

清华 大学

# 综合 论文 训 练

题目：基于开放通道 SSD 的高性能  
应用负载评测与优化

系 别：计算机科学与技术系

专 业：计算机科学与技术

姓 名：孟垂正

指导教师：舒继武 教授

2018 年 6 月 20 日

# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签 名：\_\_\_\_\_ 导师签名：\_\_\_\_\_ 日 期：\_\_\_\_\_

## 中文摘要

本文主要研究在开放通道 SSD 上利用高性能应用的 IO 负载特性优化其 IO 性能的问题。高性能应用的 IO 负载具有 IO 请求地址按页对齐，负载中覆盖写比例很高的特点。具有这种特点的 IO 序列在 SSD 上会产生大量不含有任何脏页的无效数据块，回收时可直接擦除而无需重新写入脏页。现有方法采用的页粒度的地址映射高度灵活性的优势在这种负载下无法发挥，反而徒然增加了映射表维护的开销。本文提出的混合页粒度和超级块粒度地址映射的优化方法减少了这一不必要的开销，同时能够应对负载中少量块不对齐和小于块大小的写入操作。实验表明，本文采用的混合超级块映射-贪心垃圾回收方法能够利用更小的映射表和更低的映射表维护成本达到与现有方法相近的 IO 性能。

本文的工作主要有：

- 抓取了典型高性能计算应用 LAMMPS 和 MACDRP 的 IO 负载记录并分析了二者的负载特征；
- 在用户态实现了多种 IO 优化策略与相应的评测程序；
- 将现有方法页粒度的地址映射改进为混合超级块映射，在保证 IO 性能下降不太大的同时大幅减少了映射表体积和维护开销。

**关键词：**开放通道 SSD；高性能应用；混合超级块映射

## ABSTRACT

This article focuses on optimizing the IO performance of an open channel SSD using high-performance application IO load characteristics. In the IO load of high-performance applications, the IO request address is page-aligned and the load has a high write-to-write ratio. An IO sequence with these characteristics will produce a large number of invalid data blocks on the SSD that do not contain any dirty pages, which can be erased directly with no dirty page requiring rewriting. The page-granularity address mapping used by the existing method cannot enjoy the benefit of its flexibility under such a load, but in turn increases the overhead of maintaining the mapping table. The optimization method mixing page granularity and super-block granularity address mapping proposed in this paper reduces this unnecessary overhead while dealing with some write operations not aligned to block or having a size smaller than the size of a block. Experiments show that the proposed method can use a smaller map and a lower map maintenance cost to achieve similar IO performance to the existing method.

The work of the paper includes:

- Tracking the IO load of two high-performance computing applications and analyzing characteristics of their IO records;
- Implementing multiple optimization strategies and corresponding benchmark tools;
- Modifying page-granularity address mapping in the existing method to super-block granularity address mapping, which reduces the size and the maintaining cost of the mapping table while keeping the IO performance not reducing a lot.

**Keywords:** open-channel ssd;high performance computing application;mixed super-block mapping

# 目 录

<b>第 1 章 引言 .....</b>	<b>1</b>
1.1 研究动机 .....	1
1.2 相关工作 .....	2
1.3 主要工作 .....	2
<b>第 2 章 开放通道 SSD 与高性能应用 .....</b>	<b>4</b>
2.1 开放通道 SSD 设备简介 .....	4
2.1.1 SSD 内部结构 .....	4
2.1.2 SSD 读写操作的限制 .....	4
2.1.3 开放通道 SSD 简介 .....	5
2.2 高性能应用的 IO 特征分析 .....	5
2.2.1 LAMMPS 应用的 IO 特征分析 .....	6
2.2.2 MACDRP 应用的 IO 特征分析 .....	7
2.3 本章小结 .....	9
<b>第 3 章 优化设计与实现 .....</b>	<b>10</b>
3.1 开放通道 SSD 访问 .....	10
3.1.1 访问流程 .....	10
3.1.2 开放通道 SSD 的物理地址与内部结构的对应关系 .....	12
3.2 优化策略设计 .....	12
3.2.1 页映射-贪心垃圾回收策略 (PM_GCL) .....	12
3.2.2 页映射-连续空间垃圾回收策略 (PM_GCC) .....	16
3.2.3 超级块映射-贪心垃圾回收方式 (SBM_GCL) .....	17
3.2.4 混合超级块映射-贪心垃圾回收方式 (HSBM_GCL) .....	21
3.3 本章小结 .....	23
<b>第 4 章 实验结果与分析 .....</b>	<b>24</b>
4.1 实验环境 .....	24
4.1.1 硬件环境 .....	24
4.1.2 软件环境 .....	24

4.2 IO 吞吐量 .....	25
4.3 映射表开销 .....	28
4.4 擦除次数与写放大系数 .....	29
4.5 负载均衡 .....	31
4.6 设备容量的影响 .....	32
4.7 块内页数的影响 .....	34
4.8 超级块大小的选择 .....	35
4.9 日志块数量的选择 .....	37
4.10 本章小结 .....	38
<b>第 5 章 结论与展望 .....</b>	<b>39</b>
<b>插图索引 .....</b>	<b>40</b>
<b>表格索引 .....</b>	<b>41</b>
<b>参考文献 .....</b>	<b>42</b>
<b>致 谢 .....</b>	<b>44</b>
<b>声 明 .....</b>	<b>45</b>
<b>附录 A 外文资料的调研阅读报告或书面翻译 .....</b>	<b>46</b>
A.1 引言 .....	46
A.2 背景 .....	48
A.2.1 闪存基础 .....	48
A.2.2 基于闪存的存储系统的体系结构 .....	48
A.3 基于对象的闪存转换层 .....	49
A.4 系统与闪存的协同设计 .....	51
A.4.1 反向指针辅助的惰性索引 .....	52
A.4.2 粗粒度块状态维护 .....	53
A.4.3 压缩更新 .....	54
A.5 实现 .....	56
A.6 评估 .....	57
A.6.1 实验设置 .....	58
A.6.2 总体比较 .....	58
A.6.3 元数据的写放大 .....	60

A.6.4	闪存页面大小的影响 .....	61
A.6.5	扩展更新窗口的开销 .....	62
A.7	相关工作 .....	63
A.8	结论 .....	64

# 第 1 章 引言

## 1.1 研究动机

近年来，固态硬盘（SSD，Solid-State Drive）逐渐成为各个领域占统治地位的存储器。与传统磁盘相比，固态硬盘因为没有机械结构，具有随机访问性能优异，抗震性好，噪音小，节能等优点，在个人电脑、云服务和高性能计算场景中得到了越来越广泛的应用。然而，固态硬盘自身仍然存在限制其性能进一步提高的问题。尽管固态硬盘内部的闪存芯片能够提供高性能的读写，固态硬盘用于管理这些芯片的主控程序依然不能在所有场景下充分发挥它们的性能<sup>[1]</sup>。

虽然固态硬盘的内部结构与传统块设备大不相同，其搭载的主控却能够允许操作系统像操作普通块设备一样操作固态硬盘，而主控则负责将操作系统的请求转换为对固态硬盘内部芯片的读、写和擦除操作。遗憾的是，长期以来固态硬盘内部的主控算法都由其内部的控制芯片执行，操作系统和应用程序无法得知其内部的操作细节，从而丧失了优化的机会。随着开放通道 SSD 的出现，这一问题迎来了解决的希望。开放通道 SSD 将内部结构暴露给外部应用程序，使得操作系统和应用程序参与 SSD 内部芯片的管理和承担一部分主控的工作成为可能。目前，一些顶级的云计算服务商已开始采用开放通道 SSD 改进系统性能<sup>[2]</sup>，也有一些工作尝试在 RocksDB 数据库等应用上面向开放通道 SSD 优化其 IO 性能<sup>[3]</sup>，但在高性能应用领域结合开放通道 SSD 和应用负载特点进行 IO 优化的工作还不很多。

高性能应用的 IO 有其独特的模式和特征，例如大块 IO 较多、顺序访问占绝大多数、仅添加（append-only）类型的写入为主要写入类型等<sup>[4]</sup>。与常见的操作系统和应用程序产生的 IO 序列相比，高性能应用的 IO 序列对固态硬盘的访问更为友好，这为简化 SSD 内部芯片管理机制、降低管理开销提供了优化的空间。

遵循这一思路，本论文分析了两种高性能应用的 IO 负载特征，并基于这些特征尝试在开放通道 SSD 上改进闪存管理策略，所提出的方法在保证高性能应用 IO 性能的前提下成功降低了闪存管理的开销。

## 1.2 相关工作

在开放通道 SSD 和闪存设备方面，OFTL<sup>[5]</sup> 在开放通道 SSD 上通过实现基于对象的闪存转换层减少写入放大同时延长设备的使用寿命。uFLIP<sup>[6]</sup> 和 uFLIP-OC<sup>[7]</sup> 分别针对普通 SSD 设备和开放通道 SSD 设备设计了用于评测 IO 性能的多种序列，评估了所用设备面对不同类型 IO 负载时的性能表现。LightNVM<sup>[1]</sup> 作为 Linux 上的开放通道 SSD 子系统，为 Linux 内核提供了开放通道 SSD 支持，同时提供了一个开源的闪存转换层实现 pblk，和一个用于操作开放通道 SSD 的用户态库 liblightnvm。其中 pblk 实现了读写分离、可预测的 IO 延迟和 IO 调度功能，并在多种应用场景下表现优异<sup>[8]</sup>。He 等人的工作<sup>[9]</sup> 通过分析来自真实负载的 IO Trace，揭示了希望在 SSD 上获得较高读写性能时必须遵守的规则。

在高性能应用的 IO 负载评测方面，Huong 等人的工作<sup>[10]</sup> 创建了一个能够在多个层面跟踪 IO 的框架，进而快速确定应用出现 IO 性能低效的原因。他们的另一项工作<sup>[11]</sup> 通过分析数以千计的高性能应用的 IO 行为，总结了有利于提高应用吞吐量的措施。Liu 等人的工作<sup>[12]</sup>，研究了非易失性存储器引入后对高性能应用 IO 性能的影响：缓慢存储造成的性能问题得到缓解，且缩小存储器和 CPU 之间性能差距的机制可能是不必要的。

