碎片段清理次数增加，回收温数据段时需要迁移大量未失效的有效块，导致迁移成本增加，影响整个系统的性能。一个文件或多个文件的文件块热度可能相同，也可能不同，本章研究区分文件块级的数据热度策略以减少空闲空间碎片数量，减少段清理开销。

为了解决上述问题，提出一种基于动态识别热度的多日志延迟写策略 M2H (Multi-log delayed writing based on Hotness)。为了准确描述温数据热度，M2H 基于工作负载的特征选择文件块更新距离、最近使用距离和读次数定义数据热度，数据热度随 F2FS 文件块的异地更新而更新；为了在文件块级准确区分数据热度，使用 K-means 聚类算法识别数据热度，并基于负载 I/O 模式的变化动态识别热度。基于细粒度的温数据热度分别优化数据的写入和清理，实现多日志写温数据并延迟提交以避免随机写性能下降，与传统 F2FS 段热度受一个有效块更新时间的影响不同使用段上每个有效块的平均热度作为段热度，分别管理被清理段上的有效块和无效块以减少检查点操作，在实际平台上部署了 M2H 策略与并分析了测试结果。

3.1 温数据冷热分离的需求和挑战

3.1.1 不同热度数据混合存储的问题

因为 F2FS 异地更新文件数据，产生大量离散分布的无效块，需要执行段清理回

华中科技大学博士学位论文

收无效数据以获得新的空闲空间。分别运行 TPC-C 和雅虎云服务负载 (Yahoo! Cloud Serving Benchmark, YCSB^[90]) 令文件系统空间使用率达到 90%，计算段清理迁移的有效块数量和文件系统写入的数据量，运行 TPC-C 和 YCSB 写入的段清理迁移数据与正常写入数据的比例分别是 1:6.67 和 1:2.96，即 TPC-C 和 YCSB 分别平均每写入 7 个或 3 个块，F2FS 段清理就需要迁移一个块，表明段清理迁移开销比较大。

F2FS 根据文件系统语义信息将文件块静态地划分为冷、温、热的节点和数据块，表 3.1 展示了不同文件系统空间使用率，不同热度倾斜的工作负载中热、温、冷数据和节点块数量的分布，Fileserver 和 Webserver 都选自 Filebench¹，工作负载特征在第 3.4.1 节详细介绍。如表 3.1 所示，无论使用何种工作负载，F2FS 的六种文件块分类中温数据都占大多数，均超过 80%。F2FS 中用户产生的大量常规文件数据被分类为温数据，从静态语义信息能推断出温数据体量大，而 F2FS 仅使用一个 log 写温数据。

表 3.1 不同负载中各类文件块的数量分布

负载	数据 (Data)			节点 (Node)			空间使 用率
	热	温	冷	热	温	冷	
Micro-benchmark	1.023%	98.736%	0.017%	0.002%	0.214%	0.008%	88%
Fileserver	1.801%	98.096%	0.001%	0.001%	0.1%	0.001%	94%
Webserver	9.969%	88.242%	0.001%	0.086%	1.701%	0.001%	99%
Postmark ²	8.559%	89.669%	0.001%	0.085%	1.685%	0.001%	94%
TPC-C	0.007%	82.613%	16.555%	0.003%	0.807%	0.015%	50%
YCSB	1.801%	98.093%	0.001%	0.001%	0.103%	0.001%	86%

F2FS 中有大量的温数据，运行 tpcc-mysql³负载并计算各个文件块的写入次数，当空间使用率为 50%时，文件系统中所有有效块的写次数如图 3.1 所示。因为 F2FS 异地更新数据，文件块的写入次数随文件块异地更新，每个点的横坐标表示 50% 空

¹ <https://github.com/filebench/filebench>

² <https://www.yumpu.com/en/document/view/18367213/postmark-a-new-file-system-benchmark>

³ <https://github.com/Percona-Lab/tpcc-mysql>

间使用率时，文件块目前的逻辑地址，纵坐标对应文件块的写次数。不同文件块间的累计更新次数存在较大差异，而 tpcc-mysql 负载产生的数据被 F2FS 分类为温数据，计算其热、温、冷节点块数和数据块数，发现温数据和节点分别占 82.61% 和 0.81%。综上 F2FS 中温数据占大多数，但不同热度的温数据存储在一块区域。

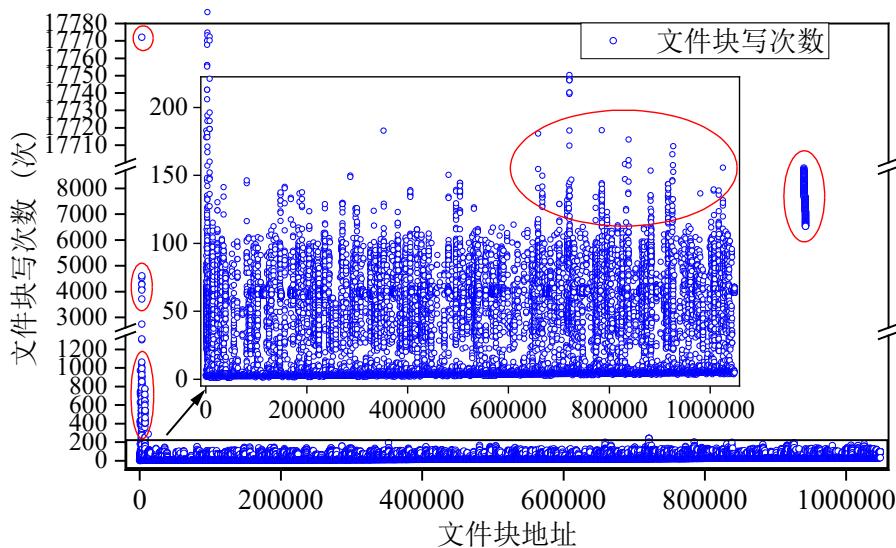


图 3.1 文件系统空间使用率为 50% 时 TPC-C 的文件块地址写次数

分析上述实验结果得到一些关键发现，(1) 在 F2FS 的冷、温、热数据和节点块中，温数据所占比例最大，至少占 80%。(2) 温数据中不同文件块的访问频率差异大。所以需要区分不同热度的温数据分布以减少段清理开销，图 3.2 中的理想情况是预期目标。热度相差较大的温数据混合存储造成更新时间不同的有效数据混合存储，如图 3.2 (a) 蓝色椭圆圈中的段 2 所示，段上较热数据失效的同时较冷的数据仍然有效，导致图 3.2 (a) 中段清理选择的清理对象迁移成本高于图 3.2 (b) 中的迁移成本，导致迁移开销增加，需要在现有热度分类上进一步区分温数据的热度，并基于细粒度区分的数据热度优化 F2FS 性能。

理想的数据分布如图 3.2 (b) 所示，不同热度的数据存储在不同的段，热度相近的数据存储在同一个段，较热的数据块因为异地更新更快失效成为无效空间，如图 3.2 (b) 蓝色椭圆圈中的段 2 所示，较热的数据因为更新失效后，选择存放较热数据的段作为清理对象，无需复制有效块，只需要修改段的元数据信息就可以作为新段。与一个段中所有有效块失效相比，离散分布的有效块也会增加空闲空间碎片，图 3.2

(a) 中是三个空闲空间碎片，而图 3.2 (b) 中是一个空闲空间碎片，空闲空间碎片增加段清理开销，导致更高的迁移开销，如选择图 3.2 (a) 蓝色椭圆圈的段作为清理对象，需要迁移两个有效块。实际情况中不同热度的数据混合存储导致段清理开销增加的问题不可忽略，频繁的段清理会迁移大量的有效数据，降低文件系统性能的同时也缩短闪存寿命。

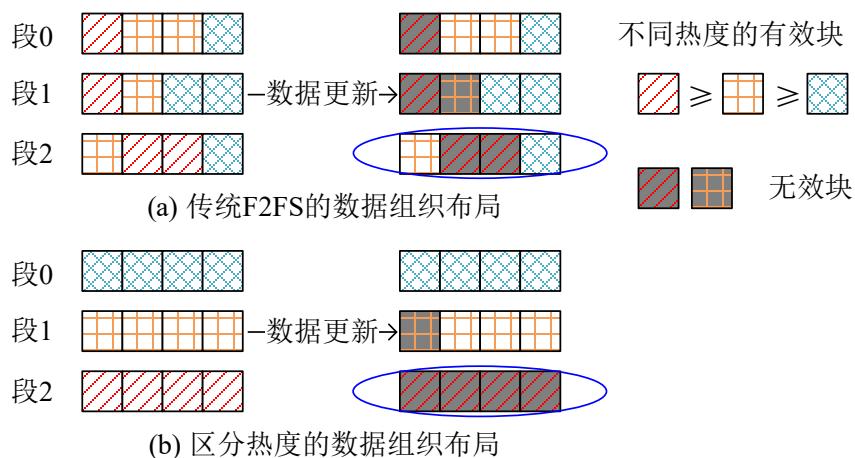


图 3.2 不同热度的温数据在传统 F2FS 与区分数据热度的系统中的数据组织布局

3.1.2 区分数据热度面临的挑战

在 F2FS 中区分数据热度有两个挑战，(1) 热度的定义和管理，需要准确描述数据热度的参数，访问频率^[91]和访问时间间隔^[79]是常用的参数。就地更新的文件系统可以使用逻辑块地址 (Logical Block Address, LBA) 管理文件块热度，但是 F2FS 异地更新文件块时相应地修改了文件块的逻辑块地址，如何管理数据热度成为难点。

(2) 准确地动态识别热度，如果识别的热度不准确，根据热度做出的优化将无法生效，此外系统中负载数据的 I/O 模式随着时间急剧变化^[93]，如果识别的热度不能随着 I/O 模式变化，则所区分的热度仍然不准确，因此准确地动态识别热度是关键。

3.2 基于 K 均值聚类的数据热度识别

3.2.1 模块设计

为了解决 F2FS 中不同热度的温数据混合存储加剧段清理开销的问题，提出基于

动态识别的热度设计多日志延迟写策略 M2H。图 3.3 展示了 M2H 设计架构，M2H 部署在文件系统层，基于热度参数使用 K-means 聚类算法识别数据热度，基于精准识别的数据热度将温数据写入多个日志并优化段清理，最后将数据写入闪存设备。

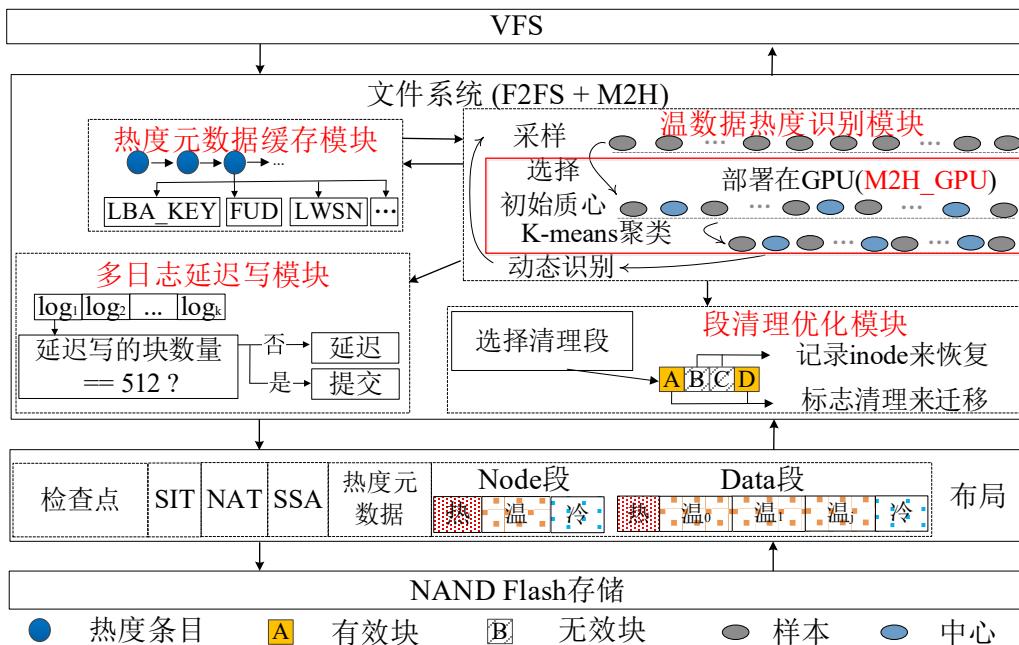


图 3.3 M2H 策略模块处理过程示意图

整体上 M2H 分为四个模块，分别是温数据热度识别、热度元数据缓存、多日志延迟写和段清理优化模块。在温数据热度识别模块，经过采样、选择初始质心和决定 K-means 算法的 K 值，使用 K-means 聚类算法识别热度。在配置 GPU 的场景中，因为 GPU 计算能力比较强，不需要采样就能完成大规模数据计算，使用 GPU 完成聚类过程，将聚类获得的质心传递给文件系统用作热度标准。热度元数据缓存模块与温数据热度识别模块间互相传递数据，当热度元数据缓存模块追踪到 I/O 模式的热度变化时，将其反馈给热度识别模块进行热度的动态识别。基于热度识别模块区分数据热度，多日志延迟写模块将不同热度的数据批量写入对应热度的 log，并在 log 大小达到一个段大小时提交。段清理优化模块使用热度识别模块维持的热度优化被清理段的选择算法，并分别记录被清理段上的有效块和无效块，使得释放被清理段时不用检查点就能维护系统的一致性。

接下来主要介绍 M2H 设计中热度的定义与管理、热度的准确识别、热度元数据

缓存、多日志延迟写和段清理优化策略。

3.2.2 热度的定义与更新

一些 SSD 和瓦记录磁盘的性能优化工作^[92]基于频率和时间局部性定义热度，频率为遍历历史访问信息计算更新次数，时间局部性关注最近时间的更新。结合频率和时间局部性的定义，根据负载读写特点采用三个参数定义热度，分别是文件块更新距离(File Update Distance, FUD)、最近使用距离(Most-Recently-Used Distance, MRUD)和读次数(Read)。

$$FUD = CWSN - LWSN \quad (3.1)$$

$$MRUD = SN_{K-means} - LWSN \quad (3.2)$$

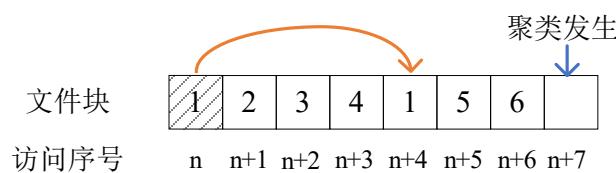


图 3.4 FUD 和 MRUD 的计算示意图

经典缓存替换算法 LIRS (Low Inter-reference Recency Set Replacement)^[94]采用最近连续访问同一个数据块期间访问其他非重复数据块的个数，用于反映页面的访问频率，受此启发定义文件块更新距离反映数据更新的热度，根据最近两次访问期间的文件块数量计算 FUD，使用式 (3.1) 计算 FUD，CWSN 是自文件系统启动后维护的当前写入文件块序列号 (Currently-Written Sequence Number)，LWSN 是该文件块上一次的写入序列号 (Last-Written Sequence Number)，用间隔的文件块数量衡量距离。文件块的访问频率越高，FUD 值越小，当文件块更新时，FUD 同时更新，但是 FUD 值并不能准确反映数据块的热度，考虑如下情形，当触发 K-means 聚类时，距离聚类发生时文件块序号较远的文件块和距离聚类发生时文件块序号较近的文件块的 FUD 相同，但后者是未来一段时间内更有可能被访问到的，更容易变成热的数据块。抓取 TPC-C 负载的文件块 FUD，发现聚类时有些最近更新的文件块 FUD 与旧文件块 FUD 相同，即使两个文件块的 FUD 相同，它们的热度也可能不同，为进一步区分数据热度提出了最近使用距离 MRUD，指文件块最近一次更新到 K-means 聚

华中科技大学博士学位论文

类时文件块位置的距离，使用式（3.2）计算， $SN_{K\text{-means}}$ 表示发生 K-means 聚类时的文件块序列号，当一个块在一个聚类周期内几乎没有更新时，该块的 MRUD 值比较大，表示其热度较小。一个具体的统计 FUD 和 MRUD 的示例如图 3.4 所示，对于文件块 1，其 $FUD=(n+4)-n=4$ ， $MRUD=(n+7)-(n+4)=3$ 。

FUD 和 MRUD 是写相关的特征，此外观察到一些负载的文件块读次数比较大，用读次数作为热度标准，使读次数较多且大概率同时被读取的文件数据块存储在同一块区域，保护数据的读局部性。一个文件中数据的读写具有空间局部性，对服务器工作负载的分析表明，大多数读请求发生在只读数据上，而大多数写请求发生在只写数据上^[95]。Postmark 负载的读吞吐量也启发关注读请求和维护读数据的局部性，以避免降低读性能，测试 Postmark 读写带宽，从实验结果发现在某些情形下仅用 FUD 定义热度时，Postmark 的读带宽低于传统 F2FS，图 3.5 展示了 Postmark 的读带宽，当文件系统空间使用率为 60% 时，M2H 读带宽急剧下降，比传统 F2FS 减少了 82.64%，由于此时连续的大段空闲空间被占用，导致新写入数据的连续性不佳，触发的段清理次数迅速增加，而数据热度仅关注 FUD，没有关注读请求次数，没有保证读次数较多的新写入数据的读局部性。当空间使用率从 40% 增加到 70%，段清理次数快速增加，传统 F2FS 和使用 FUD 定义热度的 M2H 的读带宽都急剧下降。仅使用 FUD 时过于关注写特性而忽略了维护读性能，影响了读操作比例高的工作负载性能，在具有较高读操作比例的工作负载中，需要保证读请求数据的局部性。

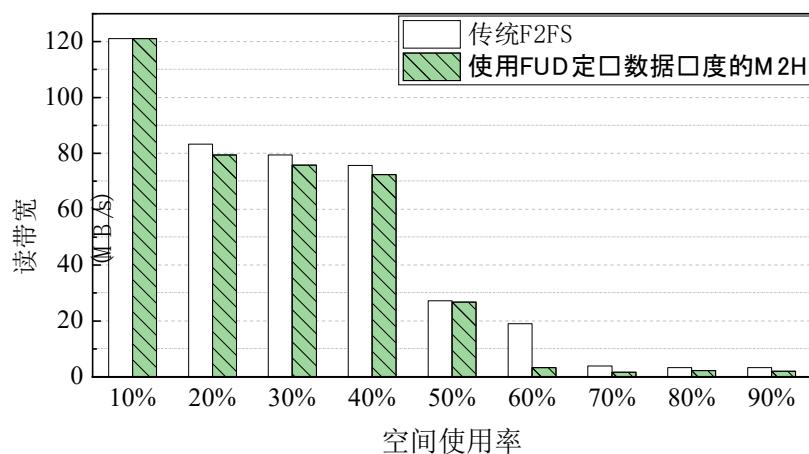


图 3.5 Postmark 负载的读带宽

综上所述，热度标准包含三个部分：FUD、MRUD 和读次数，根据工作负载读写特征选择热度定义，对于读操作比例较大的工作负载，如 Webserver 和 Postmark，选择 FUD、MRUD 和读次数定义热度，对于写操作比例较大的工作负载，如 Fileserver 和 TPC-C，增加读次数标准对总体读写性能提升不大，对于这些工作负载，选择 FUD 和 MRUD 定义热度。

就地更新的文件系统使用文件块逻辑地址作为关键字直接管理数据热度是可行的，而 F2FS 每次更新文件块时都分配一个新的 LBA，不能使用固定的 LBA 直接表示某一文件块，使用 LBA 作为关键字的统计不能正确表示热度。为了解决该问题，热度和 LBA 随着文件块的异地更新而更新，当一个文件块更新时，热度信息从旧 LBA 迁移到新 LBA。

3.2.3 准确识别热度

传统 F2FS 通过语义信息，如文件角色（目录或常规文件）和文件类型（多媒体文件），静态区分数据热度，分类开销比较低，用户产生的大量常规文件数据被分类为温数据，而不同温数据块间热度存在明显差距，需要进一步进行区分。传统方法根据经验设置多级阈值区分热度，存在对访问模式动态变化敏感性不足的问题，当热度变化后，之前设置的阈值将失效，无法获得准确的热度分布。数据聚类通过模型计算发现数据之间的内在关联，根据属性的相似性对数据进行分组，能够准确灵活地识别热度。F2FS 以块为单位完成读写操作，一个块大小为 4 KB，在文件块级管理数据热度，100 GB 的文件系统有 26214400 个文件块，即使将空间缩小到 1 GB，手动识别 262144 个块的热度也比较有难度。相比之下，机器学习可以有效地计算大量数据热度，并适应动态变化的访问模式，ASA-FTL^[79]和 HAML-SSD^[80]使用 K-means 聚类算法对 SSD 中的热、温、冷数据有效进行分类，因此采用 K-means 聚类算法可用于分类 F2FS 温数据的热度。

K 均值聚类算法是经典的机器学习算法，首先选择代表 K 个类别中心的初始质心，并将每个数据分配给距离最近的类别，然后重新计算每个类别的平均值作为新的质心，并迭代分配过程，直到质心不再移动^[96]或达到指定的迭代次数。K-means 聚类算法的复杂度为 $O(nkl)$ ，n 为聚类对象数量，k 为类别数量，l 为迭代次数，式 (3.3)

表示每个数据到每个聚类中心的欧式距离，其中 X_i 表示第 i 个数据 ($1 \leq i \leq n$)， C_j 表示第 j 个类别中心 ($1 \leq j \leq k$)， X_{it} 表示第 i 个数据的第 t 维属性，M2H 中 $1 \leq t \leq 3$ ，最少使用一维属性 FUD，最多使用三维属性， C_{jt} 表示第 j 个类别中心的第 t 维属性，使用 K-means 聚类算法需要在样本的代表性和聚类计算开销间权衡，以较低计算开销准确识别数据热度。

$$\text{distance}(X_i, C_j) = \sqrt{\sum_{t=1}^m (X_{it} - C_{jt})^2} \quad (3.3)$$

合理地决定 K 值是 K-means 聚类算法高效准确识别数据热度的关键，较好的 K 值能准确描述数据热度类别数，并减少聚类计算成本，在文件系统中使用 K-means 聚类算法需要减少聚类计算成本，同时实现感知 I/O 模式变化的数据热度动态识别。