在基于开放通道 SSD 优化特定应用的性能表现方面，大部分工作集中于数据库系统。Lee<sup>[13]</sup> 通过实证方法测试了现有数据库对闪存的利用情况，并证明在事务日志、回滚段和临时表空间替换等环节换用 SSD 能带来大幅度的性能提升。Gonzalez<sup>[3]</sup> 针对 RocksDB 数据库的特点为其设计了直接读写开放通道 SSD 的 IO 后端，优化了其 IO 性能。Wang<sup>[2]</sup> 提出了一种能更好地利用 SSD 并行性的 FOCS 系统改善了基于日志结构的合并树（LSM-Tree）的性能。

## 1.3 主要工作

本文主要研究的问题是基于开放通道 SSD，针对高性能应用的 IO 负载特征对其 IO 性能进行优化，主要工作如下：

- 抓取了典型高性能计算应用 LAMMPS 和 MACDRP 的 IO 负载记录并分析了二者的负载特征，发现其负载特征具有绝大部分请求按页对齐、单次请求数量恰好覆盖整数个页大小、覆盖写比例很高的特点；
- 在用户态实现了多种 IO 优化策略与相应的评测程序，通过吞吐量、映射表维护开销、擦除次数和写放大系数等指标评价不同 IO 优化策略的优劣；

- 根据高性能计算应用的 IO 负载特征，针对目前已有闪存转换层的映射方式进行改进，通过将映射粒度从页级别提高到超级块级别，并引入日志块实现混合映射以应对块不对齐和小于块大小的写入，在保证 IO 性能不下降的同时大幅减少了映射表体积和维护开销。

本文接下来各章的主要内容如下：第 2 章介绍了 SSD 读写的特性与开放通道 SSD 的特点，然后分析了两种高性能应用 LAMMPS 和 MACDRP 的负载特性；第 3 章按照目前已有的闪存转换层所采用的映射与垃圾回收方法实现了基本的 IO 优化策略，然后根据高性能应用的负载特征分别从垃圾回收方式与映射方式两方面提出改进的 IO 优化策略；第 4 章通过比较基本优化策略与改进的几种优化策略在重放高性能应用 IO 负载时的性能指标，包括吞吐量、映射表维护成本、擦除次数和写放大系数，证明采用混合映射的改进策略能够在保持 IO 性能的同时减少映射表大小和维护成本。第 5 章总结了本文的工作并列出了今后需要继续改进的方向。

## 第2章 开放通道 SSD 与高性能应用

### 2.1 开放通道 SSD 设备简介

#### 2.1.1 SSD 内部结构

SSD，全称为固态硬盘（solid-state drive），是一种由多个闪存芯片组成的存储设备。本文主要关注由 NAND 闪存芯片组成的 SSD，因为目前主流的 SSD 设备均采用 NAND 芯片，对 NAND 的管理与现实应用关系更加密切。

典型 SSD 设备的内部结构如图2.1所示。SSD 的控制器上并行连接了多组闪存芯片，每组闪存芯片一般包含数十个 NAND 存储芯片，并通过名为 Channel 的结构连接到控制器。每个 Channel 下的芯片按照 Die、Plane、Block 和 Page 的层级组织。一个 Die 每次只允许执行一个 IO 请求，即 SSD 并行执行 IO 请求的最小单元是 Die。在开放通道 SSD 设备中，一个或者多个 Die 也会被合并成为一个 Lun。每个 Die 含有相同数量的 Plane，同样，每个 Plane 含有相同数量的 Block，每个 Block 也含有相同数量的 Page。每个 Page 还继续被分为固定大小的 Sector，同时包含一个 OOB 区域（out-of-bound area）用于存储元数据。

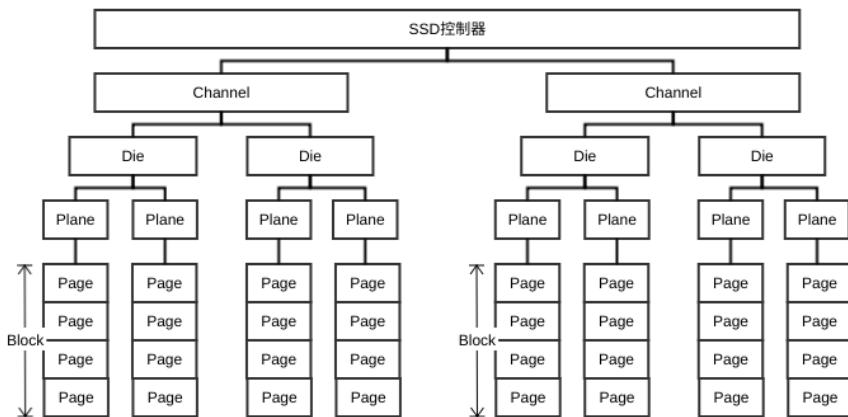


图 2.1 SSD 设备的内部结构

#### 2.1.2 SSD 读写操作的限制

和传统磁盘相比，尽管 SSD 有随机访问迅速的优势，其读写操作有一些额外的限制：

1. 任何一个页（Page）在写入或者再次写入前必须经过擦除操作。
2. SSD 读写操作的最小单位为页（Page），而擦除操作的最小单位是块（Block）。
3. 闪存芯片可被擦除的次数是有限的。对于 TLC/QLC 类型的芯片，其可经受的擦除循环数数量级在  $10^2$ ，而对于 MLC 和 QLC 芯片，这一数值的数量级分别为  $10^3$  和  $10^5$ 。

这些限制使得 SSD 设备在覆盖写入已有数据的区域时不能像传统磁盘一样就地写入。这种做法会导致被覆盖写位置所在的整个块被擦除，而为了保证未被覆盖的有效数据得到保留，位于同一块内但没有被覆盖写的数据需要先被读入缓存后在擦除完成后重新写入。这就增加了额外写操作，使得写入效率大幅下降。可行的做法是如非必要则不去真正清除被覆盖写的数据，而是将其标记为无效，同时在设备上为新写入的数据分配一块新空间。这种做法会导致同一个逻辑地址在多次写入时对应的物理地址发生变化，因此需要建立逻辑地址到物理地址的映射方式。同时，被标记为无效的数据所占用空间需要定期进行释放，否则整个设备的可写入量将远远小于实际容量，因此需要设计垃圾回收算法以回收空间。由于芯片存在最大擦除次数的限制，映射方式和垃圾回收算法需要确保设备所有块的被擦除次数尽量均匀，否则被擦除次数较多的块将在达到其最大擦除次数限制后损坏，导致设备可用空间下降。SSD 设备内部设计了闪存转换层（FTL，Flash Translation Layer）实现以上地址映射、垃圾回收和负载均衡的功能。

### 2.1.3 开放通道 SSD 简介

开放通道 SSD（Open-Channel SSD）是近年新出现的一种 SSD 设备。与传统 SSD 设备相比，开放通道 SSD 的特点是外部程序可直接控制其内部结构的读写，从而允许主机控制数据的分布和进行物理 IO 调度；而传统 SSD 设备仅对外暴露块设备的接口，外部程序很难获知其内部的闪存转换层是如何将系统对块设备的操作转换为对内部结构的操作的。

## 2.2 高性能应用的 IO 特征分析

这里选择 LAMMPS 和 MACDRP 两种高性能应用，采集它们运行一段时间内的 IO Trace 并分析其特点。由于 2 种应用运行时会向不同的主机写入数据，且经汇总发现对于 IO 操作数最多的前 10 台主机其 IO 特征高度一致，故以下分析

时只取其中操作数最多的一台主机上的 IO Trace 进行分析。

### 2.2.1 LAMMPS 应用的 IO 特征分析

LAMMPS 的全称为大规模原子/分子大量并行模拟器。它能够并行模拟原子、中观或者连续体尺度上的大规模粒子行为。

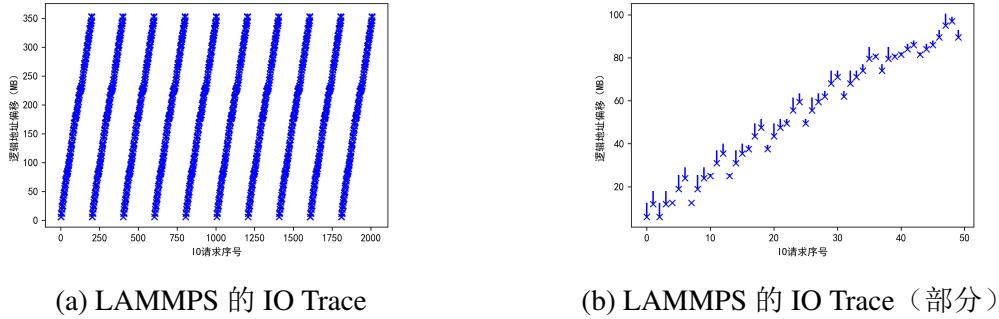


图 2.2 LAMMPS 的 IO Trace

图2.2展示了LAMMPS的IO Trace，采集到的IO Trace全部请求均为写请求。如图2.2(a)，LAMMPS对逻辑地址范围0-358MB内的空间反复写入。图2.2(b)为对其中一段IO Trace放大显示的结果，其中x表示IO请求的起始位置，铅直线覆盖部分表示IO请求涉及的逻辑地址空间。可以观察到LAMMPS对同一段空间进行了多次写操作。统计得到LAMMPS的这段IO Trace共写入了7075MB的数据，而访问的逻辑地址最大仅为358MB，显然这一过程中存在大量的覆盖写操作。

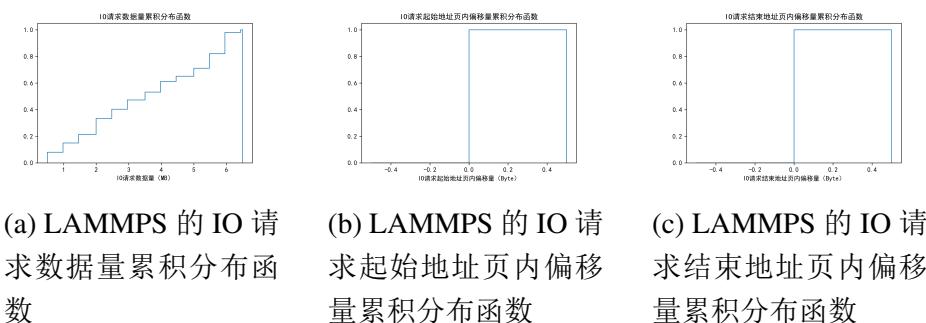


图 2.3 LAMMPS 的 IO 请求按页对齐情况

图2.3展示了LAMMPS的IO请求按页对齐的情况。这里的页大小取为后续实验使用的SSD页大小32768KB。如图2.3(a)，几乎所有的IO请求大小都大于

500KB。如图2.3(b)和图2.3(c)，全部的 IO 请求起始地址和结束地址都对齐到页，同时表明 IO 请求的大小也按页对齐。

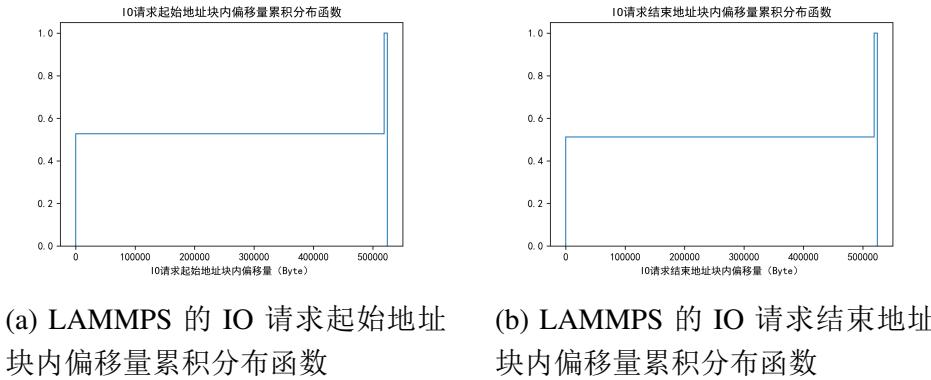


图 2.4 LAMMPS 的 IO 请求按块对齐情况

图2.4展示了 LAMMPS 的 IO 请求按块对齐的情况。这里的块大小取为后续实验使用的 SSD 块大小 1MB (32 个页)。由图2.3(a)知，约 90% 的 IO 请求数量均大于一个块。如图2.4(a)，大约半数的 IO 请求起始地址按块对齐，另一半 IO 请求的起始地址偏移为 512KB，恰好为半个块大小。如图2.4(b)，IO 请求结束地址的情况与此类似。

## 2.2.2 MACDRP 应用的 IO 特征分析

MACDRP 能够模拟地震波在具有足够精度地表地形下的传播效果。

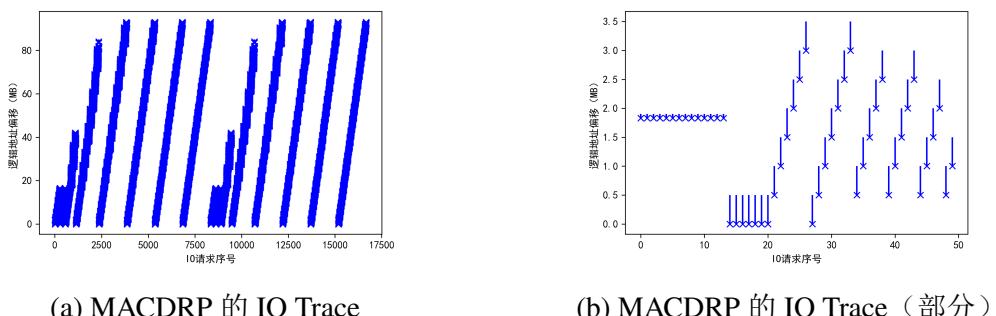


图 2.5 MACDRP 的 IO Trace

图2.5展示了 MACDRP 的 IO Trace，采集到的 IO Trace 全部请求均为写请求。与 LAMMPS 的情况类似，MACDRP 同样在有限的逻辑地址空间内进行了

大量覆盖写操作。如图2.5(a), MACDRP 在逻辑地址范围 0-93MB 内的空间反复写入了高达 8256MB 的数据。图2.5(b)为对其中一段 IO Trace 放大显示的结果, 符号含义与图2.2(b)相同。可以观察到 MACDRP 对同一段空间进行了高达 7 次写操作。这一过程中覆盖写的比例比 LAMMPS 更高。

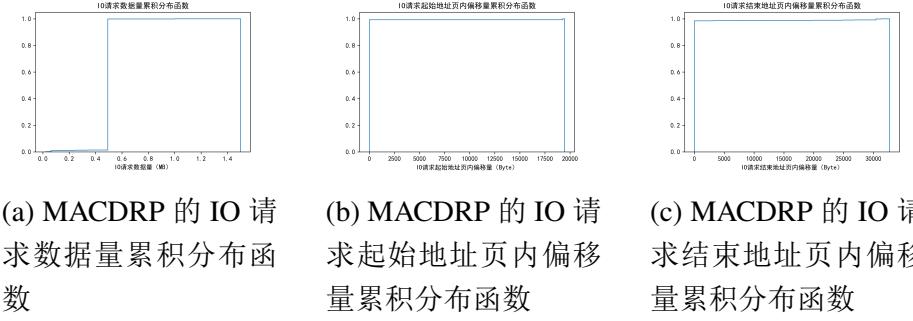


图 2.6 MACDRP 的 IO 请求按页对齐情况

图2.6显示了 MACDRP 的 IO 请求按页对齐的情况。由图2.6(b)和图2.6(c)可知, 其绝大部分 IO 请求都能按页对齐, 仅有少量请求的开始/结束地址页内偏移量不为 0。图2.6(a)表明 MACDRP 的几乎所有 IO 请求数据量大小均相同, 为 512KB。

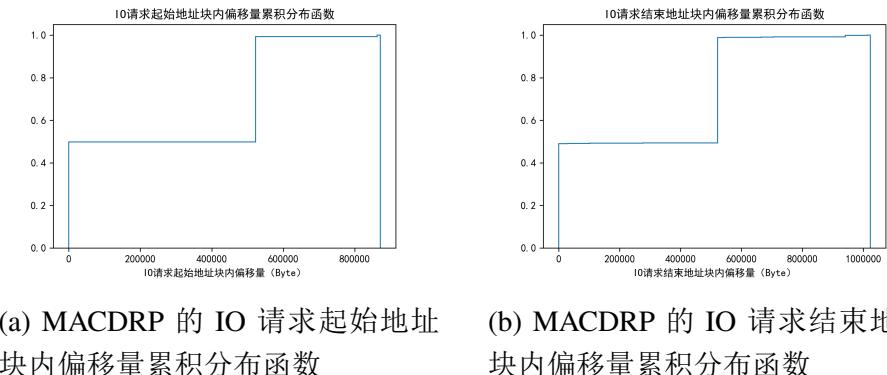


图 2.7 MACDRP 的 IO 请求按块对齐情况

图2.7展示了 MACDRP 的 IO 请求按块对齐的情况。这里的块大小取为后续实验使用的 SSD 块大小 1MB (32 个页)。由图2.6(a)知, 几乎所有的 IO 请求数据量均为 512KB, 为半个块大小。如图2.7(a), 大约半数的 IO 请求起始地址按块对齐, 另一半 IO 请求的起始地址偏移为 512KB, 恰好为半个块大小。如图2.7(b),

IO 请求结束地址的情况与此类似。

根据以上的分析，可以发现所选高性能应用的 IO 负载具有覆盖写比例高、请求按页对齐、一次请求写入的数据量恰好为整数页的特点。这样的负载在写入时必然会在过程中创造大量连续的无效数据区域，这些区域在被垃圾回收时可以直接擦除而无需重新写入脏页。根据这一特点，可以尝试简化现有闪存转换层的垃圾回收算法，或者简化其映射方式。

同时注意到 LAMMPS 与 MACDRP 两种负载的不同之处在于，LAMMPS 约有一半的写请求按块对齐，且大部分写请求数据量大于等于一个块大小；而 MACDRP 几乎所有写请求都只有半个块大小。后者的这一特性使得映射方式需要对其进行特殊处理。

### 2.3 本章小结

本章首先介绍实验所用的开放通道 SSD 设备的内部结构、读写特点以及其定制性强的优势，然后分析了 LAMMPS 和 MACDRP 两种高性能应用 IO 负载的特征，并探讨了根据其特征对闪存转换层进行优化的可能性。

## 第3章 优化设计与实现

### 3.1 开放通道 SSD 访问

#### 3.1.1 访问流程

传统 SSD 的访问方式如图3.1(a)所示。用户态程序发出的文件操作请求经内核的虚拟文件系统和物理文件系统处理后，物理文件系统将请求转换为对块设备的读写操作。传统 SSD 设备内部设置的闪存转换层（Flash Translation Layer）对外提供了与块设备兼容的接口，使得原本为块设备设计的文件系统能够在 SSD 上直接使用；闪存转换层对内负责处理逻辑地址到物理地址的映射、垃圾回收和负载均衡工作，将对块设备的请求转换为对闪存芯片的读、写和擦除操作。

开放通道 SSD 的访问方式如图3.1(b)，用户态程序可以通过 liblightnvm 库<sup>①</sup>对 SSD 内部结构进行直接访问，从而无需经过文件系统和闪存转换层。在这种访问方式下，用户态程序可以模拟出闪存转换层需要提供的映射、垃圾回收和负载均衡功能，并能够根据自身 IO 特征进行适当的优化。