(1) 决定 K 值

K-means 聚类算法需要预先定义 K 个初始质心， K 值会影响聚类效果^[97]，常见的决定 K 值的方法有：肘部法则（Elbow Method）^[98]、间隔统计量（Gap Statistic）^[99]和轮廓系数（Silhouette Coefficient）^[100]。肘部法则适用于 K 值比较小的情况，在不同 K 值的损失函数图中，找出 K 值增大过程中，损失函数值下降幅度最大的位置所对应的 K 值，即为肘部，肘部法则是强依赖经验的方法，误差平方和是肘部法则的核心指标。斯坦福大学的 Tibshirani 等人提出了间隔统计量^[99]方法，需要找到最大间隔统计量所对应的 K ，即 $\text{Gap}(K)$ ，使用式 (3.4) 求 $\text{Gap}(K)$ ，当划分 K 组时对应的损失函数为 D_k ，通过蒙特卡洛模拟产生的 $E(\log D_k)$ 是 $\log D_k$ 的期望，求取 $\text{Gap}(K)$ 最大时对应的 K 值。轮廓系数法使用式 (3.5) 求某个数据 X_i 的轮廓系数，其中， a 是 X_i 与同类别其他数据的平均距离，称为凝聚度， b 是 X_i 与最近类别中所有数据的平均距离，称为分离度，用 X_i 到某个类别所有数据的平均距离作为衡量该数据到该类别的距离，选择离 X_i 最近的一个类别作为最近类别，求出所有数据的轮廓系数后求平均值就得到了平均轮廓系数，在 $[-1, 1]$ 范围内，平均轮廓系数最大的 K 就是最佳聚类数。实际场景中获取 K 值比较复杂，使用肘部法则、间隔统计量或轮廓系数可能无法获得唯一准确的 K 值，需要综合实际情况做出判断。

$$\text{Gap}(K) = E(\log D_k) - \log D_k \quad (3.4)$$

$$S = \frac{b-a}{\max(a,b)} \quad (3.5)$$

基于 F2FS 区分数据热度可以减少 GC 次数，以 GC 触发次数作为衡量指标进行了 K 值敏感性评估实验，图 3.6 展示了不同 K 值的 GC 次数，用红色圆圈圈出了各个负载中最小的 GC 次数，横坐标表示触发最少 GC 次数的 K 值。从图 3.6 中看到，有些工作负载的 GC 次数对 K 值敏感，如 TPC-C 和 YCSB，而另外一些工作负载的 GC 次数对 K 值则不敏感，如 Postmark, Micro-benchmark 模拟均匀分布的数据热度，TPC-C 和 YCSB 的数据热度差异较大，在 TPC-C 和 YCSB 中，随着分类数 K 的增加，GC 次数减少，随着分类数 K 的不断增加，下降趋势逐渐平缓。本章实验中评估 Micro-benchmark、Fileserver、Webserver、Postmark、TPC-C 和 YCSB 时，K 值分别选择为 5、11、4、4、15 和 12，当访问模式发生明显变化时，可以离线校准 K 值。

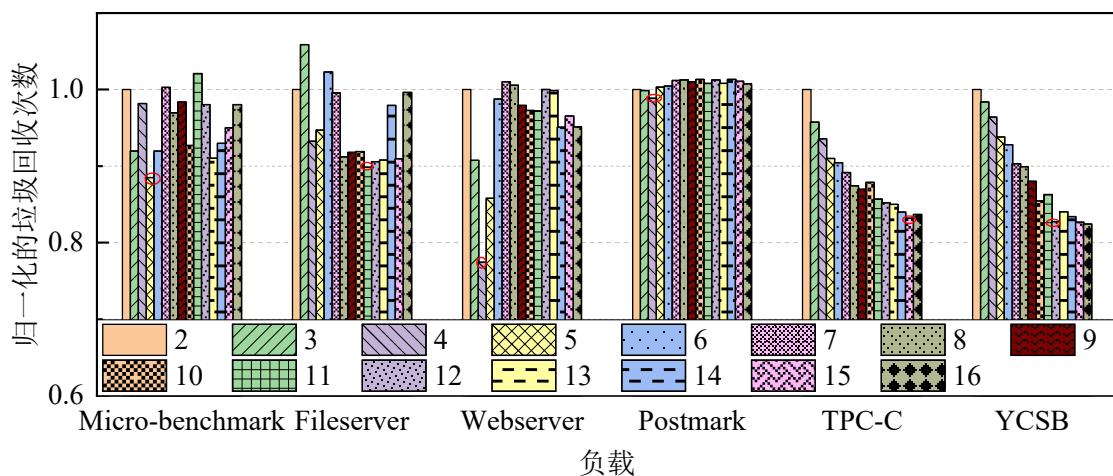


图 3.6 对各个负载，不同 K 值时系统的垃圾回收次数

(2) 减少 K-means 聚类开销

为了减少在内核使用 K-means 聚类算法的计算开销，M2H 基于时间将数据一分为二后，在时间较近的一半数据上采样 70%，在时间久远的一半数据上采样 30%，使用分层统计方法选择初始质心。为了进一步减少计算开销，在接受少量数据热度分类不准确的前提下，M2H 仅使用 FUD 表示数据热度以减少聚类时的属性维度。

为进一步降低基于普通采样使用 K-means 聚类的计算开销，提出了 M2H_MBK，使用 Mini Batch K-means 算法的采样方法选择样本，使用 K-means++^[101]选择初始质心。Mini Batch K-means^[102]是对 K-means 聚类算法的改进，主要用于减少计算时间，与 K-means 每次进行聚类迭代时使用数据集中所有数据不同，Mini Batch K-means 使

用输入数据的一部分^[103]，在每次训练迭代时随机抽样，有效减少了计算量，适用于大型数据集。Mini Batch K-means 聚类比 K-Means 聚类速度快，花费的时间少，聚类质量一般略低于 K-Means 聚类^[104]，但也可能优于 K-means 聚类^[105]，与具体数据集有关。M2H_MBK 在内核中部署 Mini Batch K-means 算法，使用随机抽样选择一部分样本，对不同的负载使用敏感性实验确定的不同 K 值，使用 K-Means++选择初始质心，直到前后两次迭代的类别中心点不变。

在配置 GPU 的场景下进一步减少聚类开销，为保证数据完整性，提出 M2H_GPU 策略，基于所有数据样本，使用 K-means++初始化质心，由 GPU 承担 K-means 聚类计算，当一次迭代后样本的相对重新分配数据量小于 1% 时，K-means 聚类停止。使用 GPU 实现 K-means 聚类时，F2FS 维护的热度元数据作为 K-means 聚类算法的输入传递给 GPU，经 GPU 计算得到类别质心并通过/sys 传递给 F2FS，放置在 *f2fs_sb_info* 结构体中，传输的数据量在未来可以进一步减少。

为监控数据热度变化，使用 K-means 聚类算法在线识别数据热度。对于 TPC-C 工作负载，假设区分的类别数为 10，M2H 区分 10 GB 大小文件系统的数据热度需要 1.64 到 4.15 毫秒，发生在文件系统内；M2H_MBK 使用 Mini Batch K-means 聚类区分 10 GB 大小文件系统的数据热度需要 0.546 到 1.504 毫秒，该时间远小于 M2H 使用 K-means 聚类花费的时间；M2H_GPU 区分 10 GB 大小文件系统的数据热度需要 2.275 到 3.319 秒，由 GPU 承担，GPU 计算不在关键路径上，和 F2FS 工作并行，获得热度识别标准真正需要等待的是 GPU 将分类质心传输到文件系统的时间，该传输时间比较小。

(3) 动态识别数据热度

每更新一个文件块时，该文件块的 FUD 也更新，MRUD 在数据聚类发生时更新，考虑到 FUD 的变化比 MRUD 和读次数敏感，M2H、M2H_MBK 和 M2H_GPU 定义数据热度时都有使用 FUD，如算法 3.1 所示，基于文件块更新距离对数据热度变化的敏感性提出了一种动态识别热度的算法，以感知 I/O 模式的变化实现数据热度的动态识别。对于一轮跟踪，M2H 在写入文件块时跟踪数据热度变化，如果新旧 FUD 两者之间的差距超过了热度差异的阈值，热度发生变化的写入数会增加，跟踪

华中科技大学博士学位论文

的写操作数量也被记录，直到达到追踪的写入数阈值 $track_threshold$ ，算法 3.1 中 $track_threshold$ 设置为 5000。选择 5000 是为了在访问模式的变化和足够大的样本数量间平衡，该阈值能根据实际场景的数据量发生变化。当达到 5000 时，M2H 停止跟踪，并计算发生变化的写入数与所跟踪的写入总数的变化比例，如果变化比例超过预定义的热度发生变化的阈值，则认定当前热度分布发生了变化，M2H 重新识别热度，否则继续追踪数据热度变化。

算法 3.1 热度的动态识别算法

输入: 追踪的写入数阈值, $track_threshold$;

追踪的写入数, m ;

热度发生变化的写入数, n ;

热度发生变化的占比阈值, $change_threshold$;

系统当前写入文件块的序列号, $CWSN$;

文件块上次写入的序列号, $LWSN$;

FUD 热度差异的阈值, $FUD_threshold$;

旧的 FUD 值, FUD_{old} ;

输出: 重新聚类计算识别的热度标准;

```
1: 文件块写入,  $m++$ ;  
2:  $FUD_{new} \leftarrow CWSN - LWSN$ ;  
3: if  $FUD_{new} - FUD_{old} > FUD\_threshold$  then  
4:    $n++$ ;  
5: end if  
6: if  $m < track\_threshold$  then  
7:   继续追踪;  
8: else  
9:   if  $\frac{n}{m} > change\_threshold$  then  
10:    重新聚类计算热度识别标准;  
11:    开始新一轮的追踪,  $m \leftarrow 0, n \leftarrow 0$ ;  
12: else  
13:   继续追踪;  
14: end if  
15: end if
```

3.2.4 热度元数据缓存

所有的热度元数据持久化存储在闪存中，为了有效地管理热度元数据，设计了一个热度元数据缓存策略缓存部分数据，通过基数树索引数据，并选择最近最少使用算法进行替换。缓存使用热度项存放文件块级的热度信息，用无符号整型数记录 LWSN、FUD、MRUD 以及读次数，用于记录的字段大小为 4 B，M2H 使用 FUD 定义热度，一个热度项包含 LWSN 和 FUD，大小为 8 B，一个 4 KB 大小的块可以存储 512 个项，当 M2H_MBK 和 M2H_GPU 使用 FUD、MRUD 和读次数定义一个块的热度时，一个热度项的大小为 16 B，一个 4 KB 大小的块可以存储 256 个项。在缓存替换过程中，当脏项数较多时，下刷热度元数据使其持久化，测试不同的热度元数据缓存长度对命中率的影响，分析发现当缓存长度占文件系统热度元数据总数的 1/20 时，能在缓存命中率和置换开销间较好地权衡。

3.3 多日志延迟写及段清理策略

3.3.1 多日志延迟写数据

F2FS 以日志结构的方式写，提高了随机写性能，而仅使用一个 log 写温数据容易导致不同热度的数据混合存储。为了改善多文件数据组织布局，提出多日志写入温数据方法，将不同热度级别的数据写入相应热度的日志，使得热度相近的数据存放在相同区域，这些数据失效时间相近能减少空闲空间碎片，但写多个 log 会增加随机写操作，带来额外开销。测试了 TPC-C 负载在文件系统不同空间使用率下分别采用单日志和多日志模式的写带宽，单日志模式即传统 F2FS 使用一个 log 写温数据，多日志模式根据热度差异将数据写入 15 个日志中。分析实验结果发现当文件系统空间使用率小于 60% 时，多日志模式的写带宽明显高于单日志模式的写带宽，这是由于区分数据热度的多日志写入可以有效减少空闲空间碎片和段清理开销；当空间使用率高于 60% 时，多日志写带宽下降，甚至低于单日志写带宽，空间使用率较高时 F2FS 没有无效块的连续空闲空间耗尽，段清理次数急剧增加，分配给每个日志的地址是离散的，此时多个日志写数据的逻辑地址过于离散，离散的写请求地址导致写性能下降幅度较大，超过了多日志区分数据热度减少段清理带来的收益，导致 F2FS 写性能急

剧下降，需要将写请求落在一个逻辑地址空间以提高随机写性能^[106]。

延迟向文件系统提交多级日志，达到增大数据写请求提高写性能的目的。现有研究表明随机写性能与写请求地址的离散程度和写请求大小密切相关^[14]，写请求比较大时，可以显著减少随机写请求地址离散带来的性能下降，以提高写性能。不同热度的数据写入相应日志后，不同于传统 F2FS 直接提交日志数据，M2H 延迟提交，直到相应日志中的数据大小累积到一个段大小才提交写数据请求，段是清理操作的单位，选择一个段大小可以更好地减少段清理次数，原因在于一个段上热度相似的数据几乎可以同时失效，选择清理一个失效块较多的段能以较小的迁移开销获得空闲段。

3.3.2 段清理优化

F2FS 清理过程包括选择清理对象、识别和迁移有效块以及等待检查点同步回收被清理的段，基于数据热度优化清理对象的选择与释放。相比于较热的数据，将较冷的数据选作清理对象更有价值，原因在于较热的数据会在短时间内更新后失效。F2FS 后台段清理使用成本-收益（Cost-Benefit，CB）算法考虑有效块的数量和段的年龄，选择一个有效块较少和热度较冷的段作为清理对象，但其统计的年龄是一个粗略的估计，它使用一个块的最近修改时间作为段的年龄，导致对段中某一个文件块的修改会影响整个段的热度，可能存在某一个文件块更新而其他文件块长时间未更新的情况，导致段清理需要迁移比较多的有效块。M2H 选择段清理对象时使用段中每个有效块的平均热度决定段的热度，采用 LWSN 来计算平均热度，没有选用读次数的原因在于一些负载没有使用读次数属性表示数据热度，M2H 计算一个段上有效块的平均 LWSN， $avg_lwsn_in_seg$ ，基于式（3.6）计算段的年龄，为了与式（3.7）中段使用率 u 的权重均衡，使用常数 N 缩小年龄值。基于权衡年龄和段使用率的 CB 算法，由式（3.7）计算迁移成本 seg_mg_cost ，其中 u 表示段使用率， $UINT_MAX$ 是无符号整数的最大值， u 越小， age 越大时， seg_mg_cost 越小，用 seg_mg_cost 选择一个迁移成本比较低且年龄较大比较冷的段作为清理对象。

$$age = \frac{CWSN - avg_lwsn_in_seg}{N} \quad (3.6)$$

$$seg_mg_cost = UINT_MAX - \frac{100 * (100 - u) * age}{100 + u} \quad (3.7)$$

在识别和迁移有效块时，M2H 增加清理标志，用于追踪被清理段中有效块的移动，并在专门空间记录无效块的 inode 号，当该段被覆盖写后发生异常掉电时能通过清理标志和记录的 inode 号在前滚恢复时恢复该段的数据，保证文件系统的一致性。在第三步等待检查点同步回收被清理的段中，F2FS 为保证系统一致性需要创建检查点，释放被清理的段。M2H 释放被清理的段时，不用触发检查点操作，这是由于对于有效数据，下刷脏数据并为其直接索引节点添加清理标志，对于无效数据，将其 inode 号存储在特定位置用于恢复数据。当意外断电发生时，M2H 通过清理标志直接恢复有效数据，通过段摘要区找到无效数据的直接索引节点，并在前滚恢复中恢复数据，因此对于被清理段的数据迁移，无需做检查点操作就能够保证系统一致性，减少了触发检查点操作增加的读写开销。

3.4 性能评估与分析

在 Linux 内核 4.13.0 中使用 872 行 C 语言代码实现了 M2H 原型，M2H_MBK 基于 M2H 实现，使用 1622 行 C 语言代码实现了 M2H_GPU 的原型。在不改变其他文件系统元数据、node 和数据块管理的情况下，在文件块级进一步区分温数据的热度。

3.4.1 实验环境

使用 F2FS 现有策略作为比较基准，在下文称作传统 F2FS，Li 等人分别提出多级阈值同步写入技术（Multi-level Threshold Synchronous Write，MTS Write）减少文件碎片和高检测频率后台段清理机制（High Frequency BSC）优化段清理^[69]，将其作为比较方案之一，使用 MWHFB 表示，比较传统 F2FS、MWHFB、M2H、M2H_MBK 和 M2H_GPU 的性能和开销。

实验平台的基本参数如表 3.2 所示，使用的服务器上配置了 NVIDIA Quadro RTX 6000¹。M2H 方案在服务器和移动场景的 F2FS 上都能使用，本章是在服务器上评估 M2H 性能，一方面是因为 M2H_GPU 使用 GPU 加速聚类算法，而移动设备文件系统中暂时没有开放使用 GPU 的协议接口，在移动设备存储系统中使用 GPU 需要移

¹ <https://www.nvidia.com/en-us/design-visualization/quadro/rtx-6000/>

华中科技大学博士学位论文

动设备厂商支持；另一方面是缺少数据集，解析基于 Mobibench 收集的半小时应用 trace 需花费一天以上的时间，然而半小时 trace 的数据热度差异可能不明显，触发的 GC 次数少，无法衡量 M2H 区分数据热度减少段清理的效果，此外使用 fio 合成负载的数据不够真实，无法评估实际场景的性能。所以选择在服务器上衡量 M2H 的性能，不同热度的温数据混合存储的问题在移动设备 F2FS 上同样存在，使用 M2H 优化同样能提升性能。

表 3.2 实验平台的软件和硬件参数

部件参数	具体信息	部件参数	具体信息
处理器	Intel Xeon Silver 4208, 2.10 GHz	Memory	128574 MB
系统	Ubuntu 18.04	内核	Linux 4.13.0
SSD	Intel 730	SSD 接口	SATA
SSD 容量	480 GB	NAND Flash 存储	MLC
顺序读/写带宽	550/470 MB/s	随机读/写 IOPS	89000/74000 IOPS
GPU	NVIDIA Quadro RTX 6000	GPU 容量	24220 MB

表 3.3 展示了工作负载的细节参数，参考 SFS^[14] 和 ParaFS^[15] 中的衡量指标，选择 Filebench、Postmark、TPC-C 和 YCSB^[90] 作为衡量性能的工作负载。为了衡量空闲空间碎片数量和单个文件的同步 I/O 性能，构建了 Micro-benchmark，Micro-benchmark 是由 fio 和第 2.4.2 节碎片化负载构造的工作负载。六个工作负载之间的热度倾斜不同，Micro-benchmark 创建、删除和写入时不区分冷热文件，用于模拟均匀分布的热度，在 Filebench 中选择了 Fileserver 和 Webserver 负载。TPC-C 和 YCSB 是两个实际场景的文件系统数据集，TPC-C 是一个在线事务处理（online transaction processing, OLTP）负载，它支持多种事务类型，通过在 MySQL 上运行 tpcc-mysql，在文件系统层抓取 TPC-C 数据集，测试文件系统的 I/O 性能，MySQL 是一个开源的关系型数据库。

为了全面地衡量 M2H 策略的性能，测试并分析了文件系统空闲空间碎片数量、垃圾回收次数、迁移块数量、I/O 性能、读操作次数和写操作次数以及 M2H 开销，其中 I/O 性能的评估包括单个文件的 I/O 性能和各种负载的工作吞吐量，文件系统的

华中科技大学博士学位论文

空间使用率写入到 95% 左右，以评估文件系统快填满时的性能。

表 3.3 工作负载的特征

负载	描述	文件数量 (千个)	I/O 大小	线程数 量(个)	读写 比例	是否 同步
Micro-benchmark	大量的文件创建、删除和写入，少量的文件读	N/A	4 KB	16	N/A	是
Fileserver	模拟一个文件服务器，由创建、删除、追加写、普通读和写组成	60	1 MB	16	1:2	否
Webserver	模拟一个执行文件读取和日志追加写的 Web 服务器	60	1 MB	100	10:1	是
Postmark	NetApp Postmark 基准，旨在测量小文件的 Internet 应用（电子邮件、电子商务和新闻事务）的服务器性能	60	1 KB	16	1:1	是
TPC-C	在线事务处理（OLTP）基准	N/A	N/A	20	17:83	是
YCSB	用于衡量云服务质量的基本工具	N/A	1 KB	50	N/A	是

3.4.2 空闲空间碎片分布

多日志写温数据可能会导致数据离散，造成更多的空闲空间碎片，空闲空间碎片越多，需要文件系统触发更多次数的段清理。如果区分热度能获得理想的数据分布，则能有效地减少空闲空间碎片数量和段清理次数。使用碎片化负载快速老化文件系统，构造文件碎片和空闲空间碎片，并评估空闲空间碎片的数量。构造不同大小的碎片测量空闲空间碎片的数量，碎片大小的划分见表 2.1。通过/proc 中的 *segment_bits* 获得 F2FS 段中每个块的分布，分析得到的空闲空间碎片的分布如图 3.7 所示。第七级碎片表示大于 128 KB 的碎片，M2H_MBK 和 M2H_GPU 有更多大于 128 KB 的空闲空间碎片，第七级碎片的数量越多越好，碎片总数量越少越好。M2H_GPU 的空闲空间碎片总数分别比传统 F2FS 和 M2H 少 15.07% 和 12.41%。M2H_MBK 和 M2H_GPU 比 M2H 更加有效地减少空闲空间碎片，因为 M2H_MBK 优化了采样算法，M2H_GPU 不用采样保留了数据完整性，使得 M2H_MBK 和 M2H_GPU 可以更好地理解数据的分布。因为碎片化负载不断地创建文件到写满文件系统，然后间隔删除文件，文件系统的空间使用率不断变化，所以 MWHFB 减少空闲空间碎片的效果有限，总的空闲空间碎片数量比传统 F2FS 多 9.71%。