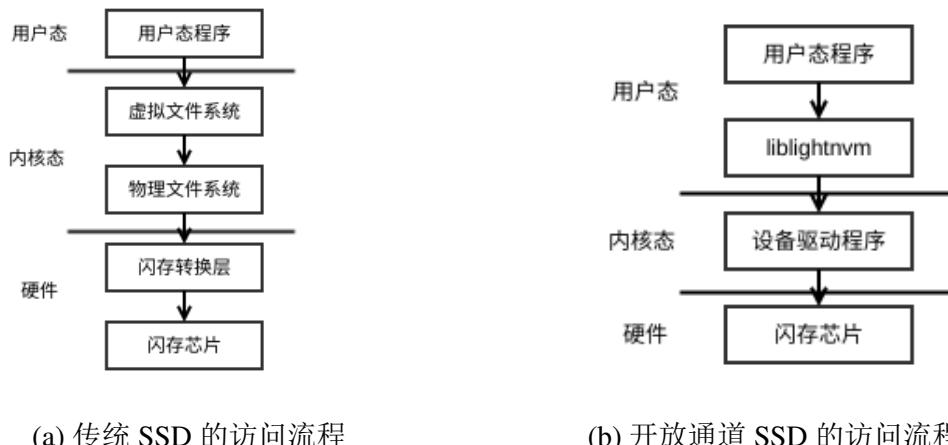


图 3.1 两种 SSD 的访问流程

具体到本文的实现，由于可供评测的数据为高性能应用运行时文件系统下发的 IO Trace，因此可以在用户态的评测程序中实现不同的优化策略，读取 IO Trace 进行重放并通过 liblightnvm 直接操作开放通道 SSD。为了便于评测时统计

<sup>①</sup> <https://github.com/OpenChannelSSD/liblightnvm>

相关信息，实现时没有直接使用 liblightnvm 提供的读写接口，而是在开源的开放通道 SSD 评测工具 fox<sup>①</sup>的基础上进行改进。fox 在 liblightnvm 提供的接口基础上增加了统计信息的记录，能够测量全过程的平均吞吐量，读/写/擦除操作的平均延迟以及精确到每页的读/写/擦除操作的大小和延迟。fox 本身只提供了几种预先定义的 IO 序列，用于测试开放通道 SSD 的连续读取、连续写入和读取写入按一定比例混合交替进行时的性能。利用这些序列可以得到所用 SSD 的基本性能如表3.1所示：

表 3.1 开放通道 SSD 的基本性能

操作	延迟（单位 us）
读一个 Page	101
写一个 Page	116
擦除一个 Block	434

由表3.1可知，SSD 的基本操作中擦除耗时远远高于读、写耗时，因此为高性能应用负载设计合适的优化策略时，应尽量避免过多的擦除操作和多余的写入操作。位于用户态的负载优化策略同样需要实现传统 SSD 的闪存转换层具有的地址映射、垃圾回收和负载均衡功能。实际上为了提高性能，闪存转换层还需要设计 IO 调度，写缓存，故障恢复等机制，并在垃圾回收等环节使用多线程提高效率。但完整实现闪存转换层的所有功能工作量太大，限于时间和精力，这里的做法是在现有方法的基础上改进地址映射和垃圾回收方式，同时注意实现负载均衡，以探索目前的方法是否有优化空间。

---

① <https://github.com/DFC-OpenSource/fox>

### 3.1.2 开放通道 SSD 的物理地址与内部结构的对应关系



图 3.2 开放通道 SSD 的物理地址与内部结构的对应关系

开放通道 SSD 按照如图3.2方式将物理地址编号与内部结构进行对应。一个用整形数表示的物理地址 PA，从低位到高位各取若干位作为业内偏移量 offset, Channel 序号 C, Lun 序号 L, Page 序号 P, Block 序号 B，则 PA 指向的开放通道 SSD 内部具体位置即为：第 C 个 Channel 的第 L 个 Lun 的第 B 个 Block 的第 P 个 Page 中从 Page 起始位置算起的第 offset 个 Byte。将 Lun 和 Channel 放在地址的低位可以使连续的地址被分配到不同的 Lun 和 Channel 上，有利于利用设备内部的并行性。

页映射使用图3.2中 Block 序号、Page 序号、Lun 序号和 Channel 序号拼接得到每个物理页的唯一 ID——物理页号。由于 Lun 和 Channel 的操作均相互独立，二者都被视为并行单元，为了对并行单元统一编号，使用图3.2中 Lun 序号和 Channel 序号拼接得到唯一的并行单元序号。

## 3.2 优化策略设计

由于目前开放通道 SSD 上开源的闪存转换层 pblk 采用页映射和贪心垃圾回收方式，并通过在各个 Channel 和 Lun 轮转选择写入和擦除块的方式实现负载均衡，故首先使用上述方式实现一个基本的优化策略，而后分别在映射方式和垃圾回收方面加以改进，同时保持原有的负载均衡机制。

### 3.2.1 页映射-贪心垃圾回收策略 (PM\_GCL)

#### 3.2.1.1 重要结构描述

表3.2描述了页映射-贪心垃圾回收策略用到的重要结构。

表 3.2 页映射-贪心垃圾回收策略 (PM\_GCL) 重要结构描述

结构名称	作用
vpg2ppg	页映射表
ppg2vpg	逆页映射表
blk_list	每个并行单元上的块分配管理结构
empty_blk	当前并行单元上的空闲块队列
active_blk	当前并行单元上的活跃写入块
non_empty_blk	当前并行单元上已写满的块队列
next_ch_lun_i	记录上次进行空间管理操作的并行单元序号的后继

vpg2ppg 为页映射表，用于完成逻辑页到物理页的映射。页映射方式的好处是映射关系较为灵活，缺点是页映射表的占用空间较大：设每个表项占用空间为  $T\text{Byte}$ ，设备每个页容量为  $P\text{Byte}$ ，则映射表所需空间为设备总容量的  $T/P$ 。取  $T = 8, P = 32768$ ，则映射表体积为整个设备容量的  $1/4096$ 。当设备容量为 4TB 时，映射表所需内存高达 1GB。页映射的另一个缺点是对映射表的更新操作较频繁，在进行覆盖写或垃圾回收操作时，物理位置发生变化的每个页的映射表项都需要更新。ppg2vpg 为逆页映射表，记录了每个物理页对应的逻辑页，用于标记物理页的分配状态，同时用于在垃圾回收过程中快速确定需重新写入的脏页对应的逻辑页号从而修改映射关系。

blk\_list 为每个并行单元上用于分配单元内部块空间的管理结构。SSD 设备的每个 Channel 和 Lun 的操作互相独立，均可视为一个并行单元。该结构通过 2 个队列和 1 个活跃块指针完成块空间的分配。empty\_blk 用于管理当前单元内尚未被写入的块（空闲块），当需要分配空间时该结构将从空闲块队列上取下一个块，并将其设置为活跃写入块 active\_blk 用于放置对该单元的所有写入请求。活跃写入块写满后会被加入 non\_empty\_blk 队列等待垃圾回收。

为了最大程度利用设备的并行性，需要尽量使连续的逻辑页被分配到不同并行单元上。next\_ch\_lun\_i 用于记录上次进行空间管理操作使用的并行单元序号的后继，用于使写入在不同单元之间进行轮转。

### 3.2.1.2 映射方式

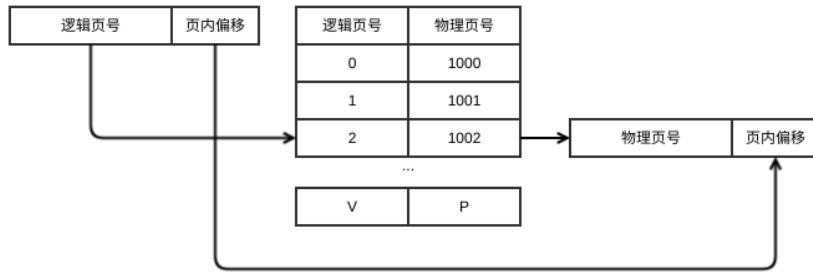


图 3.3 页映射过程示意图

这一策略的映射方式使用页映射，映射过程如图3.3所示。逻辑页号由逻辑地址/页大小得到。物理页号由物理地址/页大小得到，对应图3.2中 Block 序号、Page 序号、Lun 序号和 Channel 序号共同组成的部分。对于传入的每个逻辑地址，策略通过查找页映射表将逻辑页号转换为物理页号，页内偏移量不进行改变，从而得到对应的物理地址。

对于写请求，策略总是需要为传入逻辑地址的逻辑页号分配新的物理页号：若当前逻辑页号尚未分配物理页号则显然需要；若已分配物理页号，则当前的写操作为覆盖写，需要将该物理页数据标记为无效，然后重新分配新的物理页存放覆盖写入的数据。分配新物理页号的过程如下：从 `next_ch_lun_i` 开始遍历所有的并行单元，若遇到某个并行单元存在活跃写入块，或者不存在活跃写入块但存在空闲块则选择该单元进行空间分配。若所选单元没有活跃写入块则从空闲块队列中取出一个作为活跃写入块。若遍历所有并行单元后不存在符合条件的并行单元，则进行一次垃圾回收后重复遍历过程。遍历完成后，更新 `next_ch_lun_i` 为当前选择单元的下一单元。最后，在所选的并行单元的活跃写入块中任意选择一个空闲页作为分配的新物理页，写入数据并更新页映射表和逆页映射表。

### 3.2.1.3 垃圾回收

垃圾回收过程中回收块的选择基于贪心策略：每次均选择所有已写满块中有效数据最少的一个块进行回收，在擦除的时间消耗一定时，贪心策略可以最小化重新写入脏页消耗的时间，同时最大化释放的空间。垃圾回收时机的选择遵循懒惰原则：当前写请求需要写入的数据量大于当前可用空闲页所能容纳的

表 3.3 页映射-贪心垃圾回收策略（PM\_GCL）映射过程主要接口

接口名称	作用
vgp2ppg(lm, vpg_i)	在映射表 lm 中查询，获取逻辑页号 vpg_i 对应的物理页号
ppg2vgp(lmr, ppg_i)	在映射表 lmr 中查询，获取物理页号 ppg_i 对应的逻辑页号
isalloc(lm, vpg_i)	查询逻辑页号 vpg_i 是否已被分配物理页号
allocate_page(lm, vpg_i)	为逻辑页号 vpg_i 分配物理页号，并维护映射表

数据量时进行一次或多次垃圾回收，直到剩余空闲页能够容纳当前写入数据量为止。

垃圾回收的具体逻辑如下：首先从 `next_ch_lun_i` 开始，遍历所有逻辑单元中的已写满的块队列，记录含脏页最少的块并选择为待回收块。遍历完成后更新 `next_ch_lun_i` 为待回收块所在单元的后继单元。将待回收块内的所有脏页读入缓存并记录每个脏页所属的逻辑页号。将其移出该单元的已写满块队列，擦除该块，然后加入该单元的空闲块队列。对于读入的每个脏页，按照3.2.1.2中的空间分配方式为其分配新的物理页，写入相应数据并更新映射表。由于被垃圾回收的任何块中脏页数量都小于该块的总页数，故每个脏页在原块被擦除成为空闲块后均能被分配新的物理页。

#### 3.2.1.4 负载均衡

负载均衡主要通过以下 2 种机制实现：

物理页分配过程中，每次分配都会选择不同的并行单元上的活跃块空闲页，如此可以保证逻辑空间上连续的写入请求被均匀分布到不同的并行单元上，同时每个并行单元活跃块的选取遵循先进先出（FIFO）原则，刚刚被写满的活跃写入块只能加入写满块队列的尾部，等到其完成垃圾回收后又只能加入空闲块队列的尾部，在它第二次成为活跃写入块之前其他所有块都要完成一次擦除-写满的过程，因此不会出现同一个块被反复擦写而其他块不被访问的情况，故写负载在每个并行单元内也能够均匀分布到每个块上。

垃圾回收过程中，由于多个块同时具有最小脏页数量的情况时常出现，而垃圾回收算法总会选择其中第一个被访问的块，因此遍历时利用 `next_ch_lun_i` 指

针同样可以使得回收块的选择在不同并行单元上均匀分布。而物理页分配机制保证了每个单元中的块不会被短时间内多次写满，故擦除操作同样可以均匀分配到单元内的所有块上。

### 3.2.2 页映射-连续空间垃圾回收策略（PM\_GCC）

贪心垃圾回收需要为每个并行单元提供空间管理功能，结构略显复杂。一种可能的改进方式是模仿日志结构化文件系统<sup>[14]</sup> 的做法改进空间分配和垃圾回收方法。

#### 3.2.2.1 重要结构描述

表 3.4 页映射-连续空间垃圾回收策略（PM\_GCC）重要结构描述

结构名称	作用
vpg2ppg	页映射表
ppg2vpg	逆页映射表
used_begin_ppg	已写入区的起始物理地址
used_end_ppg	已写入区的结束物理地址

表3.4描述了该策略用到的重要结构。与页映射-贪心垃圾回收策略相比，该策略取消了对每个并行单元的空间管理，代之以划分设备空间为 2 个物理地址上连续的空间——已写入区和空闲区的做法。策略通过维护已写入区的起始物理地址 used\_begin\_ppg 和结束物理地址 used\_end\_ppg（同时分别为空闲区的结束物理地址和起始物理地址）即可实现该功能。

#### 3.2.2.2 映射方式

该策略的映射方式同样采用页映射，其地址转换机制与图3.3相同，接口形式与表3.3相同，这里不再赘述。

初始状态下设备全部空间为空闲区。每次分配新的物理页时，设备将当前已写入区所在的结束物理地址指向的物理页分配出去，同时将该物理地址数值增加一个页的大小。根据3.1.2中的对应规则，增加后的物理地址将指向下一个并行单元的相同位置。这样对每个页的写入能够被分配到不同的并行单元上。

### 3.2.2.3 垃圾回收

该策略垃圾回收时机的选择同样基于懒惰原则：当空闲区的容量小于当前写入量时进行垃圾回收。在单次写入量小于 Block 容量的情况下，一次垃圾回收往往涉及对整个设备上所有块的访问。

垃圾回收时需要遍历整个已写入区，读出脏页和所属逻辑地址，擦除完成后重新写入并更新页映射表。遍历时为了利用并行性，同样遵循并行单元->Page->Block 的遍历顺序，因此对 Block 的遍历并非逐个完成，而是一次完成并行单元数个 Block 的遍历，这要求用于存放脏页的缓存大小至少为并行单元数个 Block 大小。

### 3.2.2.4 负载均衡

从该策略的物理页分配方式可知，该策略将所有写操作均转换为对设备所有物理页的顺序写操作，因此写操作可以在所有页上均衡分布；在懒惰原则下和单次写入量不超过 Block 容量的情况下，一次垃圾回收会对设备的所有块进行擦除操作，故擦除操作同样能够均匀分布在所有块上。

## 3.2.3 超级块映射-贪心垃圾回收方式 (SBM\_GCL)

另一种改进思路是增大映射的粒度。根据之前对高性能应用负载的分析，其 IO Trace 中存在大量的按页甚至按块对齐的覆盖写，因此在覆盖写过程中很可能出现整块的数据被标记为无效、需要回收的块完全不含脏页的情况。在这种情况下，逻辑地址与物理地址之间的映射往往是以块为单位进行的，采用粒度为块的映射完全可以在达到与页映射相同效果的同时大幅减少映射表占用空间和修改开销。基于这一想法本文提出了超级块映射的改进方法，即将映射粒度从页增大到由一个或者多个块组成的超级块。

### 3.2.3.1 重要结构描述

表3.5描述了超级块映射-贪心垃圾回收策略所用的重要数据结构。

参数 PN 和 BN 定义了超级块的大小，每个超级块含有 PN 个并行单元，每个并行单元含有 BN 个块，故一个超级块含有  $PN \times BN$  个块。

该表中的其他结构与表3.2中的结构大体相同，只不过将块替换为了超级块，同时由于每个超级块可能含有多个并行单元，故每 PN 个并行单元（称为一个超并行单元）分配一个超级块管理结构而不是每个并行单元都分配一个。一个区别是超级块映射中不需要为每个并行单元设置活跃写入块，non\_empty\_sblk 队

表 3.5 超级块映射-贪心垃圾回收策略 (SBM\_GCL) 重要结构描述

结构名称	作用
PN	每个超级块包含的并行单元数
BN	每个超级块的每个并行单元包含的块数
vsblk2psblk	超级块映射表
psblk2vsblk	逆超级块映射表
sblk_list	每个超并行单元上的超级块分配管理结构
empty_sblk	当前超并行单元上的空闲超级块队列
non_empty_sblk	当前超并行单元上写入过的超级块队列
next_ch_lun_i	记录上次进行空间管理操作的并行单元序号的 后继

列不仅包含已被写满的超级块，也包含存在有效数据但尚未写满或者发生覆盖写的超级块。原因是在超级块映射下，一个超级块只要发生了覆盖写，即使该超级块尚未写满，其映射关系也需要更新，其中的所有数据被标记失效，所有脏页必须重新写入新超级块，随后该超级块中的脏页数量归零，随时可以进行垃圾回收。活跃写入块这种被写满后才可能被垃圾回收队列的机制在这里是不存在的。

### 3.2.3.2 超级块地址结构

超级块这一概念在开放通道 SSD 中并无实际物理结构对应，只是改变了块的组织方式，为了方便起见在此重新定义引入超级块后物理地址与开放通道 SSD 内部结构的对应关系。



图 3.4 引入超级块概念后物理地址与开放通道 SSD 内部结构的对应关系

对于一个 Channel、Lun、Block、Page 数分别为 C、L、B、P 的开放通道 SSD，使用参数为 PN、BN 的超级块映射后物理地址与 SSD 内部结构的对应关系如图3.4所示。一个物理地址 PA 可以从低位到高位依次划分为页内偏移量 offset，超级块内并行单元序号 inner\_pu，Page 序号 p，超级块内 Block 序号 inner\_blk，物理超级块超 Block 序号 outer\_blk，物理超级块超并行单元序号 outer\_pu。该物

理地址指向的 SSD 内部位置可按表3.6计算。在这种对应方式下，相邻的物理地址会对应到同一个超级块中，而在遍历同一个超级块内的地址时，遍历顺序仍然是按照并行单元->Page 序号->Block 序号进行的，即在同一个超级块内连续的物理地址仍然会分布在不同的并行单元上。

表 3.6 引入超级块概念后物理地址到开放通道 SSD 内部结构的转换方法

参数	值
Channel 序号	(inner_pu + outer_pu × (C × L/PN)) mod C
Lun 序号	(inner_pu + outer_pu × (C × L/PN)) ÷ C mod L
Block 序号	inner_blk + outer_blk × (B/BN)
Page 序号	p
页内偏移量	offset

### 3.2.3.3 映射方式

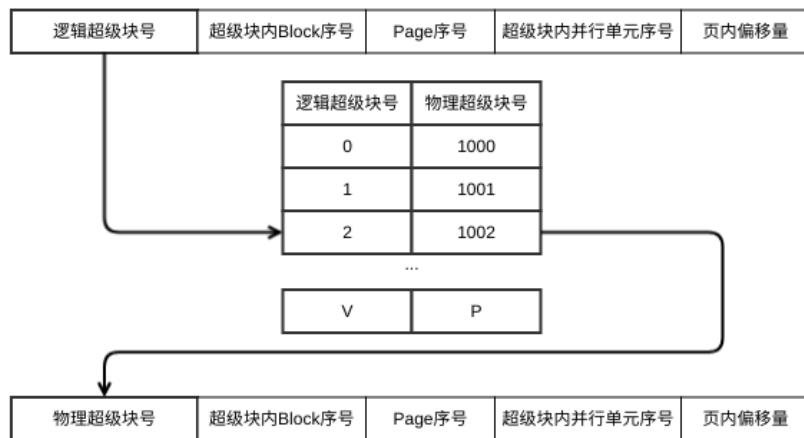


图 3.5 超级块映射过程示意图

超级块映射与页映射的区别仅在于映射粒度，故图3.5中的地址映射过程与页映射类似。

对于写请求，该策略首先遍历写请求涉及到的逻辑页，尝试为每个逻辑页所在的逻辑超级块分配可用的物理超级块映射。若该逻辑超级块已经存在物理

超级块映射，且逻辑页映射后的物理页可直接写入，则不改变映射关系；若该逻辑超级块不存在物理超级块映射或者存在映射但逻辑页对应的物理页为脏页，则需要为当前逻辑超级块分配新的物理超级块。分配流程与3.2.1.2中分配物理页的过程类似：遍历所有超并行单元的分配管理结构，从空闲超级块队列中取出一个成员用于分配。若为写入脏页的情况，分配后除了写入当前写请求的数据和维护超级块映射表外，还要将原超级块的所有页标记为失效，并将其中未被覆盖写的脏页也写入新的超级块。