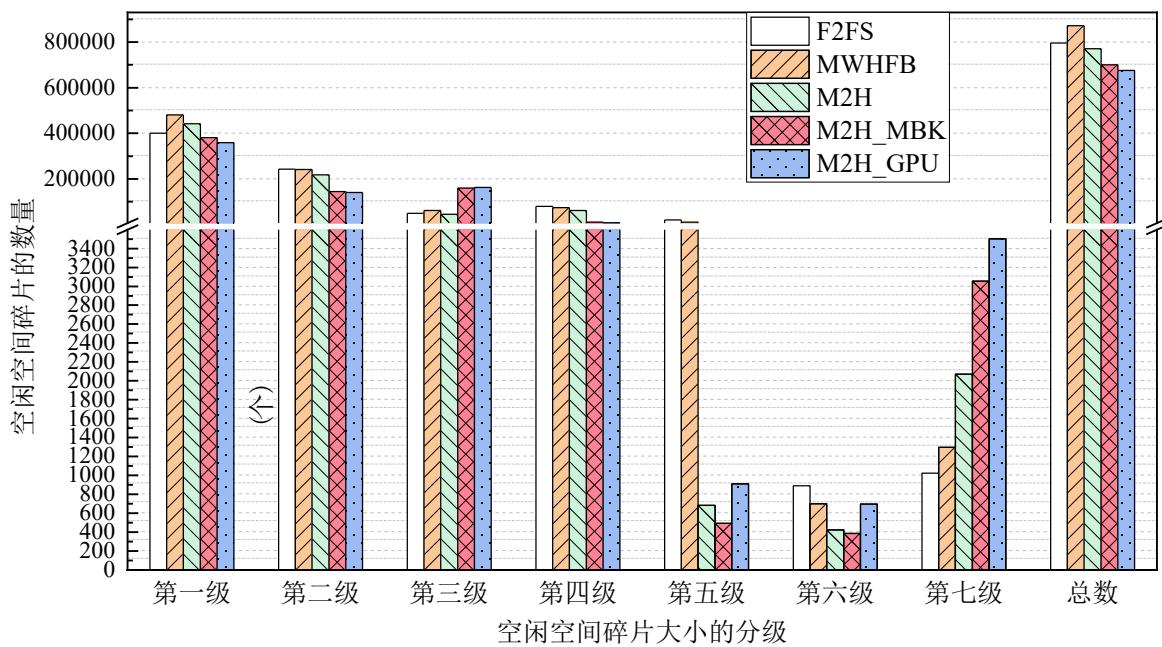


图 3.7 不同级别的空闲空间碎片数量

3.4.3 垃圾回收次数和迁移块数

表 3.4 展示了运行各个工作负载后，文件系统空间使用率到 90%时触发的 GC 次数。使用 cat status 从/sys/kernel 监视 GC 次数和迁移有效块的数量。与传统 F2FS 和 MWHFB 相比，M2H、M2H_MBK 和 M2H_GPU 有效地减少了段清理次数。与传统 F2FS 相比，Micro-benchmark、Fileserver、Webserver、Postmark、TPC-C 和 YCSB 中 M2H_GPU 触发的段清理次数分别减少了 55.97%、71.07%、97.93%、54.41%、61.92% 和 64.57%，相应地，M2H 触发的 GC 次数减少了 50.61%~94.36%。与 MWHFB 相比，M2H_GPU 触发的段清理次数减少了 54.29%~97.88%。

表 3.4 归一化的垃圾回收次数

方案	Micro-benchmark	Fileserver	Webserver	Postmark	TPC-C	YCSB
F2FS	1	1	1	1	1	1
MWHFB	1.0479	0.9987	0.9761	0.9974	1.0005	0.9989
M2H	0.4872	0.2904	0.0564	0.4669	0.4939	0.374
M2H_MBK	0.4807	0.3039	0.0587	0.4563	0.4279	0.355
M2H_GPU	0.4403	0.2893	0.0207	0.4559	0.3808	0.3543

华中科技大学博士学位论文

因为细粒度区分温数据热度后，热度相近的数据聚拢在一个段上，更热的段上的数据失效后，段上几乎没有或很少存在有效块，有效地减少了清理次数和碎片数量。当文件系统快填满时，M2H、M2H_MBK 和 M2H_GPU 的段清理次数减少和迁移的有效块数量减少能让文件系统更流畅。

表 3.5 展示了运行各个负载到文件系统空间使用率为 90% 时段清理迁移的块数量。当空间使用率达到 90% 时，与传统 F2FS 相比，Micro-benchmark、Fileserver、Webserver、Postmark、TPC-C 和 YCSB 在 M2H_GPU 中段清理迁移的有效块数分别减少了 65.62%、82.84%、99.99%、73.63%、79.08% 和 82.64%。相应的，M2H 的有效块迁移数量减少 64.31%~99.98%。

表 3.5 有效块迁移数量（百万个）

方案	Micro-benchmark	Fileserver	Webserver	Postmark	TPC-C	YCSB
F2FS	6.916012	26.584608	8.58222	48.910274	50.248654	219.903303
MWHFB	7.822676	26.540401	8.279298	48.806799	50.33193	219.594809
M2H	2.468599	4.562281	0.001599	12.977657	10.904178	37.838978
M2H_MBK	2.472683	4.60871	0.001662	12.928699	10.817285	37.59968
M2H_GPU	2.377423	4.561797	0.000587	12.895202	10.511796	38.166998

Webserver 中有效块迁移的数量以数量级减少，原因是 Webserver 中增加的读次数有利于将读操作密集的数据放置在同一个段上，此外 M2H_MBK 和 M2H_GPU 得益于良好区分的数据热度，系统中没有大量触发段清理。随着文件系统的读写压力变大，产生更多的创建和删除操作，M2H 通过区分热度减少的段清理次数和有效块迁移数量越多。因为 M2H 策略分开存储不同热度的温数据并延迟提交，所以文件系统可以获得一个如图 3.2 (b) 所示的较理想的清理对象，M2H 对段清理过程的优化也有助于选择迁移成本低的清理对象，减少因检查点操作导致的读写数据增加。

3.4.4 读写性能

(1) 单个文件的读写性能

Micro-benchmark 使用 fio 测试在运行碎片化负载后单个文件的 I/O 性能，此时

文件系统的空闲空间严重碎片化。图 3.8 展示了 Micro-benchmark 测试的单个文件的读写性能，其中文件大小为 128 MB，顺序读写的 I/O 大小为 1 MB，随机读写的 I/O 大小为 4 KB，顺序读写和随机读写的 I/O 都设置了同步使得数据落盘，所有 I/O 都设置为直接 I/O 以避免缓存对性能的影响。实验结果表明，无论是顺序读写还是随机读写，五种方案间的性能差距都很小，都没有出现文件的 I/O 性能因为温数据的日志数量增加而下降的现象，M2H_GPU 的随机写性能甚至比传统 F2FS 提升 4.41%，这是因为一个文件中的数据热度是相近的，而 M2H_GPU 构建 K-means 聚类算法的输入时选择的属性有助于将一个文件的数据放置在同一个段中，保证了数据读写时的空间局部性。

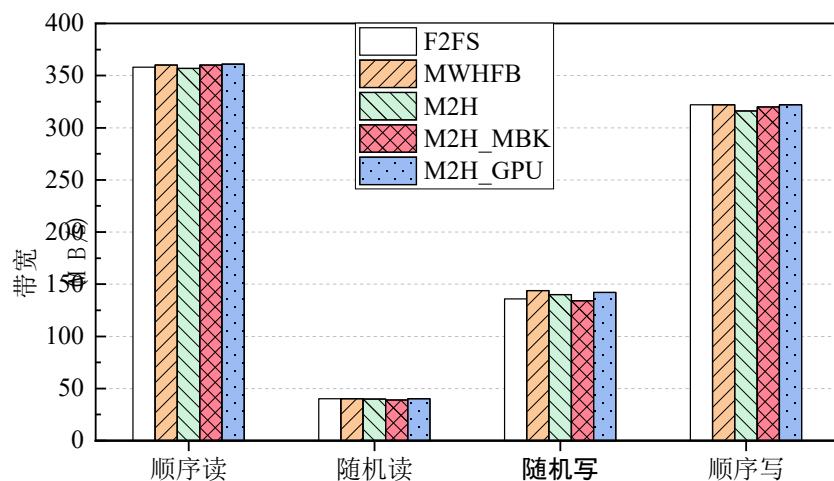


图 3.8 单个文件的读写性能

(2) 负载工作时性能

负载工作时性能是一个全面的性能评估指标，负载工作时的读写性能与段清理次数有关，全面地受系统整体情况影响。图 3.9 展示了归一化的各个负载工作时性能，图 3.9 中的文件系统空间使用率为 90%。为了更直观地比较，图 3.9 展示的是归一化的读写性能，实际上，Fileserver、Webserver 和 YCSB 的性能评估单位是 ops/s，Postmark 和 TPC-C 的性能评估单位分别是 MB/s 和 TpmC。M2H_GPU 的 Fileserver、Webserver、Postmark、TPC-C 和 YCSB 相对于传统 F2FS 分别增加了 25.27%、25.41%、13.52%、154.19% 和 69.86%。对应地，与传统 F2FS 相比，M2H 的性能分别提升了 14.3%、19.37%、3.06%、122.36% 和 35.48%。M2H 在 TPC-C 上的收益最大，在 Postmark 上

的收益最小。

M2H 性能提升主要得益于 TPC-C 负载的热度倾斜比较大，YCSB 数据读写较密集，读写压力较大，性能提升比较大。文件系统快写满时因为系统中存在大量的空闲空间碎片，需要频繁触发段清理以满足负载的 I/O 请求，导致负载工作时性能下降。在图 3.9 中，当文件系统快写满时，区分数据热度后获得较好的数据分布和更少的段清理次数，有利于 M2H_GPU 提高负载工作时性能，表明在文件系统中良好的数据分布不仅可以减少段清理次数，还可以提高负载工作时性能。

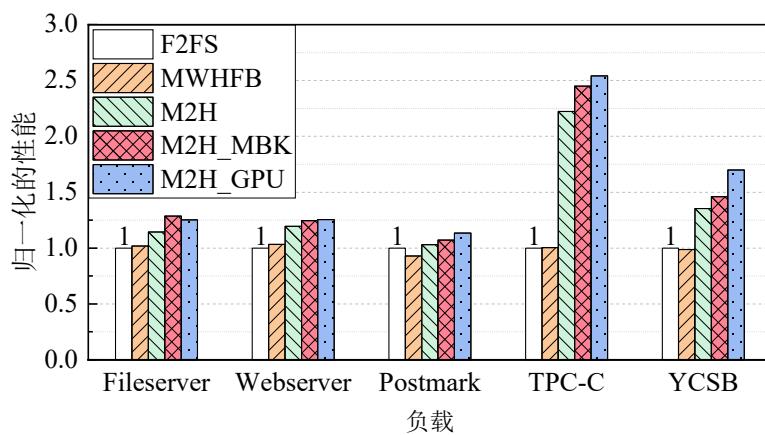


图 3.9 不同负载的工作时性能

3.4.5 读写操作次数

完成相同的 I/O 请求时触发段清理会增加读写 I/O，导致存储设备从文件系统接收并执行更多的 I/O 请求，缩短存储设备的使用寿命，读和写 I/O 次数是衡量不同方案性能的重要指标。不同方案的读写 I/O 次数的归一化结果如图 3.10 所示，与传统 F2FS 相比，对于 Micro-benchmark、Fileserver、Webserver、TPC-C 和 YCSB，M2H_GPU 的读 I/O 次数分别降低 5.38%、29.43%、99.71%、79.28% 和 42.34%。对于 Postmark，M2H_GPU 的读 I/O 次数比传统 F2FS 增加了 11.98%，这是因为 Postmark 是元数据密集型负载，文件系统的元数据和文件目录被分类为热数据，其所占的比例比较高，但这些数据不属于温数据，无法被 M2H 策略优化，而通过 `cat /sys/block/stat` 统计的是整个文件系统的读 I/O 次数，迁移有效块需要读取被清理段上所有的数据并判断是有效数据还是无效数据。M2H_GPU 在 Micro-benchmark、Fileserver、Webserver、

Postmark、TPC-C 和 YCSB 中的写 I/O 次数分别比传统 F2FS 减少 68.23%、83.99%、69.81%、59.44%、76.18% 和 70.53%。

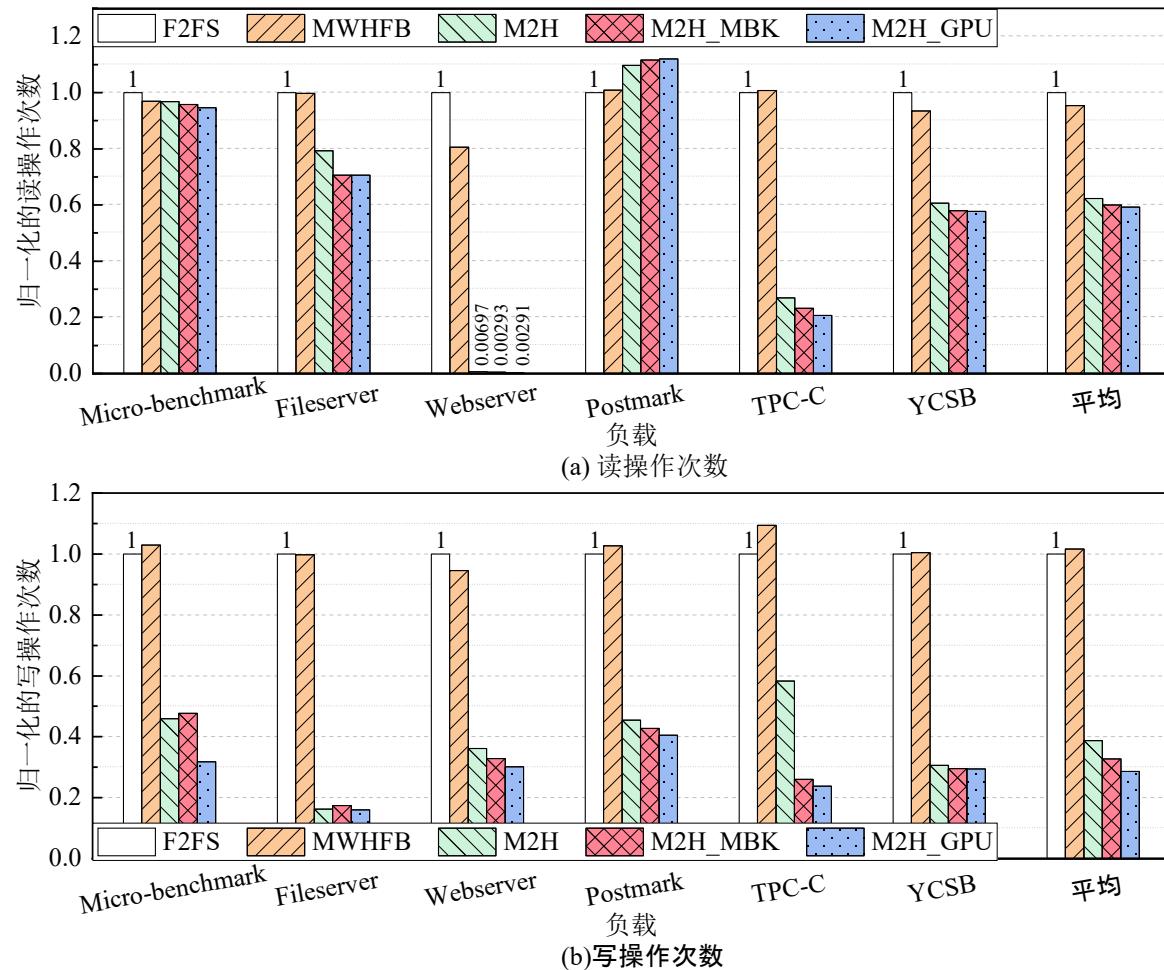


图 3.10 归一化的读操作次数和写操作次数

与传统 F2FS 相比, M2H_GPU 的读写 I/O 次数平均分别减少了 40.69% 和 71.36%, 与读 I/O 次数相比, M2H_GPU 减少了更多的写 I/O 次数, 主要原因在于段清理迁移有效块时需要读整个段的数据以判断是有效块还是无效块, 而只需要复制有效块, M2H 通过细粒度区分数据热度改善了数据组织布局, 使得选择到的清理对象包含较少的有效块, 减少了写操作, 通过减少段清理次数同时减少读写 I/O 次数。与 M2H 相比, M2H_GPU 的读和写 I/O 次数平均分别减少了 4.87% 和 26.14%, 原因在于 M2H_GPU 无需损失数据完整性的数据采样操作, 使得 K-means 聚类算法能够更加准确地区分热度。完成相同的工作任务触发更少的读和写操作有利于提高文件系统

性能，一方面，段清理的 I/O 请求减少，减少阻塞其他常规 I/O 操作的可能性，也降低缩短 SSD 寿命的可能性，另一方面，I/O 操作越少，存储软件系统的能耗就越低，降低能耗的幅度与文件系统产生的文件碎片和空闲空间碎片数量以及各个负载的 I/O 特征有关。

3.4.6 开销分析

基于 M2H 使用 K-means 聚类算法区分热度存在计算开销，在评估六种测试负载的性能时监控 CPU 和内存开销，通过 iostat 监视 CPU 使用率，通过/proc/meminfo 衡量内存使用情况，为排除偶然性做多次实验收集大量数据并取平均数值。图 3.11 比较了在聚类发生时，M2H_GPU 和其他方案的 CPU 使用率和内存使用率。

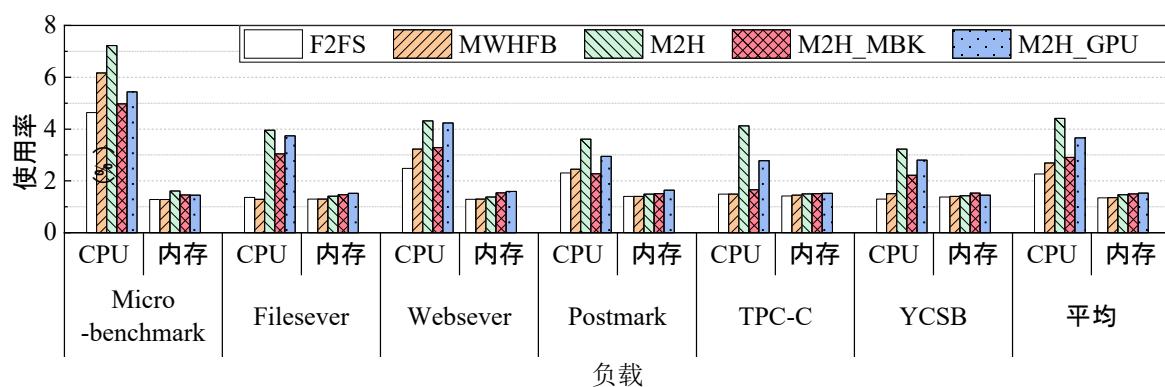


图 3.11 不同策略的 CPU 和内存开销 (M2H 测试聚类发生时的开销)

M2H 在时间更近的一半数据上抽取 70% 的样本，在另一半数据抽取 30% 的样本，M2H_MBK 使用 Mini Batch K-means 算法随机抽取样本，M2H_GPU 不需要数据采样；M2H 使用分层统计方法选择初始质心，M2H_MBK 使用 K-means++ 选择初始质心，这两种方案都使用 CPU 承担 K-means 聚类计算，M2H_GPU 使用 K-means++ 选择初始质心，使用 GPU 承担聚类计算。如图 3.11 所示，M2H_MBK、M2H_GPU 的 CPU 使用率平均分别比 M2H 降低 34.04%、17.05%，主要原因在于 M2H_MBK 使用了 Mini Batch K-means 算法，M2H_GPU 的聚类计算卸载到 GPU 完成，减少了对系统 CPU 资源的占用，M2H_MBK、M2H_GPU 平均分别比 M2H 多使用 2.18%、4.06% 的内存，主要是因为 M2H_MBK 和 M2H_GPU 增加管理 MRUD 和读次数属性数据。与传统 F2FS 相比，在 K-means 聚类发生时，M2H_GPU 的 CPU 和内存使用率平均

分别增加 61.63% 和 13.75%，在聚类发生时的短时间内是该开销，在其他时间段的 CPU 和内存使用率与传统 F2FS 持平，考虑到展示的数值大致相近，在图 3.11 中没有展示未发生聚类时各个方案的 CPU 和内存使用情况。从图 3.11 可以看到，M2H_MBK 的 CPU 和内存使用率最低，比传统 F2FS 平均分别增加 28.52% 和 11.7%，Mini Batch K-means 算法对于在文件块级识别数据热度能有效减少计算开销，对于普通服务器，建议使用 M2H_MBK 识别数据热度。从系统的角度看，与 M2H_MBK 和 M2H_GPU 带来的性能提升相比，其仅在聚类发生时产生的开销是可以接受的，聚类计算仅在建模初期和 I/O 模式发生变化需要动态更新数据热度时才发生。

对于不同的工作负载，M2H 热度元数据缓存的命中率不同，如 TPC-C 的缓存命中率优于 YCSB，这是因为 TPC-C 负载的数据局部性更强，当缓存长度是文件系统热度元数据总数的 1/20 时，其具有较高的命中率和较低的缓存替换开销。除 Fileserver 以外，其他五个负载的 I/O 是同步的，使用 M2H 策略时负载能正常工作并获得性能提升，说明该策略保证了系统的一致性，在 Micro-benchmark 中测试文件性能时使用的是直接 I/O，说明 M2H 也适用于直接 I/O。

负载的 I/O 性能与 F2FS 段清理密切相关，当段清理数量迅速增加时，I/O 性能会明显下降，然而当文件系统快被写满或文件系统中有大量空闲空间碎片时，必须通过段清理回收无效块，否则无法满足 I/O 请求，而过度的段清理使 I/O 性能下降，要求文件系统妥当处理段清理，也要求对段清理请求与 I/O 性能进行权衡。M2H 策略的收益与负载特征相关，负载数据热度差异越明显，M2H 策略性能提升越大，方案的开销与负载的 I/O 模式有关。

3.4.7 未来改进

M2H_GPU 通过将 K-means 聚类计算卸载到 GPU 以减少 CPU 开销，但 M2H_GPU 原型是简单实现从缓存传递到 GPU 的热度元数据和从 GPU 传递到文件系统的质心，可以通过优化数据结构或数据传输算法实现成本更低的数据传输。这是因为在文件块级以 4 KB 为单位管理热度，较大的数据量造成 M2H 存在管理热度元数据的内存开销。M2H_MBK 通过更好的采样算法，M2H_GPU 无需用数据采样，使得采样的数据更具代表性或数据真实性无损失，K-means 聚类算法更加准确地理解热度分布，提

升分类效果。将来能以更大的粒度管理热度，如根据负载 I/O 大小设置为 64 KB，从而减少空间成本，统计发现一个段的数据热度一致时，可以使用段首位置表示整段数据热度并在固定位置标志，段上其他逻辑块不用再记录热度元数据以进一步减少空间开销。此外计算一个块在文件系统中属于哪个类是依次比较每个质心，选择距离最近的类，该过程的时间开销也可以优化。

3.5 本章小结

专为闪存设计的 F2FS 已经广泛使用，然而 F2FS 需要段清理回收无效块，空闲空间碎片增加段清理开销。F2FS 中温数据至少占比 80%，但不同温数据块间热度存在差异，使用一个 log 写入造成不同热度的数据混合存储，加剧了空闲空间碎片化程度，加重了段清理开销，原因在于一个段上更快失效的热数据增加了空闲空间碎片，同时较冷的数据仍然有效，增加了迁移开销。针对该挑战，提出了细粒度区分温数据热度并多日志延迟写不同热度的数据及优化段清理策略。本章的主要贡献有：