由于逻辑超级块和物理超级块是一一对应的关系，在不存在映射而进行分配的情况下策略必然能找到一个空闲超级块。但在待写入页为脏页而进行分配的情况下有可能出现无空闲超级块可用的情况，这时的处理方法有两种：一种是在读出未被覆盖写的脏页后直接擦除当前超级块，然后就地写入要写的数据和脏页；另一种方式是尝试进行垃圾回收，若回收得到超级块则继续按照分配成功的情况处理，若失败再回到第一种做法就地写入。第一种做法的优势是无需修改映射表，但由于映射表得不到更新，被擦除超级块的选择将完全取决于写请求访问的逻辑地址；第二种做法则更有利于负载均衡。

#### 3.2.3.4 垃圾回收

由于引入超级块概念后物理地址与开放通道 SSD 内部结构的对应关系发生变化，连续空间垃圾回收方式中的已写入区和空闲区在引入超级块的对应方式下不再是物理地址上连续的区域，故采用超级块映射的优化策略无法使用连续空间垃圾回收，仍然采用基本优化策略的贪心垃圾回收方式。

垃圾回收的逻辑与页映射-贪心垃圾回收策略基本一致，区别仅在于将回收单位由块换成了超级块。一个简化的地方是由于超级块映射中需要回收的超级块内不存在脏页，故选择待回收超级块时可以直接选择脏页数为 0 的超级块而无需遍历所有待回收超级块获取脏页数量的最小值。

#### 3.2.3.5 负载均衡

在贪心垃圾回收策略下，本优化策略具有与页映射-贪心垃圾回收策略相同的机制保证负载均衡。此外，3.2.3.3中提到在写入页为脏页需要进行超级块分配但无空闲块可用时，先尝试垃圾回收后分配新超级块的方式与就地写入相比更有利负载均衡，原因是该方式能够保持映射表的更新，使得相同的逻辑超级块地址多次访问时都能对应到不同的物理超级块地址，从而实现负载均衡。

### 3.2.4 混合超级块映射-贪心垃圾回收方式 (HSBM\_GCL)

尽管超级块映射可以减少映射表的开销，但当负载中存在没有按块对齐或小于块的写入时将可能引发对超级块的读取-擦除-写入操作。例如对于一个已经写满数据的超级块，当对该块内容进行小于一个超级块大小的覆盖写时，若当前已经没有其他空闲块可用，则该超级块未被覆盖写的部分也需要先被读入至内存，待垃圾回收清理出新的空闲超级块后这部分内容又要被重新写入，造成不必要的写放大。为解决这一问题，混合映射方法<sup>[15]</sup> 将空间划分为应用页映射和块映射的区域。这里基于这一思路将上文的超级块映射和页映射结合，使用混合超级块映射方法解决问题。

#### 3.2.4.1 重要结构描述

表 3.7 超级块映射-贪心垃圾回收策略 (SBM\_GCL) 重要结构描述

结构名称	作用
PN	每个超级块包含的并行单元数
BN	每个超级块的每个并行单元包含的块数
vsblk2psblk	超级块映射表（数据块）
psblk2vsblk	逆超级块映射表（数据块）
lbpm	日志块映射表
sblk_list	每个超并行单元上的超级块分配管理结构
empty_sblk	当前超并行单元上的空闲超级块队列
non_empty_sblk	当前超并行单元上写入过的超级块队列
next_ch_lun_i	记录上次进行空间管理操作的并行单元序号的 后继

表3.7描述了混合超级块映射-贪心垃圾回收策略所用的重要数据结构。与表3.5相比，主要区别是增加了日志块映射表 lbpm，另外原来的超级块映射表和逆超级块映射表这里只用于映射数据块，但其结构与原来相同。

日志块映射表中每个表项包括逻辑超级块号，对应日志块的物理超级块号，以及一个超级块内页映射表，用于将超级块内的一个逻辑页映射到其中任意一个物理页。与页映射相比，由于每个超级块内的页数量远远小于整个设备的页数量，故这里的页映射表每项可以用更少的字节存储，例如 2Byte。

### 3.2.4.2 映射方式

混合超级块映射方式下，设备中的所有超级块分别属于日志区和数据区，其中日志区内的每个超级块在块内使用页映射，数据区仍然使用超级块映射。对于任意一个逻辑超级块，其最近的写入操作存储在日志超级块中，原始数据则存储在数据超级块中。

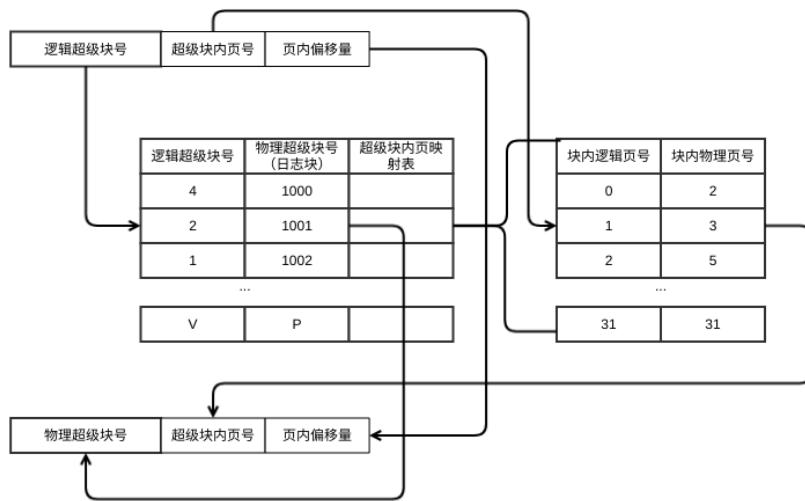


图 3.6 日志块映射过程示意图

如图3.6，经过日志块的地址映射过程如图。首先在日志块映射表中根据逻辑超级块号找到对应的日志块，然后在该日志块的块内页映射表中根据超级块内页号找到对应的物理超级块内页号，同时页内偏移量不变，得到物理地址。若以上步骤中任何一步出现表项缺失，表明当前访问的地址还在数据块中，此时应当按照图3.5的方式在数据块映射表中进行映射。

写入操作均在日志块上进行，写入一页的流程如下。首先在日志块映射表中找到当前要写入的逻辑超级块对应的表项，若不存在则分配一个新表项和一个空闲超级块作为其日志块，若日志块映射表已满则对其中某项执行一次下文提及的合并操作腾出空间。在日志块中分配一个空闲页用于存储当前要写入的页，若已经没有空闲页则执行一次合并操作，然后分配一个空闲块作为新日志块。写入后将逻辑地址中块内页号和写入的地址的块内页号对应关系记录在当前表项的块内页映射表中。

合并操作用于将某个逻辑超级块对应的日志超级块中记录的操作应用到其

数据超级块上。分配一个新的空闲超级块作为新的数据超级块存储合并后的数据，对于其中的每个页，若其在块内页映射表中有记录，则表明此处的数据经过了修改，应读出日志块中的对应页后写入，若无记录则表明该处数据未被修改，故应从数据块中读出后写入。特别地，若合并时发现日志块的块内页映射表恰好为全等映射，即序号为  $i$  的页映射后序号仍然为  $i$ ，则可以直接将当前日志块作为合并后的数据块。合并完成后原日志块和数据块内所有数据被标记无效，数据块映射表更新，日志块映射表对应表项被清空。

#### 3.2.4.3 垃圾回收

垃圾回收与超级块映射-贪心垃圾回收策略类似，仍然采用懒惰原则，当没有空闲超级块可用于分配时进行垃圾回收。由于每次合并操作都会产生至少 1 个脏页数量为 0 的超级块，故在贪心原则下的垃圾回收同样不需要重新写入脏页。

#### 3.2.4.4 负载均衡

该方法中空闲块的分配机制与超级块映射-贪心垃圾回收策略相同，故其负载均衡也由这一机制保证，此处不再赘述。

### 3.3 本章小结

本章首先根据开放通道 SSD 访问特点确定了优化策略的实现层次，然后使用目前已有的开源闪存转换层 pblk 采用的地址映射、垃圾回收和负载均衡策略实现了最基本的优化策略。随后本章分别从垃圾回收算法和地址映射粒度两方面对基本策略尝试进行了改进，同时注意保持基本策略能够实现负载均衡的特点。

## 第 4 章 实验结果与分析

实验将抓取的高性能应用的 IO Trace 通过映射和垃圾回收机制转换为对开放通道 SSD 内部结构操作，并使用 liblightnvm 对其进行直接读写，最后通过吞吐量、映射表修改次数、擦除次数和写放大系数等性能指标评价优化方法的优劣。此外，实验还研究了不同优化方法下剩余空间大小对 IO 性能的影响，以及超级块方法中超级块参数的影响。本章中的 PM\_GCL, PM\_GCC, SBM\_GCL 和 HSBM\_GCL 分别指页映射-贪心垃圾回收策略、页映射-连续空间垃圾回收策略，超级块映射-贪心垃圾回收策略和混合超级块映射-贪心垃圾回收策略。除特殊说明外，超级块映射和混合超级块映射使用的超级块大小为 1 个块，混合超级块映射中的日志块数量为 8。

### 4.1 实验环境

由于硬件条件限制，本章中所有实验均在开放通道 SSD 的模拟器上进行。因此，下文的“硬件环境”包括运行模拟器的设备的硬件环境，模拟器以及模拟出的开放通道 SSD 的基本参数。下文的“软件环境”包括模拟器使用的 Linux 内核版本与相关用户态库的信息。

#### 4.1.1 硬件环境

实验使用一台 Linux 服务器运行模拟器，服务器的 CPU 为 Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz，内存大小 16GB。模拟器使用 <https://github.com/DFC-OpenSource/qemu-ox> 提供的支持模拟开放通道 SSD 的 qemu 虚拟机。除特殊说明外，实验使用的模拟开放通道 SSD 设备基本信息如下表：

#### 4.1.2 软件环境

模拟器运行的 Linux 内核版本为 4.17.0，编译时打开了 CONFIG\_BLK\_DEV\_NVME, CONFIG\_NVM, CONFIG\_NVM\_DEBUG, CONFIG\_NVM\_PBLK 选项以提供对开放通道 SSD 的驱动支持。实验使用的用于直接操作开放通道 SSD 设备的用户态库 liblightnvm 版本为 master@ba201ca。

表 4.1 开放通道 SSD 的基本信息

参数名	数值
Channel 数量	8
每个 Channel 的 Lun 数量	4
每个 Lun 的 Block 数量	16
每个 Block 的 Page 数量	32
每个 Page 的 Sector 数量	4
Plane 数量	2
每个 Sector 能够容纳的数据量	4096 Byte
每个 Page 能够容纳的数据量	32768 Byte

## 4.2 IO 吞吐量

图4.1和表4.2展示了不同优化方法重放高性能应用的 IO Trace 时的 IO 吞吐量。可以发现，在重放 LAMMPS 和 MACDRP 两种应用的 Trace 时，页映射-贪心垃圾回收方法在两种 Trace 上吞吐量均最高，页映射-连续空间垃圾回收的 IO 吞吐量偏低。对于 LAMMPS 负载，超级块映射-贪心垃圾回收和混合超级块映射-贪心垃圾回收方法性能劣势不大；对于 MACDRP 负载，超级块映射方法性能劣势明显，混合超级块映射则略好。原因是页映射-贪心垃圾回收的方式最灵活且垃圾回收带来的额外写入最少，与之相比页映射-连续空间垃圾回收在垃圾回收时需要将有效数据全部读取和重新写入，大大增加了成本。LAMMPS 负载中单次写操作数据量大部分在一个块容量以上，块不对齐造成的影响仅限写入范围的头尾，造成的性能损失较小；MACDRP 含有大量小于一个块的写入，与 LAMMPS 相比相同写入量下块不对齐的影响更大，故造成的性能损失更明显。

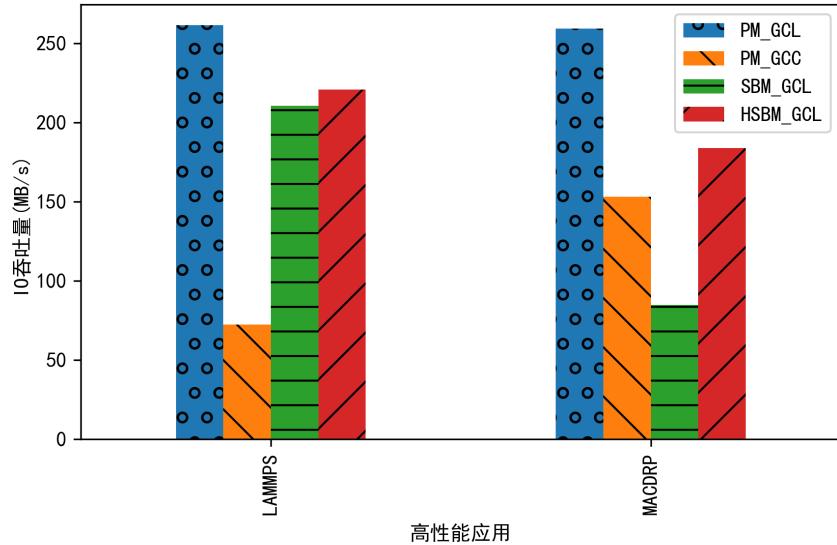


图 4.1 不同优化方法的 IO 吞吐量

表 4.2 不同优化方法的 IO 吞吐量 (MB/s)

高性能应用	优化方法			
	PM_GCL	PM_GCC	SBM_GCL	HSBM_GCL
LAMMPS	261.487	72.324	210.354	220.670
MACDRP	259.303	153.016	84.621	183.848

为了更细致地刻画几种方法在 IO 吞吐量上的表现，以下从平均 IO 吞吐量在重放过程中的变化和单次 IO 吞吐量的分布来分析不同方法下的 IO 过程。图4.2反映了重放过程中平均 IO 吞吐量随 IO 请求总量的变化过程。几种方法在早期均倾向于占据未分配的空间进行写入，当 SSD 上几乎所有空间均无法写入时开始进行垃圾回收，平均 IO 吞吐量开始下降并最终稳定到某个水平。对于页映射-连续空间垃圾回收方法，由于空闲区域写满后需要在全盘范围内进行一次垃圾回收，需要擦除的块和重写的脏页遍布整个 SSD，因此若当前 IO 无需进行垃圾回收则能够以顺序写的方式保持较高速率写入，若需要进行垃圾回收则要等待较长时间导致平均 IO 吞吐量突然下降。其他几种方法每次进行垃圾回收均只需要回收一个块或者数个块的空间，且单次回收空间量的上限被本次 IO 的写入量所限制，故每次垃圾回收需要重新写入的脏页数量和需要擦除的块数量均

相对较小。在这种机制下 SSD 上可直接写入的空间会长期维持在较低水平，每次写入均需进行小规模的垃圾回收，平均 IO 吞吐量受单次 IO 的影响较小。

连续空间垃圾回收每次均需要将当前有效数据重新写入，而贪心垃圾回收则未必。故连续空间垃圾回收的总成本更高，平均 IO 吞吐量更低，且在写入有效数据更多的 LAMMPS 的 Trace 上表现更差；而贪心垃圾回收的表现不受 Trace 写入的有效数据量影响。

对于 MACDRP 这种包含较多小于块大小写入的 Trace，超级块映射方法下同一个块的有效数据会被大量重复写入，平均 IO 吞吐很快下降到较低水平；混合超级块映射下日志块为不对齐的写入提供了更灵活的页映射，能够减少性能损失。

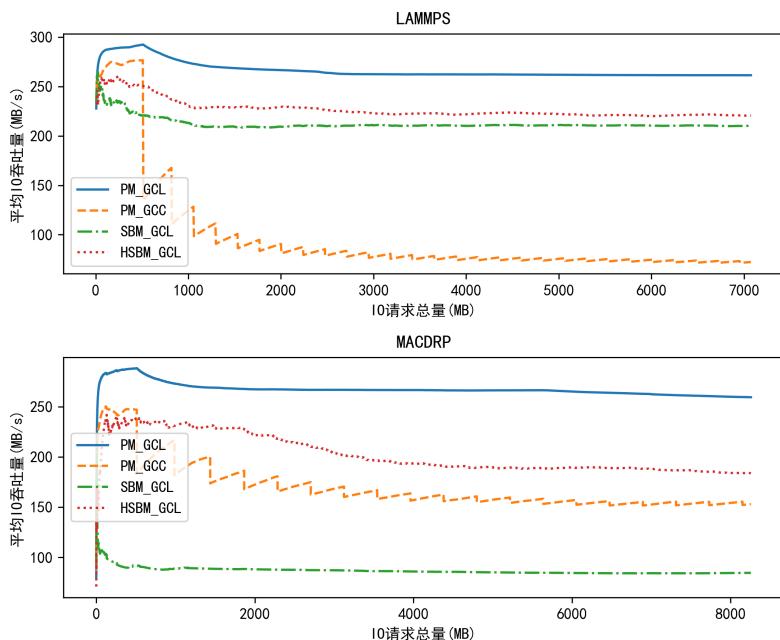


图 4.2 平均 IO 吞吐量在重放过程中随 IO 请求总量的变化

图4.3按照 IO 请求发出的先后顺序绘制了每次 IO 的吞吐量。连续空间垃圾回收方法尽管有大量写请求均能按照连续写速率（约 300MB/s）写入，但每次发生垃圾回收时需要的开销过大，导致发生垃圾回收的写请求吞吐量接近 0，整体吞吐量表现反而不如贪心垃圾回收方法。超级块映射-贪心垃圾回收方法在遇到小于块大小的写入时会重写块内其他无关数据，因此在 MACDRP 上大部分写入的速率很低。混合超级块映射-贪心垃圾回收方法一定程度上缓解了这一问题，但合并操作同样带来了部分多余的写入，导致一部分写入操作的速率也偏低。

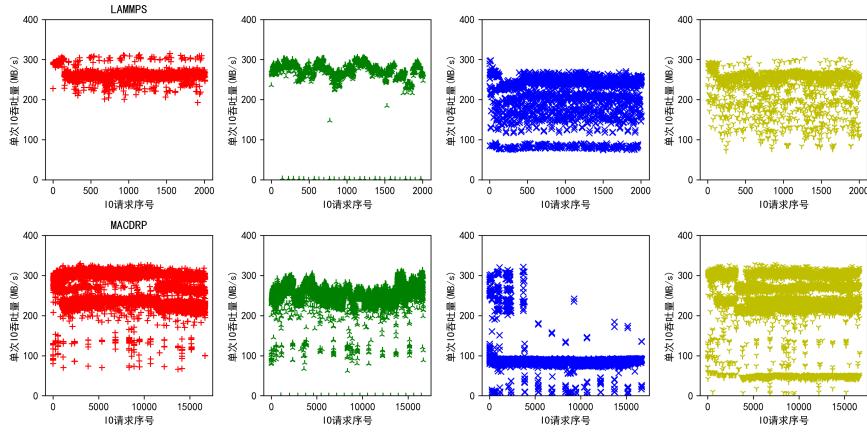


图 4.3 不同优化方法在重放过程中的单次 IO 吞吐量，每行从左至右分别代表页映射-贪心垃圾回收方法、页映射-连续空间垃圾回收方法，超级块映射-贪心垃圾回收方法和混合超级块映射-贪心垃圾回收方法