(1) 根据工作负载的读写特征，选择性使用文件块更新距离、最近使用距离和读次数准确定义热度，为了准确区分数据热度，使用 K-means 聚类算法识别热度，使用 Mini Batch K-means 采样减少文件块级区分数据热度的聚类计算成本，为了适应负载数据热度的变化，基于文件块更新距离的敏感性变化，追踪其变化次数感知 I/O 模式热度变化以动态识别数据热度。

(2) 基于细粒度区分的温数据热度，设计了多日志延迟写入技术维护写性能，优化了清理对象的选择和释放进一步减少段清理开销，提出了热度元数据缓存用于管理热度元数据。

(3) 在实际工作平台上采用热度倾斜不同的工作负载充分地评估了 M2H 策略，实验结果表明，与传统 F2FS 相比，M2H_GPU 的段清理次数减少了 54.41%~97.93%，在文件系统空间使用率达到 90% 时，M2H_GPU 的负载工作时性能比传统 F2FS 提高了 13.52%~154.19%。与 M2H 相比，M2H_MBK 和 M2H_GPU 聚类发生时 CPU 使用率平均分别降低 34.04% 和 17.05%。

4 空闲空间碎片感知的垃圾回收策略

移动设备配置的电池容量有限，导致智能手机一天内多次充电，非常影响用户使用体验，智能手机的续航问题一直是行业痛点，移动设备各个组件活动的能耗需要尽量降低。移动设备存储系统的能耗与屏幕显示、网络连接的能耗相当，约占 30%。现有移动设备的能耗研究主要关注构建移动设备能耗模型、寻找能耗瓶颈和降低能耗的解决方案，缺乏面向日志结构文件系统的能耗研究和其减少能耗的策略。

文件碎片和空闲空间碎片会影响文件系统 I/O 性能，进一步分析碎片对日志结构文件系统能耗的影响，发现空闲空间碎片数量增加导致 CPU、UFS 模块完成 I/O 任务的能耗增加。F2FS 使用 GC 减少空闲空间碎片，而一次后台 GC 的能耗比较大，且后台 GC 减少空闲空间碎片的效果有限，基于数据分析评估多大的空闲空间是空闲空间碎片，采用空闲空间碎片系数快速衡量空闲空间碎片化程度，提出 FAGC (free space Fragmentation Aware GC) 策略，感知空闲空间碎片化程度，优化清理对象的选择算法和迁移有效块的写方式，以提高每次垃圾回收减少空闲空间碎片的效率，达到无效空间的高能效回收目的，并减少垃圾回收开销与能耗。设计实验测试智能手机日志结构文件系统在不同场景完成 I/O 任务的能耗，分析能耗结果，研究空闲空间碎片数量对能耗的影响和 GC 减少空闲空间碎片的效果，提出 FAGC 策略并在实际平台评估性能，与其他优化方法对比并分析实验结果。

4.1 碎片问题对能耗的影响

现有研究工作缺乏对移动设备日志结构文件系统能耗的研究，本章使用华为提供的能耗测试环境，构造不同空间使用率和空闲空间碎片数量的三种状态，分析文件碎片与空闲空间碎片数量对系统性能的影响，比较并分析顺序读写与随机读写常规文件的能耗，以及测试分析文件碎片对顺序读性能和能耗的影响。

4.1.1 能耗分析实验环境

为了分析文件系统空间使用率和空闲空间碎片数量对系统 I/O 任务时延和能耗的影响，构造了三种日志结构文件系统状态，分别是初始、重载和末端碎片化状态，

华中科技大学博士学位论文

表 4.1 描述了三种状态的空间使用率和构造方法。为了快速构造末端碎片化状态，使用 fio 模拟快速产生文件碎片和空闲空间碎片的过程，依次用 1 GB、2 MB、4 KB 和 8 KB 大小的文件写满文件系统，其中写满 2 MB、4 KB 和 8 KB 的文件后间隔删除。

表 4.1 日志结构文件系统的三种状态

状态	系统空间使用率	剩余可用空间	构造方法描述
初始	2%	216 GB	刚刚刷机的初始化系统
重载	100%	1.5 GB	在初始状态的系统上使用 fio 随机写一个 2 GB 大小的文件，设置直接同步 I/O 模式，不断复制该文件直至剩余可用空间为 1.5 GB，此过程中无删除操作
末端碎片化	100%	1.5 GB	在初始状态的系统上使用 fio 模拟快速老化过程，使文件系统产生大量的空闲空间碎片，保证剩余可用空间为 1.5 GB

初始、重载和末端碎片化三种状态的空闲空间碎片数量分布如图 4.1 所示，图 4.1 中第一级至第六级的碎片大小沿用表 2.1 中分类，考虑到此处存储空间比较大，修改第七级碎片的大小范围是 128~256 KB（包含 256 KB），增加第八级碎片，其大小范围是 256~512 KB（包含 512 KB），增加第九级碎片，为连续空间大小大于 512 KB 的碎片。从图 4.1 中可以看到，尽管重载状态文件系统空间使用率远高于初始状态（100% 与 2%），但重载状态的空闲空间碎片总数只是初始状态的 2.01 倍，原因在于构造重载状态的过程中只复制 2 GB 大文件而没有删除操作。

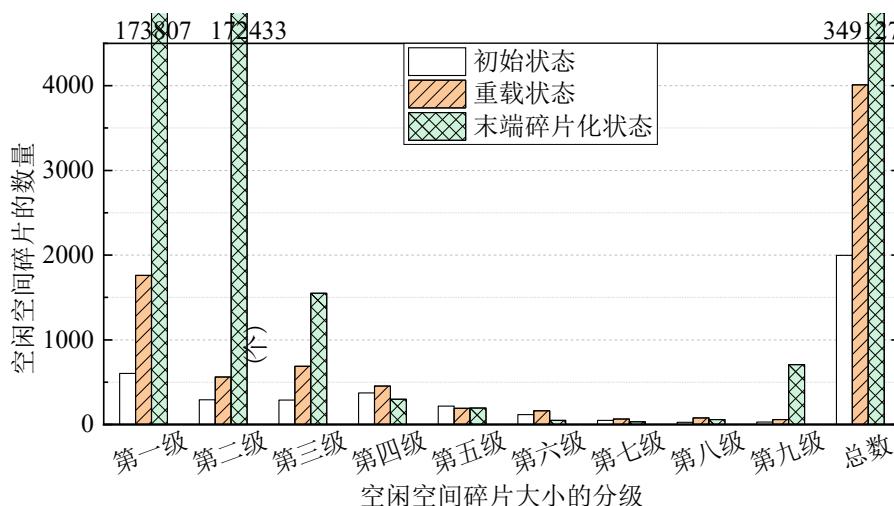


图 4.1 三种状态下的空闲空间碎片数量分布

华中科技大学博士学位论文

构造末端碎片化状态的过程中大量创建和删除 2 MB、8 KB 和 4 KB 大小的文件，与重载状态相比，末端碎片化状态的碎片数量大幅增加，根据表 4.1 和图 4.1，虽然末端碎片化状态的文件系统空间使用率与重载状态的一样，但末端碎片化状态的空闲空间碎片总数是重载状态的 87.02 倍，从图 4.1 看到末端碎片化状态的第一、第二级空闲空间碎片数量比重载状态明显增加，说明末端碎片化状态有大量 4 KB、8 KB 大小的空闲空间碎片，空闲空间碎片化程度非常严重。

采用华为提供的能耗测试系统分析能耗，该系统由供电电源、测试平台硬件能耗测试主板（简称能耗板）、能耗检测板、主机端测试软件（包括应用界面、配置文件、驱动等）、转换表格等组成，其中能耗板基于麒麟 9000¹采用 Android 10 和 Linux 内核 4.14.116 版本。能耗测试系统的工作示意图如图 4.2 所示，图 4.2 左侧是实验平台硬件组成部分，右侧是主机端软件部分，基于麒麟 9000 的能耗板插入三块能耗检测板，使用两根数据线与主机间传输能耗数据、接收 adb 命令。主要用该测试系统监测电流，使用供电电源为系统提供 4.2 伏特的稳定电压，能耗检测板在系统供电线路中串联检测电阻，通过测量电阻两端电压计算得到电流，该能耗测试系统能分别计算智能手机各个功能模块的耗电情况。

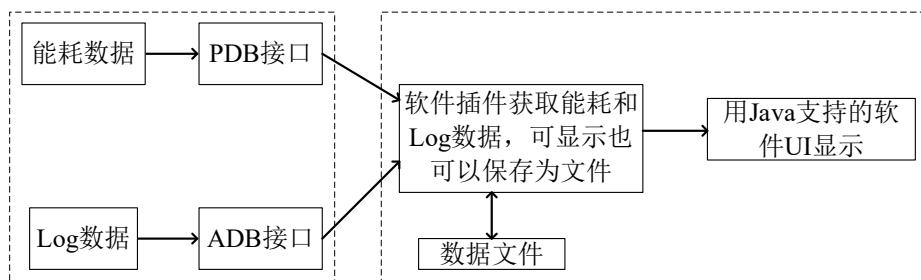


图 4.2 能耗测试系统的工作示意图

从测试软件导出电流数据并分析完成 I/O 任务的运行时间，根据 $W=UIT$ 计算能耗，U 表示电压，I 表示电流，T 表示 I/O 任务运行时间。经转换表格转换后可以得到各个模块的工作能耗，在封闭实验室中不可连接网络，第 5 代移动通信技术（5th Generation Mobile Communication Technology，5G）网络、Wifi 和蓝牙等模块的能耗

¹ <https://www.hisilicon.com/cn/products/Kirin/Kirin-flagship-chips/Kirin-9000>

很小，此外由于测试负载主要是存储系统 I/O 任务，没有复杂的计算，图形处理器（Graphics Processing Unit, GPU）和神经网络处理单元（Neural-Network Processing Unit, NPU）的能耗同样很小，最终选择观察 CPU、双倍速率同步动态随机存取存储器（Double-Data-Rate Synchronous Dynamic Random Access Memory, DDR）和通用闪存存储（Universal Flash Storage, UFS）三个模块完成 I/O 任务的能耗。

4.1.2 读写能耗分析

文件碎片使得顺序读写的 I/O 变成随机的，在连续空闲空间耗尽时，F2FS 使用线程日志方式覆盖写空闲空间碎片，造成新入文件的碎片化。设计 I/O 密集型负载比较顺序读写和随机读写常规文件的时延和能耗，在测试每次 I/O 任务前重启系统以避免缓存数据的影响，每顺序读写、随机读写 1 MB 数据的能耗和读写整个文件花费的时间如图 4.3 所示。三种状态顺序读、顺序写、随机读和随机写 1 MB 数据的整机能耗平均分别是 2.84 mJ、3.4 mJ、33.76 mJ 和 34.81 mJ，三种状态平均的随机读时间是顺序读时间的 17.57 倍，随机写时间是顺序写时间的 10.81 倍，与现有研究工作^[16]的结论一致，与顺序读写相比，由于 I/O 请求不连续，随机读写的 CPU 能耗、DDR 能耗和 UFS 能耗都明显增加。

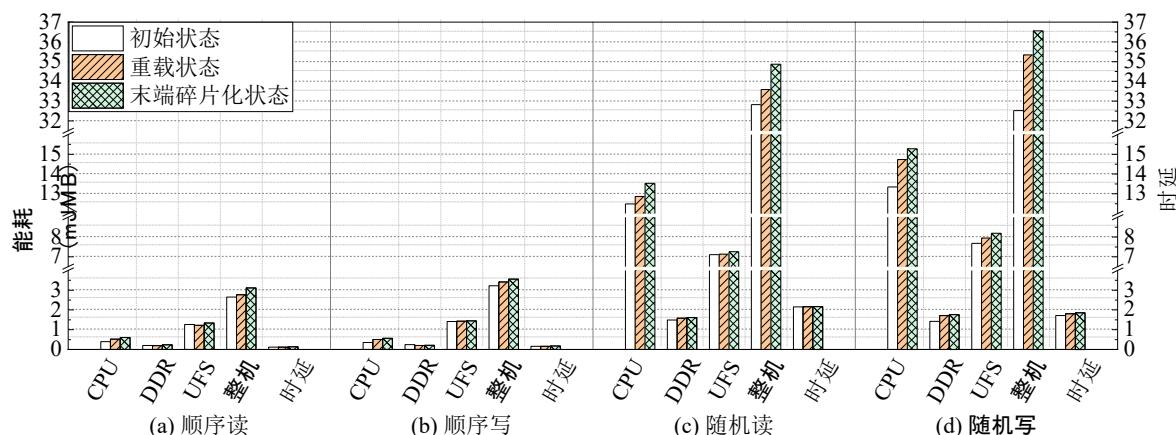


图 4.3 随机读写与顺序读写文件的时延与能耗

三种状态之间，末端碎片化状态的时延和能耗是最高的，原因在于末端碎片化状态存在大量的文件碎片和空闲空间碎片。该 I/O 密集型负载在测试时熄屏、没有连接网络，CPU、DDR 和 UFS 三个模块的能耗占整机能耗的比例非常大，顺序读写的

UFS 能耗占比最高,三种状态平均的顺序读、写的 UFS 能耗分别占整机能耗的 44.77%、42.07%,随机读写的 CPU 能耗占比最高,随机读、写的 CPU 能耗占比分别是 38.49%、41.53%。

分析发现运行时间的小幅增加,导致系统能耗的大幅增加,需要减少完成读写任务和应用活动的运行时间,即使是小幅度的减少,也能有效地减少能耗。分析发现读写数据的 CPU 能耗随着空闲空间碎片数量明显增加,需要减少空闲空间碎片数量,挖掘空闲空间碎片数量导致 CPU 能耗增加的关键 I/O 并优化。分析发现随着文件系统空间使用率和空闲空间碎片数量增加,UFS 能耗也明显增加,顺序读写的 UFS 能耗占比大于随机读写的 UFS 能耗占比,需要研究 UFS 能耗增加的根本原因与优化方法。综上需要减少文件碎片和空闲空间碎片数量,减少运行时间,降低顺序读写的 UFS 能耗,减少随机读写的 CPU 能耗。

4.1.3 顺序读碎片文件能耗分析

考虑到碎片数量对顺序读性能的影响最大,分别在三种状态比较顺序读连续文件和碎片文件的能耗以分析碎片数量对能耗的影响,连续文件和碎片文件的构造方法与第 2.4.2 节一样,测试时删除文件和重启系统以避免缓存数据的影响。图 4.4 展示了每顺序读 1 MB 数据的能耗和读整个文件的时延,对于顺序读 1 MB 连续文件和碎片文件数据,三种状态的整机能耗平均分别是 2.53 mJ 和 8.58 mJ,后者是前者的 3.39 倍。与顺序读连续文件相比,三种状态平均的顺序读碎片文件时延增加 254.38%,CPU 能耗增加 371.92%,DDR 能耗增加 128.65%,UFS 能耗增加 167.72%,顺序读碎片文件的时延和各模块的能耗都明显增加。发现顺序读碎片文件的 CPU 模块能耗增幅最大,因为文件碎片分割 I/O 请求,需要重点关注 CPU 能耗因文件碎片增加的问题。实际上顺序读碎片文件的时间比随机读该文件的时间少,与已有研究工作^[54]不同,不是简单使用随机读该文件的性能模拟该文件存在文件碎片的读性能。

对于顺序读连续文件,与初始状态和重载状态相比,末端碎片化状态的顺序读时延和能耗都显著增加,说明文件系统快写满时,F2FS 采用线程日志方式写,空闲空间碎片数量的增加会严重影响顺序读的时延和能耗。对于顺序读碎片文件,重载状态各项数据比初始状态稍微减少,一方面碎片文件自身的文件碎片数量比较多,使得在

初始状态时顺序读碎片文件的时延和能耗就比较大；另一方面，构造重载状态的过程中没有删除文件，产生的空闲空间碎片数量比较少，仅是文件系统空间使用率的增加，这表明比起文件系统空间使用率的增加，文件系统的性能与碎片化程度等系统使用情况关系更密切，这与现有研究工作的结论一致^[64]。

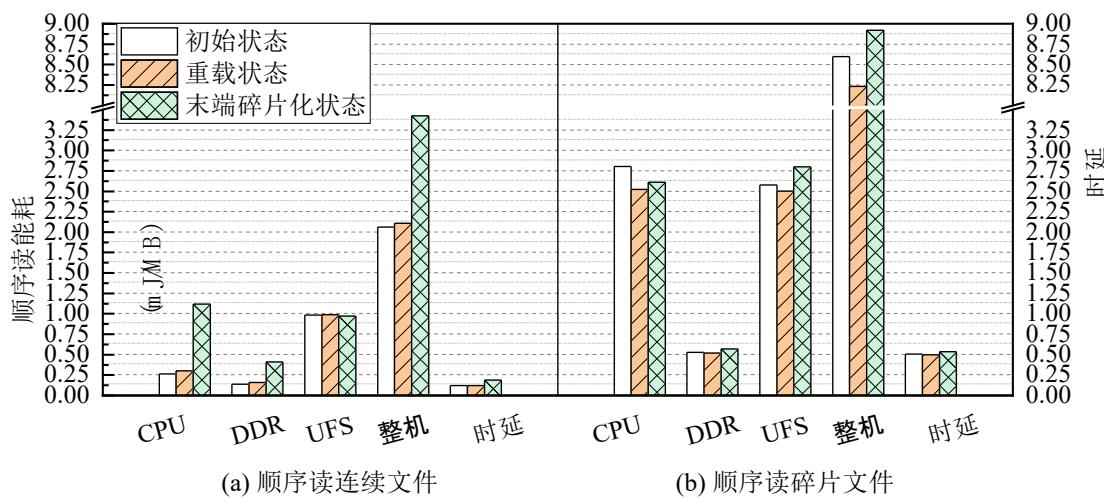


图 4.4 顺序读连续文件与碎片文件的时延与能耗

顺序读碎片文件的能耗明显高于顺序读连续文件，说明文件碎片非常影响系统能耗，末端碎片化状态的顺序读能耗相比初始和重载状态明显增加，说明空闲空间碎片同样影响系统性能。这是由于文件碎片和空闲空间碎片使连续 I/O 变得碎片，导致 I/O 次数增加，需要有效减少文件碎片和空闲空间碎片。第 2 章的 ARST 策略能有效减少文件碎片，但还需要有效缓解空闲空间碎片化的技术。

4.2 垃圾回收能耗分析及挑战

F2FS 依赖垃圾回收减少空闲空间碎片，获得空闲段，测试前台垃圾回收与后台垃圾回收的能耗，并衡量垃圾回收减少空闲空间碎片的效果。

4.2.1 前台与后台垃圾回收的能耗

通过修改触发时间间隔实现不同触发频率的后台 GC，完成 GC 后系统调用同步命令，以初始状态触发时间间隔为 120 s 的一次后台 GC 产生的整机能耗为基准，记录 120 s 内不同触发频率的后台 GC 因为完成 GC 增加的能耗，该时间段内智能手机

没有运行其他 I/O 任务，没有连接网络和点亮屏幕。120 s 内不同触发频率后台 GC 的整机能耗如图 4.5 所示，触发后台 GC 的时间间隔分别是 1000 ms、100 ms 和 10 ms。触发时间间隔为 1000 ms 的后台 GC 整机能耗比 100 ms 的平均减少 17.5%，比 10 ms 的平均减少 55.46%，原因在于触发时间间隔为 1000 ms 时触发的 GC 次数较少，需要减少不必要的 GC。从初始状态到末端碎片化状态，还观察到随着空间使用率和空闲空间碎片数量增加，UFS 模块能耗和整机能耗呈现明显上升趋势，相比重载状态，末端碎片化状态整机能耗明显增加，表明需要减少空闲空间碎片数量。

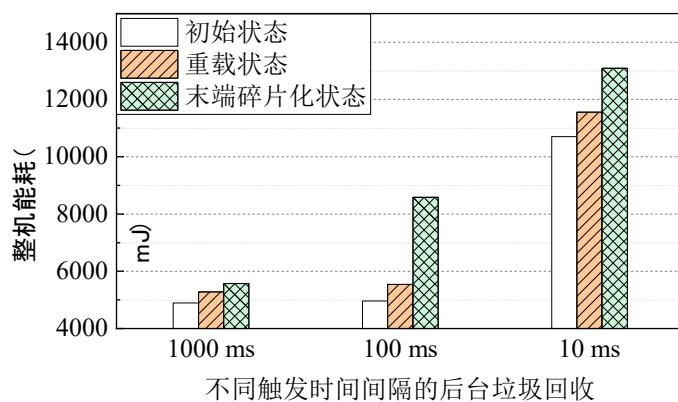


图 4.5 不同触发时间间隔的后台垃圾回收在 120 秒内的整机能耗

通过修改内核主动触发前台 GC，记录主动触发 10 次前台 GC 的时间和能耗，连续触发并完成 10 次前台 GC 花费的时间不同，初始、重载和末端碎片化状态分别花费 4.619 s、4.923 s 和 5.389 s，比较重载状态和末端碎片化状态，空闲空间碎片数量的增加导致完成 GC 的时间和能耗明显增加。前台 GC 和不同触发频率的后台 GC 完成一次 GC 的能耗如图 4.6 所示，三种状态平均的前台 GC、触发时间间隔为 1000 ms、100 ms 和 10 ms 的后台 GC 完成一次 GC 的能耗分别是 438.95 mJ、123.02 mJ、134.67 mJ 和 184.3 mJ。前台 GC 的 CPU 能耗和 UFS 能耗都比较大，完成一次前台 GC 的能耗约能完成四次触发时间间隔为 1000 ms 的后台 GC。触发时间间隔为 10 ms 的后台 GC 完成一次 GC 的能耗比触发时间间隔为 100 ms 的后台 GC 高 36.85%，主要原因在于前者 GC 次数和 CPU 模块能耗明显增加，需要减少 GC 次数和 CPU 模块能耗。分析发现前台 GC 和后台 GC 的能耗都比较大，在保证文件系统正常工作的前提下需要懒惰地触发 GC，尽可能减少前台 GC 和后台 GC 的次数，并优化 GC 过程，

减少因为 GC 迁移有效块增加的读写数据能耗。

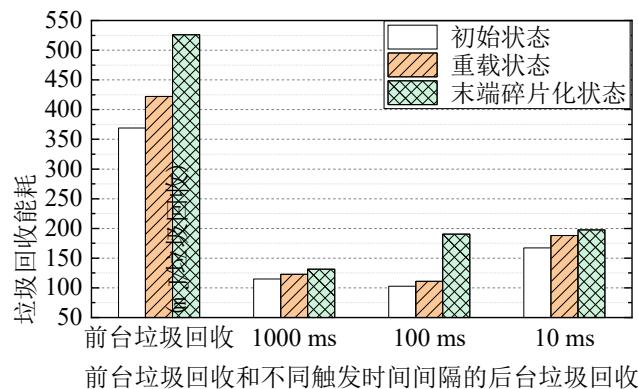


图 4.6 一次前台垃圾回收或后台垃圾回收的能耗

4.2.2 垃圾回收减少空闲空间碎片的效果

F2FS 通过垃圾回收整理空闲空间碎片获得空闲空间，测试分析能耗成本较大的垃圾回收减少空闲空间碎片的效果。末端碎片化状态存在大量的空闲空间碎片，随着时间垃圾回收次数增加，不同大小的空闲空间碎片数量变化如图 4.7 所示，图 4.7 中前台表示前台 GC，由系统根据需要触发，其触发频率比图 4.6 中前台 GC 低，后台表示后台 GC，其触发时间间隔为 10 ms，考虑到后台 GC 触发次数比前台 GC 少，记录 800 s 内的后台 GC，图 4.7 展示了经过 GC 的空闲空间碎片数量变化和对应的 GC 触发次数，可以计算对应 GC 次数所需的能耗。

垃圾回收有效减少空闲空间碎片数量的情形是 2 MB 大小的碎片（第九级）数量增加，小的空闲空间碎片（第一、二、三级）数量和总数减少。从图 4.7 中看到，随着前台 GC 次数的增加，系统中第一级、第二级、第三级和第八级的空闲空间碎片数量和碎片总数量都明显减少，第九级的空闲空间碎片数量增加。随着后台 GC 次数的增加，系统中第九级的碎片数量反而减少，其他大小的碎片数量以及碎片总数量都没有明显的变化，根据空闲空间碎片数量的变化发现后台 GC 减少空闲空间碎片数量的效果有限。基于图 4.7 (b) GC 次数和图 4.6 中末端碎片化状态一次 GC 的能耗计算文件系统因为 GC 消耗的能量理论值，600 s 内前台 GC 的能耗是 261.36 J，800 s 内后台 GC 的能耗是 73.68 J，前台 GC 的能耗比较大，减少空闲空间碎片数的效果也比较好，800 s 内后台 GC 的能耗并不小，而其减少空闲空间碎片的效果不佳。

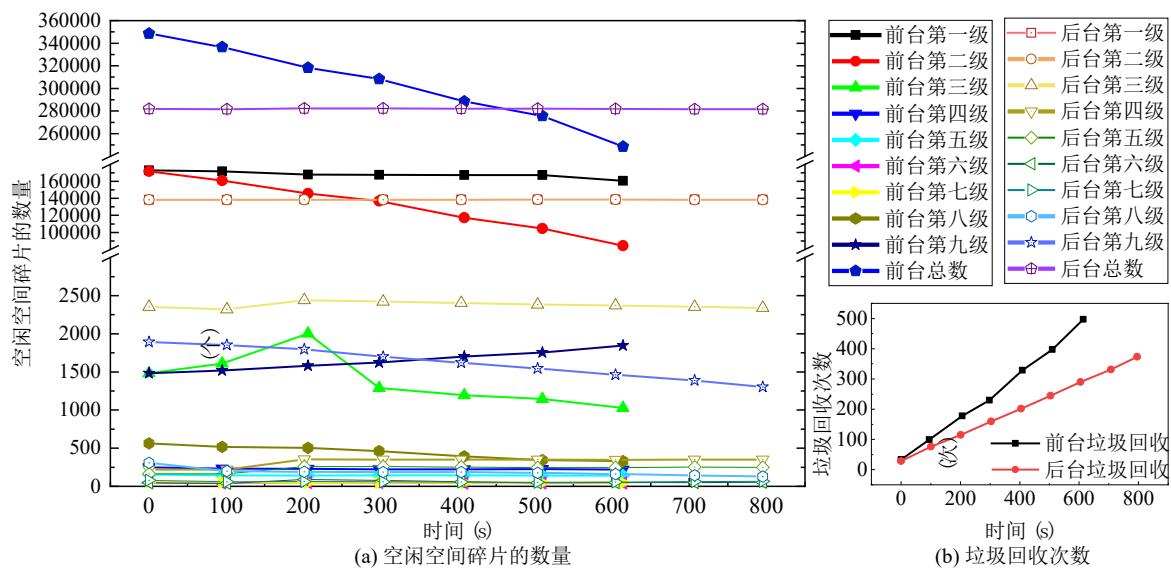


图 4.7 空闲空间碎片数量随垃圾回收次数的变化

考虑到 GC 能获得 2 MB 的空闲空间段，除了分析不同大小空闲空间碎片的数量变化，还衡量不同大小区间的空闲空间片段容量变化。不同大小的空闲空间片段容量变化如表 4.2 所示，733.79↓表示容量减少了 733.79 MB，0.84↑表示容量增加了 0.84 MB，其他依此类推，可以看到相比后台 GC，前台 GC 更有效地减少小于 128 KB 的空闲空间碎片容量，后台 GC 更有效地减少大小在 128 KB 到 1 MB 范围的空闲空间片段容量。第九级碎片为连续空间大小大于 512 KB 的碎片，图 4.7 中后台第九级空闲空间碎片数量减少，说明后台 GC 减少的 512 KB 到 1 MB 大小的碎片数量远多于增加的 2 MB 及以上的片段数量，导致后台 GC 第九级碎片数量减少，从图 4.7 看到前台第九级空闲空间碎片数量增加，结合表 4.2 说明前台 GC 减少的 512 KB 到 2 MB 大小的碎片数量少于增加的 2 MB 及以上的片段数量，导致前台 GC 第九级碎片数量增加。前台 GC 减少比较小的空闲空间碎片，后台 GC 减少比较大的空闲空间片段，小于 128 KB 的空闲空间碎片容量反而增加了 0.84 MB，后台 GC 减少较小的空闲空间碎片数量和容量的效果不佳，需要重新评估一次后台 GC 的成本和收益。

表 4.2 不同大小的空闲空间片段经过垃圾回收的容量变化 (MB)

空闲空间片段大小	[0, 128 KB)	[128 KB, 1 MB)	[1 MB, 2 MB)	2 MB 及以上
前台垃圾回收	733.79↓	165.94↓	7.8↓	907.53↑
后台垃圾回收	0.84↑	681.11↓	13.83↓	694.1↑

与顺序读写或随机读写 1 MB 数据的能耗相比，不同触发频率的后台 GC 的能耗不容轻视。观察到图 4.6 中能耗最少的一次 GC 也消耗超过 100 mJ 的能量，与随机读或随机写 1 MB 数据消耗的能量相比，其所消耗的能量能够完成随机读或随机写约 3 MB 的数据。最少的一次前台 GC 和后台 GC 的能耗分别是 368.85 mJ 和为 102.79 mJ，中高端智能手机的电池容量是 3000 mAh，手机电池电压取工业界常用的 3.8 V，该电池充满电能提供 41040 J 能量，假设手机充满电原本能待机 24 小时，则每小时消耗 475 mJ 能量，如果每小时多做一次前台 GC，则待机时间缩短 10.49 小时，如果每小时多做一次后台 GC，则待机时间缩短 4.27 小时，因此一次 GC 的能耗是比较大的，在保证系统正常工作的前提下要尽可能减少 GC 的次数。一次前台 GC 的能耗非常大，应尽量减少触发前台 GC，F2FS 中前台 GC 只在空闲空间不足时才触发。后台 GC 的能耗也比较大，F2FS 中定时唤醒线程触发后台 GC，然而能耗较大的后台 GC 减少空闲空间碎片的效果不佳，导致一次后台 GC 能耗较大，获得连续空闲空间的收益较小。

4.2.3 减少能耗面临的挑战

通常有两种减少能耗的方法，一种是直接减少能耗，另一种是通过提升单位能耗的收益来提高能耗效率，已有大量研究工作^{[107][108]}致力于直接减少存储系统的能耗，而提高单位能耗收益的方法较少。通过实验结果分析发现空闲空间碎片数量增加是 CPU 和 UFS 模块能耗增加的重要原因，F2FS 现有减少空闲空间碎片的方法是垃圾回收，然而能耗成本高昂的后台 GC 减少空闲空间碎片的效果有限，需要进一步提升后台 GC 减少空闲空间碎片的效率，旨在以相同或更少的 GC 次数回收更多数量的空闲空间碎片，实现无效空间的高能效回收。

当前优化空闲空间碎片的相关工作比较少，传统定义将完全连续的空闲空间视为无空闲空间碎片，没有明确指出多大的空闲空间是空闲空间碎片，此外也缺乏文件系统中衡量空闲空间碎片化程度的方法，F2FS 迫切需要以有限能耗高效率回收无效空间的方法。而现有垃圾回收策略在选择清理对象时仅考虑了有效块数量和段年龄，没有考虑空闲空间碎片化程度，可能导致迁移有效数据后获得的空闲空间收益并不大，如清理的段上有 1 个 1 MB 大小的空闲空间碎片与清理的段上有 256 个 4 KB 大

小的空闲空间碎片，都是得到 2 MB 大小的空闲段，选择后者作为清理对象能减少更多的空闲空间碎片数量，垃圾回收的收益更大。为了解决上述问题，提出了基于空闲空间碎片化程度的 FAGC 策略。

4.3 碎片感知垃圾回收策略

为了保证日志结构文件系统正常工作，必须触发垃圾回收以获得空闲空间。文件碎片使得连续的读写请求变得不连续，造成原可以一个 I/O 请求完成的顺序读写数据变成随机读写，文件碎片和空闲空间碎片之间相互影响，观察到随机读写文件的时延和能耗远高于顺序读写相同大小文件的时间和能耗，当文件碎片和空闲空间碎片数量增加时，顺序读文件的时延和能耗大幅增加。综上碎片数量的增加导致 I/O 任务运行时间和系统能耗都增加，解决碎片化问题的方法，除了第 2 章展示的空间预留技术有效减少文件碎片外，还有在空闲空间碎片化程度严重时 F2FS 通过 GC 整理空闲空间碎片，观察到后台 GC 的能耗比较大且后台 GC 减少空闲空间碎片的效果不佳。基于应用行为特征和空闲空间碎片数量及大小，重新评估空闲空间碎片大小的阈值，衡量段的空闲空间碎片化程度，提出感知空闲空间碎片化程度的 GC 策略。

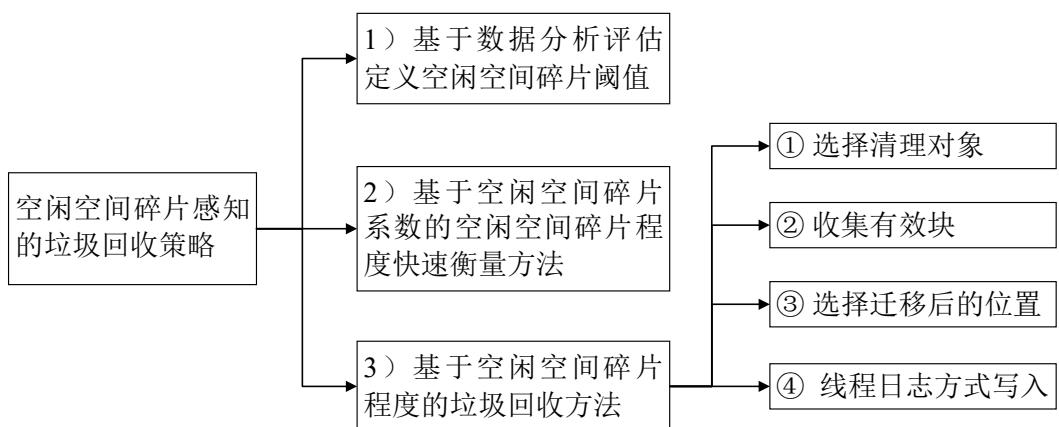


图 4.8 空闲空间碎片感知垃圾回收策略的构成示意图

如图 4.8 所示，提出空闲空间碎片感知的垃圾回收策略 FAGC 以提升单位能耗获得空闲空间的收益，该策略主要包括三个部分，明确多大的无效空间或空闲空间是空闲空间碎片，快速衡量段上空闲空间碎片化程度，基于空闲空间碎片化程度回收无效空间，以提升能耗效率。

4.3.1 空闲空间碎片阈值的数据分析

分析日志结构文件系统能耗发现，文件碎片和空闲空间碎片增加 I/O 任务读写次数，造成 CPU 模块、UFS 模块完成 I/O 任务的能耗明显增加，第 2 章提出的 ARST 能有效减少文件碎片，还缺乏有效减少空闲空间碎片的策略。F2FS 现有减少空闲空间碎片的方法是垃圾回收，而系统经常触发的后台 GC 需要迁移有效块，读写开销大，能耗开销大，且后台 GC 减少空闲空间碎片的效果有限，一方面减少的空闲空间碎片数量有限，另一方面整理大小范围在 128 KB 到 1 MB 之间的空闲空间片段，获得 2 MB 及以上连续空闲空间的收益比整理更小的空闲空间碎片获得连续空闲空间的收益小。

完成一次 GC 的能耗与清理对象的选择、迁移被清理段上有效块的读写操作以及释放被清理段时检查点同步数据的 I/O 有关。如何区分空闲空间与空闲空间碎片成为选择垃圾回收对象的关键问题，小的空闲空间片段是空闲空间碎片，比较大的空闲空间片段能够满足大部分的 I/O 请求，可视为可用空闲空间，不能当作空闲空间碎片。按照传统的碎片定义，只有空闲空间完全连续才视为没有碎片，为保证空闲空间连续会导致日志结构文件系统的 GC 比较激进，造成 GC 次数增加、GC 开销增大。基于不影响用户性能体验的前提，需要定义合适的空闲空间碎片大小，并在 GC 时考虑空闲空间碎片化程度，优化碎片整理策略。

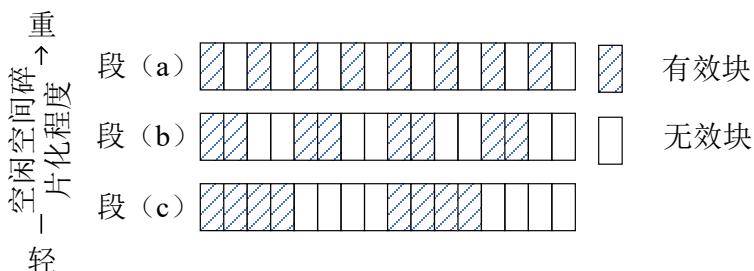


图 4.9 不同空闲空间碎片化程度的数据段示意图

图 4.9 展示了三种段中数据分布状态，段中有效块数相同，但空闲空间碎片化程度不同。图 4.9 (a)、(b)、(c) 这三个段有相同数量的有效块，无论哪一个段被选作清理对象，迁移成本皆为迁移 8 个有效块，但考虑未迁移前的情况，假如线程日志写入一个 16 KB 大小的文件，则在段 (a) 中被分割为四个 I/O 请求，在段 (b) 中被分

割为两个 I/O 请求，在段 (c) 中只用一个 I/O 请求，原因在于段 (c) 空闲空间碎片化程度最轻，三者中选择段 (a) 作为清理对象有最大的垃圾回收收益。若段 (c) 中连续的空闲空间足够大，可满足大部分文件读写数据只用一个 I/O 请求，而根据传统定义段 (c) 中存在空闲空间碎片，可作为清理对象，实际上选择段 (c) 进行段清理是没有必要的，而现有 F2FS 段清理策略有可能选择段 (c) 进行段清理，因此迫切需要基于数据分析定义合适的空闲空间碎片大小，选择不清理段 (c) 这样的段。

基于数据分析将小于 128 KB 的空闲空间定义为空闲空间碎片，选择 128 KB 作为阈值，一方面是因为 Linux 内核将 128 KB 定为预读数据默认大小，对于一个 128 KB 大小的空闲空间，假如预读的 128 KB 数据存放在该空间，用一个 I/O 请求就可完成预读，不会被分割成多个 I/O 请求。另一方面，分析应用启动的 trace，发现有大量 64 KB 大小的同步读和异步读请求，使用 128 KB 的空闲空间能连续存放读请求的数据。用 MobiBench 抓取普通用户一天中使用不同应用的 trace，分析用户一天中不同大小数据块的读写次数，其中小于等于 128 KB 的读 I/O 占 85%，小于等于 128 KB 的写 I/O 占 87%，统计应用中比较大的读写请求，百度地图仅有 2% 的请求是 512 KB 大小，图库、QQ 和淘宝中 128 KB 的请求分别占 4%、2% 和 6%。考虑到 128 KB 的空闲空间能够满足大多数 I/O 请求，将 128 KB 设置为空闲空间碎片的阈值。

从表 4.2 看到后台 GC 主要减少范围在 128 KB 到 1 MB 的空闲空间片段，而该大小范围的空闲空间片段可以满足大部分 I/O 请求，不属于空闲空间碎片。后台 GC 选择清理对象时需要更具针对性地减少小于 128 KB 的空闲空间碎片，而对于 128 KB 到 2 MB 大小的空闲空间片段所在的段，因为该大小范围内的空闲空间可以满足大部分 I/O 请求，在同样的迁移成本下不考虑作为清理对象。

4.3.2 空闲空间碎片化程度的衡量

经过数据分析将小于 128 KB 的空闲空间定义为空闲空间碎片，触发 GC 时衡量空闲空间碎片化程度聚焦于一个段内。段信息表中的有效位图用 0 表示 4 KB 大小的逻辑块空闲，用 1 表示该逻辑块上存放有效数据，基于一个段的有效位图设计空闲空间碎片系数 (factor, f) 衡量该段的空闲空间碎片化程度，计算一个段的空闲空间碎片系数 f 的过程如算法 4.1 所示。

华中科技大学博士学位论文

算法 4.1 计算空闲空间碎片系数的算法

输入: 该段的有效位图, *bitmap*;

空闲空间碎片系数并赋初值, $f \leftarrow 0$;

连续的空闲块的数量并赋初值, $X \leftarrow 0$;

空闲空间碎片大小的阈值 (free space fragmentation threshold, f_th), $f_th \leftarrow 32$

输出: 某个段的空闲空间碎片系数, *f*;

1: 将该段的有效位图, *bitmap*, 由十六进制转换为二进制;

2: **while** 没有判断完该段上的 bit **do**

3: 统计一块连续的空闲空间上空闲块的数量 (即连续的 0 的个数), *X*;

4: **if** $X \geq 32$ **then**

5: 不记作空闲空间碎片;

6: $X \leftarrow 0$, 用于统计另一块连续空闲空间大小;

7: **else**

8: 对于连续的 *X* 个 0, $f \leftarrow f + \frac{1}{X} * 126 * 127$;

9: $X \leftarrow 0$, 用于统计另一块连续空闲空间的大小;

10: **end if**

11: **end while**

基于数据分析将空闲空间碎片阈值定为 128 KB, 除以每个块的大小 4 KB, 赋值为 32, 在算法 4.1 中, ① 做判断, 大于 128 KB 的空闲空间 (位图中有大于等于 32 个连续的零) 则不视作碎片 (Line 3-6), ② 计算 *f*, 根据空闲空间碎片大小, 在位图中有几个连续的零, 记作 *X*, 一个段中有 *n* 个碎片, 对 *n* 个 $1/X$ 求和, 则 *f* 越大, 空闲空间越离散, 空闲空间碎片化程度越严重 (Line 7-11)。因为内核中不建议使用浮点数运算, 所以在计算 *f* 时放大 16002 ($126 * 127$) 倍, 使得 *f* 是个正整数, 并且 1 到 127 KB 的空闲空间碎片分别有唯一的 *f* 值。乘以 127 是为了取整, 放大 126 倍是为了快速区分不同大小的空闲空间碎片, 如一个 127 KB 大小的空闲空间碎片的 *f* 是 $126(\frac{1}{127} * 126 * 127)$, 一个 126 KB 大小的空闲空间碎片的 *f* 是 $127(\frac{1}{126} * 126 * 127)$, 空闲空间碎片系数 *f* 不仅可以用来计算 F2FS 的空闲空间碎片化程度, 还可以用于计算其他文件系统的空闲空间碎片化程度。用式 (4.1) 计算 *f* 值, 其中 X_i 表示第 *i* 个碎片的大小, 在 F2FS 中为 X_i 个 4 KB 的空闲空间。*f* 取值为 [0, 4096512], 最糟糕的

4096512 的情况是如图 4.9 (a) 所示的一共有 256 个空闲空间碎片，一个有效块接一个空闲空间碎片地交叉存储，在一个段中有 256 个不连续的 4 KB 大小的碎片。

$$f = \sum_{i=1}^n \frac{1}{X_i} * 126 * 127 \quad (4.1)$$

假设图 4.9 中一个段的大小是 64 KB，按照算法 4.1，则段 (a) 的空闲空间碎片系数为 $\frac{1}{1} * 126 * 127 * 8$ ，即 128016；段 (b) 的空闲空间碎片系数为 $\frac{1}{2} * 126 * 127 * 4$ ，即 32004；(c) 的空闲空间碎片系数为 $\frac{1}{4} * 126 * 127 * 2$ ，即 8001。空闲空间碎片化程度最严重的段 (a) 的空闲空间碎片系数最大，根据三个段的空闲空间碎片系数能快速区分空闲空间碎片化程度。

4.3.3 垃圾回收策略

基于段的空闲空间碎片系数，FAGC 进一步筛选能做垃圾回收的段，无需整理部分满足后台 GC 要求，但经过后台 GC 未能有效减少空闲空间碎片数量的段，以达到 FAGC 回收关键段的效果，关键段中的空闲空间碎片数量和分布非常影响文件系统性能。如算法 4.2 所示，设计使后台 GC 有效减少空闲空间碎片数量的方法，其中总的时间 total_time 为最大更新时间与最小更新时间的差值，参考传统后台 GC 中 age 的权重，算法 4.2 中 age 的权重 age_weight 设置为 40，空闲空间碎片系数 f 的权重 f_weight 设置为与 age 的权重一样，则 u 的权重是 20。age 越大，段的年龄越大，意味着段上数据越冷，u 越大，段迁移后获得的空闲空间越大，f 越大，段上的空闲空间碎片化程度越严重。算法 4.2 使用式 (4.2) 计算迁移成本，UINT_MAX 是最大的无符号整数数值，选择最小的 cost，使得以最小迁移成本获得连续空闲空间，age、u 和 f 在计算时都有做归一化处理 (Line 2-4)，使迁移成本不会受单一因素的影响。FAGC 策略主要分为两部分，第一部分是选择清理对象 (Line 1-12)，第二部分是选择目标段，迁移有效块，FAGC 选择清理对象时，考虑段的空闲空间碎片系数，选择空闲空间碎片化更严重的段，在迁移有效块时，尽量选择与被清理段年龄相近的目标段，与传统 F2FS 不同，不是等到空闲空间非常不足时才开始使用线程日志写方式，而是在迁移有效块到目标段时采用线程日志写方式。

$$\text{cost} = \text{UINT_MAX} - (\text{age} + f + u) \quad (4.2)$$

华中科技大学博士学位论文

算法 4.2 使后台 GC 有效减少空闲空间碎片的 FAGC 算法

输入: 文件系统中最大的更新时间, max_mtime ;

当前清理候选段的更新时间, $mtime$;

总的时间, $total_time$;

当前清理候选段上的有效块数量, $vblocks$;

当前清理候选段的空闲空间碎片系数, f ;

文件系统中最大的空闲空间碎片系数, max_f ;

文件系统中最小的空闲空间碎片系数, min_f ;

当前清理候选段的段号, $segno$;

age 的权重, age_weight ;

f 的权重, f_weight ;

迭代次数并赋初值, $iter \leftarrow 0$;

要查找的候选段的范围阈值, $dirty_threshold$;

当前选择的最小迁移成本, min_cost ;

当前选择的最大的 age , $oldest_age$;

输出: 有最小迁移开销的清理对象段号, min_segno ;

- 1: **while** $iter < dirty_threshold$ **do** /*当没有遍历完所有的能被清理的候选段*/
 - 2: 计算清理候选段的 age , $age \leftarrow 10000 * \frac{max_mtime - mtime}{total_time} * age_weight$;
 - 3: 计算迁移清理候选段后能得到的空闲空间, $u \leftarrow 10000 * \frac{512 - vblocks}{512} * (100 - age_weight - f3_weight)$;
 - 4: 调整清理候选段的 f , $f \leftarrow 10000 * \frac{max_f - f}{max_f - min_f} * f_weight$;
 - 5: 计算迁移成本, $cost \leftarrow \text{UINT_MAX} - (age + f + u)$;
 - 6: 增加迭代次数, $iter++$;
 - 7: **if** $cost < min_cost$ or ($cost == min_cost$ and $age > oldest_age$) **then**
 - 8: 更新最小迁移开销, $min_cost \leftarrow cost$;
 - 9: 更新最大 age , $oldest_age \leftarrow age$;
 - 10: 更新有最小迁移开销的清理对象段号, $min_segno \leftarrow segno$;
 - 11: **end if**
 - 12: **end while**
 - 13: 选择与被清理段有相同年龄的段做目标段;
 - 14: 通过线程日志写方式把被清理段的有效块迁移到目标段上;
-

使用线程日志方式直接写目标段上的无效区域，使得迁移有效块不会占用新的空闲段，且 FAGC 通过线程日志写方式迁移有效块时会逐个写入目标段上的无效块，尽量减少目标段的空闲空间碎片。FAGC 选择清理对象的算法与 F2FS 和 ATGC 的算法不同，FAGC 选择空闲空间碎片化较严重、更新时间较相近、段上有效块数量较少的段作为清理对象，只对非常影响系统性能的关键段做垃圾回收，旨在尽可能减少触发 GC，并使得每次 GC 尽可能多地减少空闲空间碎片。

4.4 性能评估与分析

使用与第 2.4.1 节中一样的实验平台，在 Hikey960 上进行实验。Li 等人提出了一种多级阈值同步写技术减轻文件碎片化和一种高检测频率后台段清理机制减少段清理开销^[69]，将其作为对比方案之一，下文称为 MWHFB。此外，选择能有效减少 GC 次数和有效块迁移数量的 F2FS 最新的垃圾回收策略 ATGC，以及为特定文件预留空间减少文件碎片的 ARST 策略作为实验的对比方案。ATGC 集成到了 Android 12.0 与 Hikey-kernel 5.4 版本的组合中¹，ATGC 策略、FAGC 策略部署在 Linux 内核 5.4.147 版本，在 Linux 内核 5.4.147 中用 830 行 C 语言代码实现了 FAGC 的原型，传统 F2FS、减少文件碎片的 ARST 策略、MWHFB 部署在 Linux 内核 4.9.76 版本。

4.4.1 空闲空间碎片的数量与容量

设计如下的实验，其中所有的顺序写操作是同步的， i 从 1 依次增加到 7000，
(1) 构造如下 7000 个脏段，先顺序写 1364 KB 的文件 $A[i]$ ，再顺序写大小为 684 KB 的文件 $B[i]$ ，循环 7000 次删除所有的文件 $B[i]$ ，(2) 再构造 7000 个脏段，先顺序写大小为 1364 KB 的文件 $C[i]$ ，再顺序写 684 KB 的文件 $D[i]$ ，循环 7000 次删除所有大小为 1364 KB 的文件 $C[i]$ ，(3) 再构造空闲空间碎片更加严重的 7000 个脏段，依次顺序写大小为 4 KB、12 KB、16 KB、32 KB、64 KB、128 KB、256 KB、512 KB 和 1024 KB 的文件 $E[i]$ 至 $M[i]$ ，循环 7000 次，删除所有的大小为 4 KB 的 $E[i]$ 、16 KB 的 $G[i]$ 、64 KB 的 $I[i]$ 、256 KB 的 $K[i]$ 和 1024 KB 的 $M[i]$ 文件，删除的文件大小

¹ <https://android.googlesource.com/kernel/common/+refs/heads/android12-5.4>

华中科技大学博士学位论文

之和为 1364 KB。通过删除空间大小总和相同而数量不同的空间片段构造了两种不同空闲空间碎片化程度的段，步骤（3）构造的脏段空闲空间碎片化程度更严重。经过密集的顺序写和 GC 操作后，GC 次数、有效块迁移数量、空闲空间碎片的数量和容量如表 4.3 所示，设计的实验中顺序写的文件不是预留文件，ARST 的实验结果与传统 F2FS 的一样，没有重复表述。

表 4.3 垃圾回收与空闲空间碎片情况

衡量指标	垃圾回收次数 (次)	有效块迁移数量 (千块)	空闲空间碎片数量 (个)	空闲空间碎片容量 (MB)
F2FS	1934	450.79	3187	64.65
MWHFB	2105	479.288	3010	62.29
ATGC	1314	296.956	3045	62.78
FAGC	335	77.434	3282	63.71

从表 4.3 中看到，FAGC 明显地减少了 GC 触发次数，比传统 F2FS、MWHFB 和 ATGC 分别减少 82.68%、84.09% 和 74.51%。除减少 GC 次数以外，还因为使用线程日志写方式迁移被清理段的有效块，FAGC 的有效块迁移数量比传统 F2FS、MWHFB 和 ATGC 分别减少 82.82%、83.84% 和 73.92%。FAGC 的空闲空间碎片数量分别比 F2FS、MWHFB 和 ATGC 多 2.98%、9.04% 和 7.78%，主要原因在于 FAGC 触发的 GC 次数少于 F2FS 和 ATGC，FAGC 整理的空闲空间碎片数量更少，与 FAGC 策略减少的 GC 次数相比，FAGC 比 F2FS 和 ATGC 减少整理的空闲空间碎片数量是可以接受的。FAGC 的空闲空间碎片数量比 F2FS 少，但空闲空间碎片容量反而比 F2FS 减少 1.45%，是因为 F2FS 使用追加写日志方式迁移有效块，有更多新段被写脏。

与 FAGC 减少的 GC 次数相比，FAGC 的空闲空间碎片数量和容量没有明显地增加，说明 FAGC 没有因为 GC 次数的减少导致其整理的空闲空间碎片数量大幅减少，FAGC 的每次 GC 都尽可能地选择空闲空间碎片化严重的段，尽可能多地减少小的空闲空间碎片。在设计的实验中，文件系统是在初始状态完成顺序写请求，选用图 4.6 中初始状态能耗最少的一次后台 GC，一次后台 GC 消耗 102.79 mJ 能量，则 F2FS、MWHFB、ATGC 和 FAGC 分别消耗 198.8 J、216.37 J、135.07 J 和 34.43 J 的能量，FAGC 分别比传统 F2FS 和 ATGC 少消耗 164.37 J、100.64 J 的能量。这源自以下两

个方面，一方面 FAGC 选择清理对象时增加考虑空闲空间碎片系数，使得无需迁移一部分满足后台 GC 有效块和年龄要求而迁移后获得空闲空间收益较小的段，如一个在传统 F2FS 后台 GC 中会作为垃圾回收对象的段，本就有一个大小为 1 MB 的无效空间，经垃圾回收后获得 2 MB 的空闲空间，相比于有 256 个 4 KB 大小的无效空间的段，其垃圾回收的收益较小，该段在 FAGC 中不会触发垃圾回收。另一方面 FAGC 以相同的迁移成本选择空闲空间碎片化程度最严重的段，能尽量合并小的空闲空间碎片，增加获得连续空闲空间的收益。

4.4.2 垃圾回收次数与能耗

本小节评估完成一次 I/O 密集型任务的 GC 开销，该 I/O 密集型任务包括碎片化负载、16 个 128 MB 大小文件的随机写和两次重放五个应用的 trace，此处使用预留文件，触发的 GC 次数和有效块迁移数量如表 4.4 所示，FAGC 的 GC 次数分别比传统 F2FS 和 ATGC 减少 77.68%、73.58%，FAGC 的有效块迁移数量分别比传统 F2FS 和 ATGC 减少 74.15%、72.22%。ARST 使预留文件在预留空间内就地更新以减少文件碎片，FAGC 旨在减少 GC 次数，可以看到 FAGC 的 GC 次数和有效块迁移数量比 ARST 分别减少 67.44%、72.67%。整个测试过程的时间比较长，用能耗测试系统收集采样数据并分析比较困难。此时文件系统处于末端碎片化状态，文件系统空间使用率高且文件碎片和空闲空间碎片数量都比较多，选用图 4.6 中末端碎片化状态最少的一次后台 GC 能耗 131.42 mJ 为基准，则 F2FS、MWHFB、ATGC、FAGC 和 ARST 分别消耗 57.69 J、45.73 J、48.76 J、12.88 J 和 39.56 J 的能量。FAGC 分别比传统 F2FS、MWHFB、ATGC 和 ARST 少消耗 44.81 J、32.85 J、35.88 J 和 26.68 J 的能量。这主要归功于 FAGC 会选择 GC 收益最大的段作为清理对象，使得其垃圾回收次数比 FAGC 和 ARST 都少，能够有效减少触发 GC 的能耗开销，提高单位能耗回收无效空间的收益。

表 4.4 垃圾回收开销

策略	F2FS	MWHFB	ATGC	FAGC	ARST
垃圾回收次数（次）	439	348	371	98	301
有效块迁移数量（千块）	137.669	122.413	128.139	35.593	130.213

4.4.3 应用执行时间

应用执行时间是整个系统性能的重要指标之一，文件系统空间使用率高时，不同方案重放应用 trace 的时间如图 4.10 所示。对于数据量小的脸书和推特的 trace，其触发的 GC 次数有限，FAGC 通过减少 GC 以获得减少重放时间的效果有限。对于数据量较大的百度、微信和王者荣耀的 trace，FAGC 能有效地减少应用 trace 的重放时间，与传统 F2FS 相比，对于百度、微信和王者荣耀，FAGC 分别减少 36.83%、24.4% 和 40.9% 的执行时间。

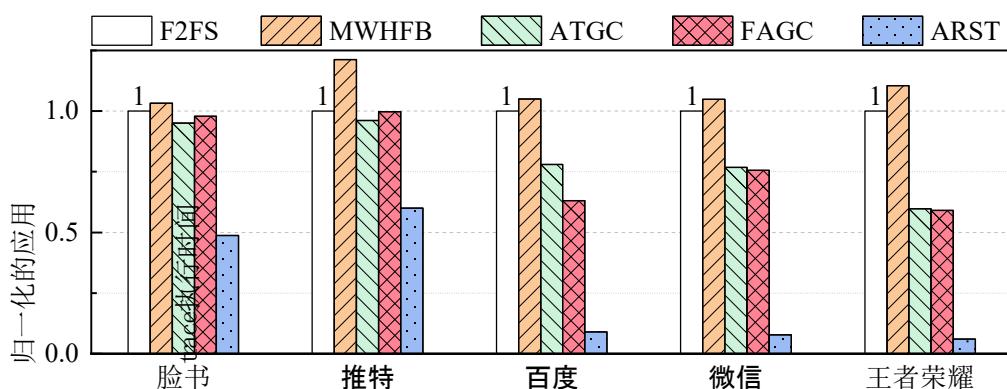


图 4.10 归一化的应用 trace 重放时间

ARST 策略有效减少了文件碎片数量，预留文件在预留空间内就地更新，其应用 trace 重放时间最短。当应用执行时间减少时，与时间呈正相关的能耗也显著减少。在较高的文件系统空间使用率，且存在大量的文件碎片和空闲空间碎片的情况下，FAGC 仍能有效地减少重放应用 trace 的时间，一方面，FAGC 增加空闲空间碎片化程度的筛选条件，减少了 GC 次数，在重放大型应用 trace 时不会因为触发 GC 阻塞常规 I/O；另一方面，FAGC 触发的 GC 尽可能选择空闲空间碎片化严重的段，减少了空闲空间碎片，保证了连续空闲空间的供应，减少了重放大型应用 trace 的执行时间，减少了能量消耗。

4.4.4 单个文件的读写性能

GC 能减少空闲空间碎片，获得空闲段来保证新写入文件的 I/O 性能，测试 FAGC 减少 GC 次数的同时衡量对末端碎片化状态文件系统中常规文件 I/O 性能的影响。设

置 128 MB 大小的文件直接和同步 I/O，单个文件的 I/O 性能如图 4.11 所示，此处 ARST 策略衡量的是预留文件的性能，可以看到 FAGC 策略常规文件的顺序读写和随机读写性能没有明显下降，甚至 FAGC 的顺序读性能比传统 F2FS 增加 4.86%，这表明 FAGC 没有因为减少触发 GC 次数影响文件的正常读写性能。一方面 FAGC 的线程日志写只是用来迁移有效块，不会影响其他数据的日志结构方式写入；另一方面 FAGC 策略能以较少的 GC 次数有效减少空闲空间碎片数量和容量，最大收益地获得连续空闲空间用于满足新文件的写入请求。因为 ARST 有效减少预留文件的文件碎片，预留文件在预留空间内就地更新，ARST 策略中该预留文件的 I/O 性能是最佳的。

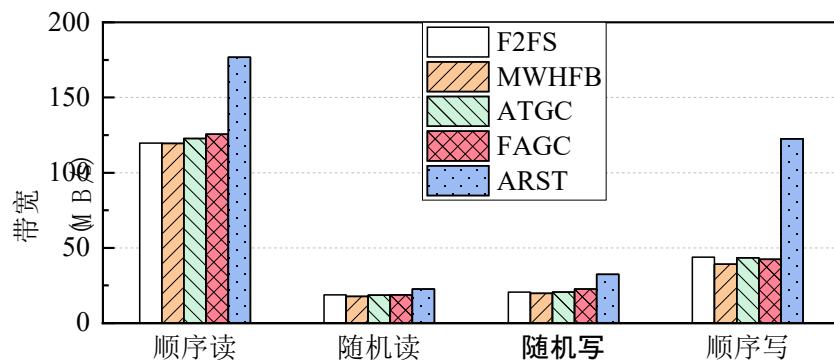


图 4.11 单个文件的直接同步读写性能

4.4.5 成本与未来改进

基于数据分析重新评估空闲空间碎片大小的阈值，并提出空闲空间碎片系数 f 快速衡量空闲空间碎片化程度，FAGC 改善选择清理对象的算法和迁移有效块的写入方式，空间开销在于结构体中增加记录的空闲空间碎片系数 f ，计算开销在于触发垃圾回收寻找清理对象时计算 f ，空间开销和计算开销都比较小。FAGC 更有效地减少 GC 次数，减少空闲空间碎片数量的效果不如预期，这是因为减少 GC 次数使得 GC 整理的空闲空间碎片数量减少，而因为每次 GC 选择空闲空间碎片化严重的段，所以空闲空间碎片容量与其他策略相差不大。

结合应用场景未来可以尝试直接减少空闲空间碎片数量的方法，如 ARST 直接减少文件碎片数量。此外因为完成 GC 的时间比较长，直观地监测减少的能耗比较困难，只能通过减少的 GC 次数计算能耗减少的理论值。若要直观地减少能耗，可以通

华中科技大学博士学位论文

过研究 CPU 能耗的瓶颈并突破，也需要关注顺序读写任务的 UFS 能耗。虽然前台 GC 只在必要的时候触发，但前台 GC 的能耗比后台 GC 能耗还高，也可以提出方法用于直接减少前台 GC 能耗。

4.5 本章小结

移动设备因为电池容量有限，提高智能手机续航能力受到工业界各大厂商和学术界科研人员的关注。本章构建了不同场景研究碎片数量和垃圾回收对存储系统能耗的影响，观察到空闲空间碎片导致 CPU 和 UFS 完成 I/O 请求的能耗增加，F2FS 通过 GC 整理空闲空间碎片，实验分析发现虽然后台 GC 能耗大，但其减少空闲空间碎片的效果有限，针对该挑战，提出了空闲空间碎片感知的 FAGC 策略，主要贡献如下：

(1) 比较了初始、重载和末端碎片化状态下 F2FS 执行顺序读写与随机读写、顺序读碎片文件与连续文件和 GC 等任务的时延与能耗，研究了空间使用率和空闲空间碎片数量对日志结构文件系统时延与能耗的影响，提出了移动设备日志结构文件系统减少能耗的需求。

(2) 观察到一次后台 GC 的能耗比较大，空闲空间碎片积累导致 GC 能耗增加，而后台 GC 减少空闲空间碎片的效果有限。基于内核预读请求和负载 I/O 请求大小的数据分析设置空闲空间碎片大小的阈值，采用空闲空间碎片系数有效衡量空闲空间碎片化程度，提出感知数据段空闲空间碎片化程度的 FAGC 策略减少 GC 次数，提高 GC 减少空闲空间碎片的效率。

(3) 在实际实验平台上比较 FAGC 与传统 F2FS、MWHFB、ATGC 和 ARST 策略的性能，实验结果显示 FAGC 减少了 GC 次数，有效地减少了 I/O 密集型负载的能耗，使得每次 GC 尽可能多地减少了空闲空间碎片，缩短了大型应用 24.4%~40.9% 的执行时间。

5 总结与展望

随着闪存技术的发展，越来越多的闪存产品在工业界发挥重要作用，如 SSD 在数据中心的普及和基于 NAND 闪存的移动设备的流行，F2FS 作为典型的日志结构文件系统，因其面向闪存特性的经典设计，受到工业界和学术界关注。在文件系统性能优化中，提高数据读写性能、缩短 I/O 请求响应时间、降低设备能耗是不变的需求。随着文件系统的使用，文件碎片和空闲空间碎片逐渐积累，导致文件系统不断老化，应用高并发的同步小写请求和 F2FS 的异地更新机制加重了碎片化程度，导致请求响应时间增加，用户使用体验不佳。F2FS 使用段清理回收异地更新后失效的旧版本数据，不同热度的数据混合存储进一步加剧了空闲空间碎片，增加了段清理需要迁移的有效块数量，导致读写数据量增加，不佳的数据组织布局非常影响文件系统性能。能量消耗过快导致用户反复充电的不良体验，还会增加经济成本，F2FS 通过 GC 整理空闲空间碎片，但后台 GC 能耗较大且减少空闲空间碎片的效果有限，导致了无效空间的低能效回收。基于读写特征改善数据组织布局，有针对性地解决文件碎片降低读写性能、多文件数据组织布局不佳增加空闲空间碎片以及低能效减少空闲空间碎片等问题，有效提高 F2FS 读写带宽和响应速度，降低能耗。

5.1 本文的主要创新点

面向 F2FS 在移动设备和服务器上的使用场景，围绕文件碎片和空闲空间碎片，基于应用的 I/O 行为特征和机器学习算法，本文针对日志结构文件系统性能优化，从文件空间分配、多文件数据组织布局和无效空间高能效回收方面展开研究，具体的研究工作和主要成果如下：

(1) 针对 F2FS 因为文件碎片积累造成 I/O 次数增加，导致响应变慢的问题，提出一种基于文件读写特征的自适应预留空间策略 ARST 以减少文件碎片。原可以使用一个 I/O 请求完成的读写任务，因为文件碎片分割 I/O 请求需要多次 I/O，且 I/O 请求变小、随机性增加，导致读写性能下降。为文件预留空间是在分配文件空间时预防碎片产生，让文件在预留空间内就地更新以减少异地更新产生的碎片，而不同文件

的读写特征不同，为仅写一次后不再更新的文件预留空间会浪费空间，需要有针对性地选择部分文件进行空间预留。在大量文件中手动选择预留文件犹如大海捞针，为了有效选择预留文件，在分析文件读写特征后，选择与产生文件碎片密切相关的特征构造数据集，使用成本低而准确率高的机器学习算法选择预留文件。观察到用于构造数据集的 trace 中 fd 和文件之间是多对多的映射关系，根据 fd 决定预留文件可能产生错误，原因在于 fd 可能会映射到普通文件，普通文件被选作预留文件造成空间浪费。因此在 trace 中追溯到写 fd 的文件对象后，根据写文件对象构造数据集并使用机器学习算法，能更加准确地获取预留文件。为了挖掘历史信息少的文件是否能作为预留文件，ARST 在构造数据集时选择时间依赖性低的文件特征并使用 Adaboost 或随机森林算法挖掘潜在预留文件。ARST 决定预留空间的大小与垃圾回收的单位适配，为了在文件系统写得比较满时不让空间预留策略浪费空间，ARST 使用敏感性实验确定文件预留功能开启与关闭的空间使用率。实验结果显示，相比于传统 F2FS 和使用 F2FS_IPU_FSYNC 就地更新策略的 F2FS，ARST 有效地减少文件碎片和垃圾回收开销，最高能提升碎片文件 179.62% 的读写带宽；相比于使用 F2FS_IPU_FSYNC 就地更新策略的 F2FS，ARST 能减少实际应用 33.78%~92.28% 的运行时间。该研究成果发表在 DATE'20^[109] 以及《IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems》^[110] 上。

(2) 针对不同热度的温数据混合存储加剧空闲空间碎片的问题，提出细粒度划分数据热度的 M2H 策略，使用 K-means 聚类算法在文件块级区分数据热度，设计多个日志延迟写温数据并优化清理过程以减少空闲空间碎片和段清理开销。F2FS 使用段清理回收无效块，段清理会增加读写开销，分析发现温数据体量较大，不同温数据块的更新热度不同，而 F2FS 使用一个 log 写入不同热度的温数据，增加了空闲空间碎片数量，导致清理开销增加。为了精准地区分温数据热度，M2H 基于段清理次数的敏感性分析决定 K-means 聚类算法的 K 值，追踪文件块更新距离的变化以感知 I/O 模式的变化实现动态识别热度，为了减少聚类计算开销，使用采样、Mini Batch K-means 算法或使用 GPU 承担聚类计算，基于区分的数据热度写入多个日志并延迟写入以改善多文件数据组织布局。实验结果显示，对于不同热度倾斜的实际负载，与传

统 F2FS 相比,文件系统空间使用率为 90%时 M2H 段清理次数减少了 50.61%~94.36%,有效块迁移数量减少了 64.31%~99.98%,读写性能提高了 3.06%~122.36%。该研究成果发表在 DATE'21^[11]上,拓展工作投稿在《ACM Transactions on Embedded Computing Systems》。

(3) 针对后台 GC 能耗较大而后台 GC 减少空闲空间碎片效果有限的问题, 提出空闲空间碎片感知的 FAGC 策略, 使得后台 GC 高能效回收无效空间。现有工作较少研究移动设备日志结构文件系统的能耗特征, 通过构造不同文件系统空间使用率和空闲空间碎片数量的初始、重载和末端碎片化三种状态, 分别比较这三种状态不同任务场景的时延与能耗, 如随机读写与顺序读写常规文件、顺序读连续文件与碎片文件、前台 GC 和后台 GC, 并观察一次 GC 的能耗和 GC 减少空闲空间碎片的效果。通过分析上述场景的时延与能耗, 发现现有 F2FS 需要减少文件碎片和空闲空间碎片, 使 I/O 请求尽可能连续, F2FS 通过 GC 整理空闲空间碎片, 一次后台 GC 的能耗较大, 超过随机读或随机写 1 MB 数据的能耗, 然而后台 GC 减少空闲空间碎片的效果不佳, 需要减少 GC 触发次数, 同时提升每次 GC 减少空闲空间碎片的效果。通过数据分析获得空闲空间碎片大小的阈值, 使用空闲空间碎片系数衡量空闲空间碎片化程度, 提出空闲空间碎片感知的 FAGC 策略, FAGC 改善选择清理对象的算法和迁移被清理段上有效块的写方式。实验结果显示, FAGC 的 GC 次数比传统 F2FS 减少 77.68%~82.68%, 比 F2FS 最新的 GC 优化方法 ATGC 减少 73.58%~74.51%, 对于 I/O 密集型负载, FAGC 比传统 F2FS 和 ATGC 分别减少消耗 44.81~164.37 焦和 35.88~100.64 焦的能量, FAGC 减少大型应用 24.4%~40.9% 的运行时间。该研究成果投稿在 DATE'23。

5.2 未来工作展望

本文的研究工作在实际项目中落地,能落实到工业界并根据应用场景变化,随着移动互联网与人工智能(Artificial Intelligence, AI)技术的发展,各大厂商如谷歌、华为和三星推出新颖的产品,日志结构文件系统的应用随着 SSD 技术的发展和移动设备场景的丰富存在很大的发展潜能。一方面,日志结构文件系统适用于 NVMe SSD,

华中科技大学博士学位论文

如可以应用于 ZNS SSD，另一方面，商用电子设备数量激增，F2FS 作为文件系统提供服务，迫切要求优化其性能。潜在的研究集中在如下几个方面：

(1) 提高能量使用效率，提供流畅的使用体验。第 4 章测试分析了移动设备日志结构文件系统的能耗特征，需要基于能耗观察深入研究以突破能耗瓶颈，提高能量使用效率，如基于使用场景和 CPU 与内存的工作特征提出减少能耗的方法。在系统工作过程中除了碎片引起的末端卡顿以外，还存在偶发卡顿，偶发卡顿可能与内存回收、前后台应用优先级管理或 FTL 内部 GC 有关，需要进一步调研和优化。为提供流畅的使用体验，除优化卡顿外，还需要分析应用层的请求，使用快速空间分配等技术保证关键请求的响应速度。在万物互联、智能家居广泛使用的时代，对计算存储资源参差不齐的各种商用电子设备，日志结构文件系统需要基于设备上应用的行为特征和负载的多样性分别优化，以提升用户的使用体验。

(2) 存储系统结构的软硬件协同优化。一方面，随着新型非易失存储器件的发展，使用 NVM 工业产品辅助日志结构文件系统的性能优化有很大的发展潜能。利用 NVM 器件优化移动设备服务的技术自 2014 年^[112]起不断发展，一般先在模拟平台上实验，如广泛使用的 FEMU 平台^[113]，提供有效的设计方法，相信很快会有厂商推出使用 NVM 器件的商用电子设备。另一方面，新型存储协议和存储设备推陈出新，如 ZNS SSD，促进存储系统软硬件协同优化，存储软件层设计结合硬件特征扬长避短，能够更好地设计架构，最大程度地发挥硬件的优点，提升系统整体性能。

(3) 灵活使用人工智能技术。人工智能技术飞速发展，在工业界和学术界都有大量研究成果，如使用深度强化学习^{[114][115]}、神经网络技术^{[116][117]}解决存储系统的问题，但在存储系统层使用人工智能技术需要控制成本。NPU、GPU、OPU（Optical Processing Unit，光计算单元）等人工智能技术的加速器快速发展，软件堆栈的发展需要匹配硬件层的发展速度，日志结构文件系统需要支持各个加速器的整合，以及与加速器控制组件的融合，以最大程度地发挥 AI 加速器的功能，包括数据的共享与快速移动，不同组件的安全层级定义与隔离，以及关键 I/O 路径不阻塞数据的计算与运输等都需要保证。

华中科技大学博士学位论文

致 谢

窗外日光弹指过，回想那些年韵苑食堂门口看过的傍晚的云，在老国光三楼走廊吹过的风，在新光电五楼的桌位上探头扫过的一楼的花，我在华科的第十年即将过去。博士的后半程，伴随着新冠疫情，过得尤其快，曾有过论文结果通知前的焦虑，有过论文被拒的沮丧，有过科研不顺时的难受，也有过论文接收时的喜悦。曾经畅想过数次，在撰写博士学位论文时要感谢很多人的帮助与关爱。

感谢我的导师谭支鹏教授。谭老师从担任我大三时《微机接口技术》这门课的授课老师起，陪我走过硕博求学生涯，见证了我求学生涯中的低谷与高兴的时刻。在我的每个研究点遇到阻塞时，谭老师都及时地与我多次讨论，求证各种可能的解决方案，让我茅塞顿开。谭老师风趣幽默，平易近人，为课题组的同学们提供轻松的求学环境，像小太阳一样温暖人心，让我们充满干劲。谭老师又严于律己，在老国光和谭老师一个办公室时，总是见到谭老师第一个到办公室，又是最后离开办公室的那批人之一，谭老师不断督促我们的科研进展，及时与我们交流科研遇到的问题，提供各种帮助，为我们的科研和生活助力。在此向谭老师表达由衷的敬意与谢意。

感谢我的导师王芳教授。王老师严谨踏实的治学风格给我留下深刻印象，王老师会及时告诉我们各大会议和期刊的征稿信息，拓宽我们的投稿渠道。在我进入实验室没多久时就找我讨论科研方向，不断指导我的科研，在开题、中期和毕业答辩前会提前找我们预答辩，为我们的科研支招，为我们的学术报告严格把关。记得我第一次投稿时不顺利，修改再投时王老师将论文打印出来与我一段一段地讨论，帮我修改，为论文的接收夯实了基础。对我们科研上的需求，如实验平台和实习请求，王老师也是有求必应，主动提供帮助，为我们的科研提供了有力的支撑。在此真诚地感谢王老师在我科研生活中的指导与帮助，向王老师表达敬意与谢意。

感谢实验室的掌舵人冯丹教授。谢谢冯老师为我们提供广阔的科研平台，每次学术年会聆听冯老师的报告时都能拓展科研视野，实验室组织的各种学术研讨会都令我受益匪浅。冯老师在学术前沿不断耕耘，为实验室营造了浓厚的学术氛围，冯老师统筹实验室的规划与建设，以卓越的管理与领导能力带领实验室不断前进，衷心感谢

华中科技大学博士学位论文

冯老师在我博士求学期间的帮助。

感谢实验室刘景宁、华宇、曹强、童薇、施展、胡鹤翀、陈俭喜、谢雨来、曾令仿、熊双莲、王子惠琦等老师对我的指导和帮助。感谢华为海思麒麟平台与解决方案软件技术部帮助我们搭建与熟悉移动环境的实验平台，特别感谢海思的何彪不断与我讨论项目与科研上的问题，感谢海思的孔飞、刘毅、卜涛、王法、李点刚、何宇杰等人在我们合作项目以及我实习期间的帮助。

感谢实验室已经毕业的师兄师姐们，周炜、徐高翔、徐洁、冯雅植、张扬、李铮、张建权、张峥、覃鸿巍、史庆宇、苏毅、徐湘灏、刘传奇等，谢谢你们与我讨论科研生活中的问题，为我提供学术上的帮助。感谢已经毕业的同学和师弟师妹们，舒熙、陈乃辉、钱佳兴、涂诗云、赵玉亭、郭甜、李春艳、余梦姗、彭梦烨、杨明顺、唐英杰、李帅、姜越、陈硕、方圣卿、杨蕾等，谢谢你们与我交流研讨，陪我走过求学生涯的一程。感谢在读的实验室小伙伴，张鑫晏、刘聪、卢梦婷、袁莹、肖阳、邓萌、魏学亮、吴兵、程稳、刘伟华、亓文杰、邵继成、任哲旋、王宣植、张婧等，谢谢你们在我学术和生活中的陪伴，祝顺利毕业。感谢鼓励我、帮助我的朋友，刘若然、向慧琳、疏文君等，谢谢你们伴我走过艰难的求学生涯，怀念聚餐聊天的快乐时光。

由衷地感谢我的爸爸妈妈和哥哥嫂嫂，谢谢你们一直在我身边，坚定地支持我、鼓励我。谢谢亲爱的爸爸妈妈将我抚养长大，不断地为孩子们付出，谢谢爸爸妈妈倾听我的烦恼和快乐，与我分享生活的酸甜苦辣，犹记得每次视频时你们见到我的开心，语音时的谆谆叮嘱，回家时丰盛的饭菜和离家时的不舍，向我挚爱的父母表达真诚的爱意，我会继续努力的。谢谢哥哥嫂嫂对我科研上的鼓励，生活中大小问题的讨论让我们都不断进步，谢谢哥哥一路陪伴我从小学到博士，你的以身作则，关怀爱护，激励我一直走下去。

2022年夏于华中科技大学

参考文献

- [1] Changman Lee, Dongho Sim, Joo Y. Hwang, Sangyeun Cho. F2FS: A New File System for Flash Storage. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 2015), Santa Clara, CA, USA, February 16-19, 2015, USENIX Association, 2015: 273-286
- [2] Mendel Rosenblum, John K. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems, 1992, 10(1): 26-52
- [3] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh J. Shetty, Jooyoung Hwang, Sangyeun Cho, et al. FStream: Managing Flash Streams in the File System. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST 2018), Oakland, CA, USA, February 12-15, 2018, USENIX Association, 2018: 257-264
- [4] Janki Bhimani, Zhengyu Yang, Jingpei Yang, Adnan Maruf, Ningfang Mi, Rajinikanth Pandurangan, et al. Automatic Stream Identification to Improve Flash Endurance in Data Centers. ACM Transaction on Storage, 2022, 18(2): 1-29
- [5] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien L. Moal, Gregory R. Ganger, et al. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In: Proceedings of the 2021 USENIX Annual Technical Conference (ATC 2021), Virtual, July 14-16, 2021, USENIX Association, 2021: 689-703
- [6] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In: Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2021), Virtual, July 14-16, 2021, USENIX Association, 2021: 147-162
- [7] 张勇, 裴雪红. 嵌入式 Linux 下 JFFS2 文件系统的实现. 计算机技术与发展, 2006(04): 138-140
- [8] 韩春晓, 陈香兰, 李曦, 龚育昌. UBIFS 损耗均衡对系统 I/O 性能的影响. 计算机工程, 2009, 35(6): 260-262
- [9] Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, Wolfgang Reif. Abstract

华中科技大学博士学位论文

Specification of the UBIFS File System for Flash Memory. In: Proceedings of the International Symposium on Formal Methods (FM 2009). Eindhoven, Netherlands, November 2-6, 2009, Springer, 2009: 190-206

- [10] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In: Proceedings of the Linux Symposium, Ottawa, Ontario, Canada, June 27-30, 2007, Citeseer, 2007: 21-33
- [11] Joontaek Oh, Sion Ji, Yongjin Kim, Youjip Won. exF2FS: Transaction Support in Log-Structured Filesystem. In: Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST 2022), SANTA CLARA, CA, USA, FEBRUARY 22-24, 2022, USENIX Association, 2022: 345-362
- [12] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, et al. File Systems Fated for Senescence? Nonsense, Says Science! In: Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 2017), Santa Clara, CA, USA, February 27- March 2, 2017, USENIX Association, 2017: 45-58
- [13] Saurabh Kadekodi, Vaishnavh Nagarajan, Gregory R. Ganger, Garth A. Gibson, Geriatric: Aging What You See and What You Don't See. A file System Aging Approach for Modern Storage Systems. In: Proceedings of the 2018 USENIX Annual Technical Conference (ATC 2018), Boston, MA, USA, July 11-13, 2018, USENIX Association, 2018: 691-703
- [14] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, Young I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 2012), San Jose, USA, February14-17, 2012, USENIX Association, 2012: 1-16
- [15] Jiacheng Zhang, Jiwu Shu, Youyou Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In: Proceedings of the 2016 USENIX Annual Technical Conference (ATC 2016), Denver, CO, USA, June 22-24, 2016, USENIX Association, 2016: 87-100
- [16] Jayashree Mohan, Dhathri Purohit, Matthew Halpern, Vijay Chidambaram, Vijay J. Reddi. Storage on Your Smartphone Uses More Energy Than You Think. In:

华中科技大学博士学位论文

Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2017), Santa Clara, CA, USA, July 10-11, 2017, USENIX Association, 2017: 9-15

- [17] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In: Proceedings of the USENIX 1995 Technical Conference (TCON 1995), New Orleans, LA, USA, January 16-20, 1995, USENIX Association, 1995: 1-21
- [18] 郑特龙. DPLFS: 基于双模相变存储器的日志结构文件系统[硕士学位论文]. 武汉: 华中科技大学, 2017
- [19] Jian Xu, Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 2016), Santa Clara, CA, USA, February 22-25, 2016, USENIX Association, 2016: 323-338
- [20] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires B. Da Silva, et al. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017), Shanghai, China, October 28-31, 2017, ACM, 2017: 478-496
- [21] Xiaojian Liao, Youyou Lu, Erci Xu, Jiwu Shu. Max: A Multicore-Accelerated File System for Flash Storage. In: Proceedings of the 2021 USENIX Annual Technical Conference (ATC 2021), Virtual, July 14-16, 2021, USENIX Association, 2021: 877-891
- [22] Kisung Lee, Youjip Won. Smart Layers and Dumb Result: IO Characterization of an Android-Based Smartphone. In: Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT 2012), Tampere, Finland, October 7-12, 2012, ACM, 2012: 23-32
- [23] Hyojun Kim, Nitin Agrawal, Cristian Ungureanu. Revisiting Storage for Smartphones. ACM Transaction on Storage, 2012, 8(4): 1-25
- [24] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, Youjip Won. I/O stack

华中科技大学博士学位论文

- Optimization for Smartphones. In: Proceedings of the 2013 USENIX Annual Technical Conference (ATC 2013), San Jose, CA, USA, Junn 26-28, 2013, USENIX Association, 2013: 309-320
- [25] Cheng Ji, Ruiwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, et al. Inspection and Characterization of App File Usage in Mobile Devices. ACM Transaction on Storage, 2020, 16(4): 1-25
- [26] Jace Courville, Feng Chen. Understanding Storage I/O Behaviors of Mobile Applications. In: Proceedings of the 32nd Symposium on Mass Storage Systems and Technologies (MSST 2016), Santa Clara, CA, USA, May 2-6, 2016, IEEE, 2016: 1-11
- [27] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington, Qi Zhang. On the Energy Overhead of Mobile Storage Systems. In: Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 2014), Santa Clara, CA, USA, February 17-20, 2014, USENIX Association, 2014: 105-118
- [28] Weichao Guo, Kang Chen, Huan Feng, Yongwei Wu, Rui Zhang, Weimin Zheng. MARS: Mobile Application Relaunching Speed-Up through Flash-Aware Page Swapping. IEEE Transactions on Computers, 2016, 65(3): 916-928
- [29] Yuhao Zhu, Matthew Halpern, Vijay J. Reddi. Event-based Scheduling for Energy-Efficient QoS (eQoS) in Mobile Web Applications. In: Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA 2015), Burlingame, CA, USA, February 7-11, 2015, IEEE, 2015: 137-149
- [30] Luis Cruz, Rui Abreu. EMaaS: Energy Measurements as a Service for Mobile Applications. In: Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER 2019), Montreal, QC, Canada, May 25-31, 2019, IEEE, 2019: 101-104
- [31] Eunji Kwon, Sodam Han, Yoonho Park, Young H. Kim, Seokhyeong Kang. Reinforcement Learning-Based Power Management Policy for Mobile Device Systems: Late Breaking Results. In: Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC 2020), Virtual Event, July 20-24, 2020, IEEE, 2020: 1-2
- [32] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman.

华中科技大学博士学位论文

- Don't Stack Your Log On My Log. In: Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 2014), Broomfield, CO, USA, October 5, 2014, USENIX Association, 2014: 1-10
- [33] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, Youjip Won. AndroStep: Android Storage Performance Analysis Tool. In: Proceedings of Software Engineering 2013 - Workshopband, Aachen, Germany, February 26 - March 1, 2013, Gesellschaft für Informatik e.V., 2013: 327-340
- [34] Sangwook S. Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, Jihong Kim. FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones. In: Proceedings of the 2018 USENIX Annual Technical Conference (ATC 2018), Boston, MA, USA, July 11-13, 2018, USENIX Association, 2018: 15-28
- [35] Lei Han, Bin Xiao, Xuwei Dong, Zhaoyan Shen, Zili Shao. DS-Cache: A Refined Directory Entry Lookup Cache with Prefix-Awareness for Mobile Devices. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE 2019), Florence, Italy, March 25-29, 2019, IEEE, 2019: 1052-1057
- [36] Yu Liang, Qiao Li, Chun J. Xue. Mismatched Memory Management of Android Smartphones. In: Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2019), Renton, WA, USA, July 8-9, 2019, USENIX Association, 2019: 1-6
- [37] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, et al. Acclaim: Adaptive Memory Reclaim to Improve User Experience in Android Systems. In: Proceedings of the 2020 USENIX Annual Technical Conference (ATC 2020), Virtual, July 15-17, 2020, USENIX Association, 2020: 897-910
- [38] Bo Mao, Suzhen Wu, Hong Jiang, Xiao Chen, Weijian Yang. Content-aware Trace Collection and I/O Deduplication for Smartphones. In: Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017), Santa Clara, CA, USA, May 15-19, 2017, IEEE, 2017: 1-8
- [39] Bo Mao, Jindong Zhou, Suzhen Wu, Hong Jiang, Xiao Chen, Weijian Yang. Improving Flash Memory Performance and Reliability for Smartphones With I/O Deduplication. IEEE Transactions on Computer-Aided Design of Integrated Circuits

华中科技大学博士学位论文

- and Systems, 2019, 38(6): 1017-1027
- [40] Cheng Ji, Li-Pin Chang, Riwei Pan, Chao Wu, Congming Gao, Liang Shi, et al. Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices. In: Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 2021), Virtual, February 23-25, 2021, USENIX Association, 2021: 127-140
- [41] 任晶磊. 持久性内存系统中高效的数据一致性机制研究[博士学位论文]. 北京: 清华大学, 2015
- [42] 孙鉴. 存储系统能耗建模及度量方法研究[博士学位论文]. 西安: 西北工业大学, 2017
- [43] Aaron Carroll, Gernot Heiser. An Analysis of Power Consumption in a Smartphone. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (ATC 2010), Boston, MA, USA, June 23-25, 2010, USENIX Association, 2010: 1-14
- [44] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington, Qi Zhang. On the Energy Overhead of Mobile Storage Systems. In: Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 2014), Santa Clara, CA, USA, February 17-20, 2014, USENIX Association, 2014: 105-118
- [45] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, Hojung Cha. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring. In: Proceedings of the 2012 USENIX Annual Technical Conference (ATC 2012), Boston, MA, USA, June 13-15, 2012, USENIX Association, 2012: 387-400
- [46] Yi-Fan Chung, Yin-Tsung Lo, Chung-Ta King. Enhancing User Experiences by Exploiting Energy and Launch Delay Trade-off of Mobile Multimedia Applications. ACM Transactions on Embedded Computing Systems, 2013, 12(1s): 1-19
- [47] Pierre Olivier, Jalil Boukhobza, Eric Senn, Hamza Ouarnoughi. A Methodology for Estimating Performance and Power Consumption of Embedded Flash File Systems. ACM Transactions on Embedded Computing Systems, 2016, 15(4): 1-25
- [48] David T. Nguyen, Gang Zhou, Xin Qi, Ge Peng, Jianing Zhao, Tommy Nguyen, et al.

华中科技大学博士学位论文

- Storage-aware Smartphone Energy Savings. In: Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing (UbiComp 2013), Zurich, Switzerland, September 8-12, 2013, ACM, 2013: 677-686
- [49] Jian Huang, Anirudh Badam, Ranveer Chandra, Edmund B. Nightingale. WearDrive: Fast and Energy-Efficient Storage for Wearables. In: Proceedings of the 2015 USENIX Annual Technical Conference (ATC 2015), Santa Clara, CA, USA, July 8-10, 2015, USENIX Association, 2015: 613-625
- [50] Kan Zhong, Xiao Zhu, Tianzheng Wang, Dan Zhang, Xianlu Luo, Duo Liu, et al. DR. Swap: Energy-efficient Paging for Smartphones. In: Proceedings of the 2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED 2014), La Jolla, CA, USA, August 11-13, 2014, IEEE: 81-86
- [51] Kaige Yan, Xin Fu. Energy-efficient Cache Design in Emerging Mobile Platforms: The Implications and Optimizations. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE 2015), Grenoble, France, March 9-13, 2015, IEEE, 2015: 375-380
- [52] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, Kesari Mishra. Countering Fragmentation in an Enterprise Storage System. ACM Transaction on Storage, 2020, 15(4): 1-35
- [53] Keith A. Smith, Margo I. Seltzer. File System Aging Increasing the Relevance of File System Benchmarks. ACM SIGMETRICS Performance Evaluation Review, 1997, 25(1): 203-213
- [54] Yu Liang, Chenchen Fu, Yajuan Du, Aosong Deng, Mengying Zhao, Liang Shi, et al. An Empirical Study of F2FS on Mobile Devices. In: Proceedings of the 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2017), Hsinchu, Taiwan, China, August 16-18, 2017, IEEE, 2017: 1-9
- [55] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, Chun J. Xue. An Empirical Study of File-system Fragmentation in Mobile Storage Systems. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2016), Denver, CO, USA, June 20-21, 2016, USENIX Association, 2016: 76-80
- [56] Cheng Ji, Li-Pin Chang, Sangwook S. Hahn, Sungjin Lee, Riwei Pan, Liang Shi, et al.

华中科技大学博士学位论文

- File Fragmentation in Mobile Devices: Measurement, Evaluation, and Treatment.
IEEE Transactions on Mobile Computing, 2019, 18(9): 2062-2076
- [57] Aneesh K. KV, Mingming Cao, Jose R. Santos, Andreas Dilger. Ext4 Block and Inode Allocator Improvements. In: Proceedings of the Linux Symposium, Ottawa, Ontario, Canada, July 23-26, 2008, Citeseer, 2008: 263-273
- [58] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, Geoff Peck. Scalability in the XFS File System. In: Proceedings of the USENIX 1996 Annual Technical Conference (ATC 1996), San Diego, CA, USA, January 22-26, 1996, USENIX Association, 1996: 1-14
- [59] Ohad Rodeh, Josef Bacik, Chris Mason. BTRFS: The Linux B-Tree Filesystem. ACM Transaction of Storage, 2013, 9(3): 1-32
- [60] Jeff Bonwick, Bill Moore. ZFS: The Last Word in File Systems. In: Proceedings of the 21st Large Installation System Administration Conference (LISA 2007), Dallas, TX, USA, November11-16, 2007, USENIX Association, 2007: 1-17
- [61] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, et al. BetrFS: A Right-optimized Write-optimized File System. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 2015), Santa Clara, CA, USA, February 16-19, 2015, USENIX Association, 2015: 301-315
- [62] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, et al. BetrFS: Write-Optimization in a Kernel File System. ACM Transaction of Storage, 2015, 11(4): 1553-3077
- [63] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, et al. Optimizing Every Operation in a Write-optimized File System. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 2016), Santa Clara, CA, USA, February 22-25, 2016, USENIX Association, 2016: 1-14
- [64] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, et al. Filesystem Aging: It's more Usage than Fullness. In: Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2019), Renton, WA, USA, July 8-9, 2019, USENIX Association, 2019: 1-6
- [65] Sangwook S. Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, et

华中科技大学博士学位论文

- al. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In: Proceedings of the 2017 USENIX Annual Technical Conference (ATC 2017), Santa Clara, CA, USA, July 12-14, 2017, USENIX Association, 2017: 759-771
- [66] Jonggyu Park, Young I. Eom. FragPicker: A New Defragmentation Tool for Modern Storage Devices. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021), Virtual Event, Germany, October 26-29, 2021, ACM, 2021: 280-294
- [67] Takaki Nakamura, Norihisa Komoda. Pre-allocation Size Adjusting Methods Depending on Growing File Size. In: Proceedings of the 5th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2008), Baltimore, MD, USA, September 22, 2008, IEEE, 2008: 19-25
- [68] Borislav Djordjevic, Valentina Timcenko. Ext4 File System in Linux Environment: Features and Performance Analysis. Citeseer International Journal of Computers, 2012,6(1): 37-45
- [69] Qi Li, Aosong Deng, Congming Gao, Yu Liang, Liang Shi, Edwin H-M Sha. Optimizing Fragmentation and Segment Cleaning for CPS based Storage Devices. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC 2019), Limassol, Cyprus, April 8-12, 2019, ACM, 2019: 242-249
- [70] Woo H. Ahn, Kyungbaek Kim, Yongjin Choi, Daeyeon Park. DFS: A De-fragmented File System. In: Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2002), Fort Worth, TX, USA, October 16, 2002, IEEE, 2002: 71-80
- [71] Jonggyu Park, Dong H. Kang, Young I. Eom. File Defragmentation Scheme for a Log-structured File System. In: Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2016), New York, NY, USA, August 4-5 2016, ACM, 2016: 1-7
- [72] Jonggyu Park, Young I. Eom. Anti-Aging LFS: Self-Defragmentation with Fragmentation-Aware Cleaning, IEEE Access, 2020, 8: 151474-151486
- [73] Stephanie N. Jones, Ahmed Amer, Ethan L. Miller, Darrell D. E. Long, Rekha Pitchumani, Christina R. Strong. Classifying Data to Reduce Long-Term Data

华中科技大学博士学位论文

- Movement in Shingled Write Disks. ACM Transaction of Storage, 2016, 12(1): 1-17
- [74] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Vagelis Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In: Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 2009), San Francisco, CA, USA, February 24-27, 2009, USENIX Association, 2009: 183-196
- [75] Sang-Won Lee, Bongki Moon. Design of Flash-Based DBMS: An in-Page Logging Approach. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD 2007), Beijing, China, June 11-14, 2007, ACM, 2007: 55-66
- [76] 刘景超. UBIFS 文件系统优化[硕士学位论文]. 武汉: 华中科技大学, 2016
- [77] Dong H. Kang, Young I. Eom. Dynamic Hot-cold Separation Scheme on the Log-structured File System for CE Devices. In: Proceedings of the 2017 IEEE International Conference on Consumer Electronics (ICCE 2017), Las Vegas, NV, USA, June 12-14, 2017, IEEE, 2017: 428-429
- [78] Wenguang Wang, Yanping Zhao, Rick Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 2004), San Francisco, CA, USA, March 31- April 2, 2004, USENIX Association, 2004: 145-158
- [79] Wei Xie, Yong Chen, Philip C. Roth. ASA-FTL: An Adaptive Aeparation Aware Flash Translation Layer for Solid State Drives. Parallel Computing, 2017, 61: 3-17
- [80] Bingzhe Li, Chunhua Deng, Jinfeng Yang, David Lilja, Bo Yuan, David Du. HAML-SSD: A Hardware Accelerated Hotness-Aware Machine Learning based SSD Management. In: Proceedings of the 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2019), Westminster, CO, USA, November 4-7, 2019, IEEE, 2019: 1-8
- [81] Mei-Ling Chiang, Paul C. Lee, Ruei-Chuan Chang. Using Data Clustering to Improve Cleaning Performance for Flash Memory. Software: Practice and Experience, 1999, 29(3): 267-290
- [82] Taejin Kim, Duwon Hong, Sangwook S. Hahn, Myoungjun Chun, Sungjin Lee,

华中科技大学博士学位论文

- Jooyoung Hwang, et al. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In: Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 2019), Boston, MA, USA, February 25-28, 2019, USENIX Association, 2019: 295-308
- [83] Hyunho Gwak, Yunji Kang, Dongkun Shin. Reducing Garbage Collection Overhead of Log-structured File Systems with GC Journaling. In: Proceedings of the 2015 International Symposium on Consumer Electronics (ISCE 2015), UPM, Madrid, Spain, June 24-26, 2015, IEEE, 2012: 1-2
- [84] Hyunho Gwak, Dongkun Shin. SCJ: Segment Cleaning Journaling for Log-Structured File Systems. IEEE Access, 2021, 9: 142437-142448
- [85] Dongil Park, Seungyong Cheon, Youjip Won. Suspend-aware Segment Cleaning in Log-structured File System. In: Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2015), Santa Clara, CA, USA, July 6-7, 2015, USENIX Association, 2015: 1-17
- [86] Chao Wu, Cheng Ji, Chun J. Xue. Reinforcement Learning based Background Segment Cleaning for Log-structured File System on Mobile Devices. In: Proceedings of the 2019 IEEE International Conference on Embedded Software and Systems (ICESS 2019), Las Vegas, NV, USA, June 2-3, 2019, IEEE, 2019: 1-8
- [87] Chao Wu, Yufei Cui, Cheng Ji, Tei-Wei Kuo, Chun J. Xue. Pruning Deep Reinforcement Learning for Dual User Experience and Storage Lifetime Improvement on Mobile Devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020, 39(11): 3993-4005
- [88] 李爱军, 罗四维, 黄华, 刘蕴辉. 基于决策树的神经网络. 计算机研究与发展, 2005(08): 1312-1317
- [89] Hua Wang, Xinbo Yi, Ping Huang, Bin Cheng, Ke Zhou. Efficient SSD Caching by Avoiding Unnecessary Writes using Machine Learning. In: Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018), Eugene, Oregon, USA, August 13-16, 2018, ACM, 2018: 1-10
- [90] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In: Proceedings of the 1st ACM

华中科技大学博士学位论文

Symposium on Cloud Computing (SoCC 2010), Indianapolis, Indiana, USA, June 10-11, 2010, ACM, 2010: 143-154

- [91] Radu Stoica, Anastasia Ailamaki. Improving Flash Write Performance by Using Update Frequency. Proceedings of the VLDB Endowment, 2013, 6(9): 733-744
- [92] Kevin Kremer, André Brinkmann. FADaC: A Self-Adapting Data Classifier for Flash Memory. In: Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR 2019), Haifa, Israel, June 3-5, 2019, ACM, 2019: 167-178
- [93] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, et al. Reducing Garbage Collection Overhead in SSD Based on Workload Prediction. In: Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2019), Renton, WA, USA, July 8-9, 2019, USENIX Association, 2019: 1-6
- [94] Song Jiang, Xiaodong Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. SIGMETRICS Performance Evaluation Review, 2002, 30(1): 31-42
- [95] Qiao Li, Liang Shi, Chun J. Xue, Kaijie Wu, Cheng Ji, Qingfeng Zhuge, et al. Access Characteristic Guided Read and Write Cost Regulation for Performance Improvement on Flash Memory. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 2016), Santa Clara, CA, USA, February 22-25, 2016, USENIX Association, 2016: 125-132
- [96] John A. Hartigan, Manchek A. Wong. Algorithm AS 136: A K-means Clustering Algorithm. Journal of the Royal Statistical Society, 1979, 28(1): 100-108
- [97] 黄晓辉, 王成, 熊李艳, 曾辉. 一种集成簇内和簇间距离的加权 k-means 聚类方法. 计算机学报, 2019, 42(12): 2836-2848
- [98] Marutho Dhendra, Hendra H. Sunarna, Wijaya Ekaprana, Muljono. The Determination of Cluster Number at k-mean Using Elbow Method and Purity Evaluation on Headline News. In: Proceedings of the 2018 International Seminar on Application for Technology of Information and Communication (iSemantic 2018), Semarang, Indonesia, September 21-22, 2018, IEEE, 2018: 533-538
- [99] Robert Tibshirani, Guenther Walther, Trevor Hastie. Estimating the Number of

华中科技大学博士学位论文

- Clusters in a Data Set via the Gap Statistic. Journal of the Royal Statistical Society: Series B (Statistical Methodology), 2001, 63(2): 411-423
- [100] S. Aranganayagi, K. Thangavel. Clustering Categorical Data Using Silhouette Coefficient as a Relocating Measure. In: Proceedings of the International Conference on Computational Intelligence and Multimedia Applications (ICCIMA 2007), Sivakasi, India, December 13-15, 2007, IEEE, 2007: 13-17
- [101] David Arthur, Sergei Vassilvitskii. K-Means++: The Advantages of Careful Seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007), New Orleans, Louisiana, USA, January 7-9, 2007, Society for Industrial and Applied Mathematics, 2007: 1027-1035
- [102] D. Sculley. Web-Scale k-Means Clustering. In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010), Raleigh, North Carolina, USA, April 26-30, 2010, ACM, 2010: 1177-1178
- [103] Kai Peng, Victor C. M. Leung, Qingjia Huang. Clustering Approach Based on Mini Batch Kmeans for Intrusion Detection System Over Big Data. IEEE Access, 2018, 6: 11897-11906
- [104] Mohammed A. Ahmed, Hanif Baharin, Puteri NE. Nohuddin. Mini-batch K-means versus K-means to Cluster English Tafseer Text: View of Al-Baqarah Chapter. Journal of Quranic Sciences and Research, 2021, 2(2): 48-53
- [105] Ali Feizollah, Nor B. Anuar, Rosli Salleh, Fairuz Amalina. Comparative Study of K-means and Mini Batch K-means Clustering Algorithms in Android Malware Detection Using Network Traffic Analysis. In: Proceedings of the 2014 International Symposium on Biometrics and Security Technologies (ISBAST 2014), Kuala Lumpur, Malaysia, August 26-27, 2014, IEEE, 2014: 193-197
- [106] Luc Bouganim, Björn Jónsson, Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In: Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR 2009), Asilomar, CA, USA, January 4-7, 2009, Online Proceeding, 2009: 1-12
- [107] Junghoon Kim, Sundoo Kim, Juseong Yun, Youjip Won. Energy Efficient IO stack Design for Wearable Device. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC 2019), Limassol, Cyprus, April 8-12, 2019, ACM, 2019:

华中科技大学博士学位论文

2152-2159

- [108] Jawad Haj-Yahya, Yanos Sazeides, Mohammed Alser, Efraim Rotem, Onur Mutlu. Techniques for Reducing the Connected-Standby Energy Consumption of Mobile Devices. In: Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA 2020), San Diego, CA, USA, February 22-26, 2020, IEEE, 2020: 623-636
- [109] Lihua Yang, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, Shiyun Tu. ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE 2020), Grenoble, France, March 9-13, 2020, European Design and Automation Association press, 2020: 1061-1066
- [110] Lihua Yang, Zhipeng Tan, Fang Wang, Dan Feng, Hongwei Qin, Jiaxing Qian. Improving F2FS Performance in Mobile Devices with Adaptive Reserved Space Based on Traceback. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022, 41(1): 169-182
- [111] Lihua Yang, Zhipeng Tan, Fang Wang, Shiyu Tu, Jicheng Shao. M2H: Optimizing F2FS via Multi-log Delayed Writing and Modified Segment Cleaning based on Dynamically Identified Hotness. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE 2021), Grenoble, France, February 1-5, 2021, European Design and Automation Association press, 2021: 808-811
- [112] Hao Luo, Lei Tian, Hong Jiang. qNVRAM: Quasi Non-volatile RAM for Low Overhead Persistency Enforcement in Smartphones. In: Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2014), Philadelphia, PA, USA, June 17-18, 2014, USENIX Association, 2014: 1-6
- [113] Huaicheng Li, Mingzhe Hao, Michael H. Tong, Swaminathan Sundararaman, Matias Bjørling, Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST 2018), Oakland, CA, USA, February 12-15, 2018, USENIX Association, 2018: 83-90
- [114] Subhash Sethumurugan, Jieming Yin, John Sartori. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In: Proceedings of the 27th IEEE

华中科技大学博士学位论文

International Symposium on High Performance Computer Architecture (HPCA 2021),
Seoul, South Korea, Feb. 27- Mar. 3, 2021, IEEE, 2021: 291-303

- [115] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, Onur Mutlu. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In: Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2021), Virtual Event, Greece, October 18-22, 2021, ACM, 2021: 1121-1137
- [116] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, et al. CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime. In: Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST 2022), Santa Clara, CA, USA, February 22-24, 2022, USENIX Association, 2022: 445-459
- [117] 郭佳. 基于改进的人工神经网络对存储系统性能进行预测的方法. 计算机科学, 2019, 46(S1): 52-55

附录 1 攻读博士学位期间取得的研究成果

发表与接收论文

- [1] **Lihua Yang**, Zhipeng Tan, Fang Wang, Dan Feng, Hongwei Qin, Shiyun Tu, Jiaxing Qian, Yuting Zhao. Improving F2FS Performance in Mobile Devices with Adaptive Reserved Space Based on Traceback. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022, 41(1): 169-182 (SCI 收录; IF: 2.807;
中国计算机学会推荐 A 类期刊; 署名单位: 华中科技大学)
- [2] **Lihua Yang**, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, Shiyu Tu. ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees. In: *Proceedings of the 23th Design, Automation and Test in Europe (DATE 2020)*, Grenoble, France, March 9-13, 2020, European Design and Automation Association press, 2020: 1061-1066 (EI 收录; 中国计算机学会推荐 B 类国际会议; 署名单位: 华中科技大学)
- [3] **Lihua Yang**, Zhipeng Tan, Fang Wang, Shiyu Tu, Jicheng Shao. M2H: Optimizing F2FS via Multi-log Delayed Writing and Modified Segment Cleaning based on Dynamically Identified Hotness. In: *Proceedings of the 24th Design, Automation and Test in Europe (DATE 2021)*, Grenoble, France, February 1-5, 2021, European Design and Automation Association press, 2021: 808-811 (EI 收录; 中国计算机学会推荐 B 类国际会议; 署名单位: 华中科技大学)
- [4] **Lihua Yang**, Zhipeng Tan, Fang Wang, Dan Feng, Yang Xiao, Wenjie Qi. Improving F2FS Performance with the Hotness of Warm Data Identified by K-means Clustering. *ACM Transactions on Embedded Computing Systems*, 2022 (SCI 收录; 中国计算机学会推荐 B 类期刊; 署名单位: 华中科技大学; 在投)
- [5] **Lihua Yang**, Zhipeng Tan, Fang Wang, Yang Xiao, Biao He. Understanding and Reducing the Energy Consumption of F2FS on Smartphones. In: *Proceedings of the 26th Design, Automation and Test in Europe (DATE 2023)*, Antwerp, BE, April 17-19, 2023 (EI 收录; 中国计算机学会推荐 B 类国际会议; 署名单位: 华中科技大学; 在投)

华中科技大学博士学位论文

- [6] Wenjie Qi, Zhipeng Tan, Jicheng Shao, **Lihua Yang**, Yang Xiao. InDeF: An Advanced Defragmenter Supporting Migration Offloading on ZNS SSD. In Proceedings of the 2022 IEEE International Conference on Computer Design (ICCD 2022), Lake Tahoe, USA, October 23-26, 2022, 8 pages (EI 收录; 中国计算机学会推荐 B 类国际会议; 署名单位: 华中科技大学; 已接收)

专 利

- [1] 谭支鹏, **杨梨花**, 王芳, 冯丹, 涂诗云, 钱佳兴. 一种选择性预留空间的减少碎片的方法及系统. 中国, 发明专利, ZL 201910681862.8, 2019.7.26 (专利权人: 华中科技大学)
- [2] 谭支鹏, **杨梨花**, 王芳, 冯丹, 涂诗云. 一种基于热度的日志结构文件系统数据管理方法. 中国, 发明专利, 202010705796.6, 2020.9.2 (专利权人: 华中科技大学)
- [3] 谭支鹏, **杨梨花**, 王芳, 冯丹, 钱佳兴. 一种减少日志结构文件系统碎片的方法及闪存存储系统. 中国, 发明专利, 202010811619.6, 2020.9.23 (专利权人: 华中科技大学)
- [4] 谭支鹏, **杨梨花**, 王芳, 冯丹, 肖阳. 一种提高文件系统无效空间回收能效的方法. 中国, 发明专利, 已申请 (专利权人: 华中科技大学)

软件著作权

- [1] 谭支鹏, **杨梨花**, 王芳, 涂诗云, 肖阳. 基于热度识别优化的 F2FS 段清理软件. 中国, 发明专利, 登记号: 2022SR0046575 (专利权人: 华中科技大学)

华中科技大学博士学位论文

附录2 公开发表的学术成果与博士学位论文的关系

序号	成果名称	成果形式	成果主要内容	与学位论文对应的关系
1	ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees	会议论文	使用决策树选择预留文件减少文件碎片	对应学位论文第二章
2	Improving F2FS Performance in Mobile Devices with Adaptive Reserved Space Based on Traceback	期刊论文	基于机器学习的自适应预留空间策略	对应学位论文第二章
3	一种选择性预留空间的减少碎片的方法及系统	发明专利 ZL201910681862.8	预留空间减少碎片的方法实现	对应学位论文第二章
4	M2H: Optimizing F2FS via Multi-log Delayed Writing and Modified Segment Cleaning based on Dynamically Identified Hotness	会议论文	基于数据热度的多日志延迟写与段清理优化	对应学位论文第三章
5	Improving F2FS Performance with the Hotness of Warm Data Identified by K-means Clustering	期刊论文（在投）	使用 K-means 聚类算法识别数据热度并减少开销	对应学位论文第三章
6	一种基于热度的日志结构文件系统数据管理方法	发明专利 202010705796.6	基于细分数据热度优化文件数据组织布局	对应学位论文第三章
7	Understanding and Reducing the Energy Consumption of F2FS on Smartphones	会议论文（在投）	空闲空间碎片感知的 GC 策略	对应学位论文第四章
8	一种提高文件系统无效空间回收能效的方法	发明专利（已申请）	提高无效空间回收能效的方法	对应学位论文第四章

附录 3 攻读学位期间参加的科研项目

1. 国家自然科学基金重点项目

项目名称: 大规模固态存储系统结构与技术

项目编号: No. 61832020

起止时间: 2019 年 1 月至 2023 年 12 月

担任角色: 学术骨干

2. 华为合作项目

项目名称: F2FS 冷热数据管理技术合作项目

项目编号: No. TC20220106485

起止时间: 2022 年 2 月至 2023 年 3 月

担任角色: 学术骨干

2. 华为合作项目

项目名称: F2FS 空间预留及 pagecache 智能感知技术合作项目

项目编号: No. YBN2017110084

起止时间: 2017 年 6 月至 2018 年 6 月

担任角色: 学术骨干

附录4 中英文缩写对照表

AI	Artificial Intelligence（人工智能）
CB	Cost-Benefit（成本-收益）
CP	CheckPoint（检查点）
F2FS	Flash Friendly File System（闪存友好型文件系统）
FS	Fragment Size（碎片大小）
FTL	Flash Translation Layer（闪存转换层）
FUD	File block Update Distance（文件块更新距离）
GC	Garbage Collection（垃圾回收）
GPU	Graphics Processing Unit（图形计算单元）
LBA	Logical Block Address（逻辑块地址）
LFS	Log-structured File System（日志结构文件系统）
MRUD	Most-Recently-Used Distance（最近使用距离）
NAT	Node Address Table（节点地址表）
OLTP	Online Transaction Processing（在线事务处理）
PBA	Physical Block Address（物理块地址）
POSIX	Portable Operating System Interface（可移植操作系统接口）
SIT	Segment Information Table（段信息表）
SSA	Segment Summary Area（段总结区）
SSD	Solid-State Drive（固态硬盘）
SC	Segment Cleaning（段清理）
TPS	Transactions Per Second（每秒完成事务数）
YCSB	Yahoo! Cloud Serving Benchmark（雅虎云服务负载）