### 4.3 映射表开销

超级块映射和混合超级块映射方法与页映射方法相比，其显著优势在于映射表占用空间更小，且维护映射表需要的成本更低。在映射表每个表项使用 64 位整形数存储对应的物理地址，占用空间为 8Byte 的条件下，对于实验用表4.1所示设备，页映射需要的映射表空间为  $8 \times 4 \times 16 \times 32 \times 8\text{Byte} = 128\text{KB}$ ，占设备全部可用空间的  $1/4096$ 。而使用超级块大小为一个块的超级块映射，映射表所需空间为  $8 \times 4 \times 16 \times 8\text{Byte} = 4\text{KB}$ ，占比仅为  $1/131702$ 。使用超级块大小为 1，日志块数量为 8 的混合超级块映射，每个块内页表项大小为 2Byte 的情况下，映射表所需空间为  $8 \times 4 \times 16 \times 8 + 8 \times (2 \times 8 + 32 \times 2)\text{Byte} = 4.625\text{KB}$ ，占比仅为  $1/113379$ 。这两种映射的表项更少，维护起来也更加简单。

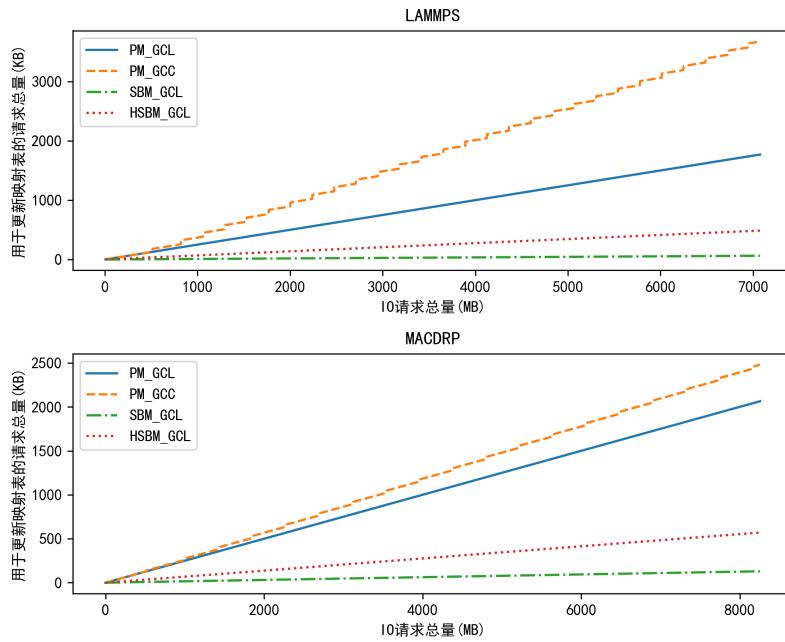


图 4.4 用于更新映射表的请求总量在重放过程中随 IO 请求总量的变化

如图4.4，在 LAMMPS 负载下，页映射-贪心垃圾回收方法和页映射-连续空间垃圾回收方法每完成 7G 的 IO 写请求分别需要传输 1768KB 和 3674KB 的数据用于映射表更新，而超级块映射方法和混合超级块映射方法仅需要 63KB 和 487KB；在 MACDRP 负载下，这一维护成本对于两种页映射方法分别为每 8GB 2065KB 和 2481KB，对于超级块映射和混合超级块映射方法则仅为每 8GB 130KB 和 571KB。

#### 4.4 擦除次数与写放大系数

在设备自身的读写性能一定的情况下，映射算法和垃圾回收算法造成的擦除与写放大是影响整体吞吐量的最主要因素。这里以 IO 请求总量为变量，分析重放过程中擦除次数与写放大系数的变化。这里擦除次数定义为从重放开始擦除的 Block 总数，写放大系数定义为  $\frac{\text{向设备写入的总数据量}}{\text{完成的写请求总数据量}}$ 。

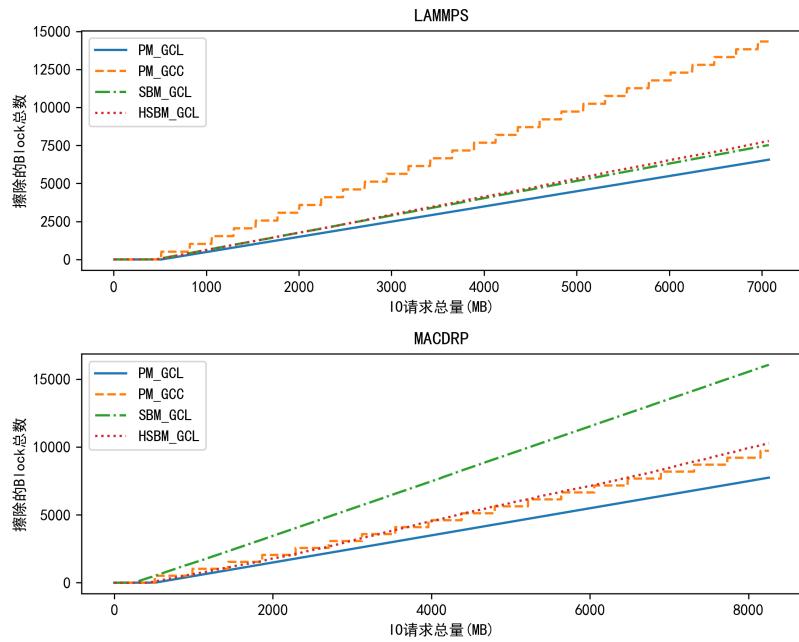


图 4.5 擦除的 Block 数在重放过程中随 IO 请求总量的变化

如图4.5，对于 LAMMPS 负载，页映射-贪心垃圾回收方法，超级块映射-贪心垃圾回收方法和混合超级块映射方法的擦除次数相近且与 IO 请求总量的关系近似线性变化。这是因为它们每次进行垃圾回收需要擦除的块数均受到当前 IO 写入量的控制，应用每次 IO 请求的写入量与方法本身无关；设备可直接写入的空间稳定在较低水平，每次写入均需要进行小范围的垃圾回收。页映射-连续空间垃圾回收方法的擦除次数呈现阶梯状变化，且写入有效数据较多的 LAMMPS 应用下擦除次数变化的间隔更小，原因是该方法每次垃圾回收后会腾出设备上除已写入有效数据的所有空间用于写入，待这段空间写满后才会再次进行垃圾回收，故两次垃圾回收之间有较大间隔；写入的有效数据越多，腾出的空间越少，垃圾回收间隔越小。

对于 MACDRP 负载，其他方法受小于块大小的写入影响较小，而超级块映射-贪心垃圾回收方法对于每个小于块大小的覆盖写在空闲块用尽后都要通过擦除获取新块，故擦除量最高。

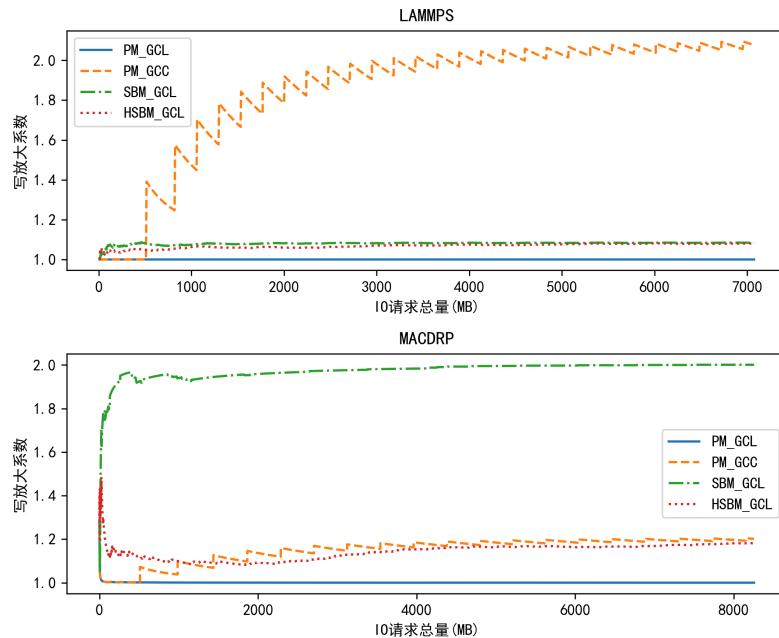
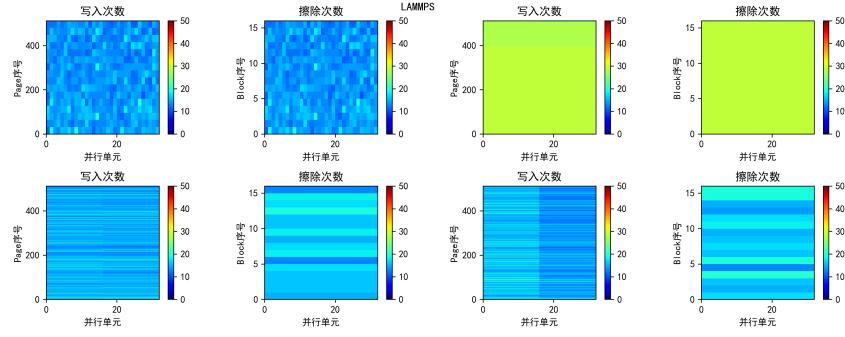


图 4.6 写放大系数在重放过程中随 IO 请求总量的变化

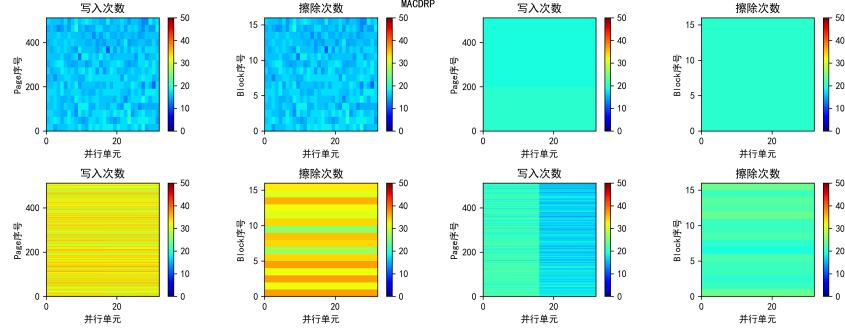
如图4.6，对于 LAMMPS 负载，几种贪心垃圾回收方法的写放大系数随着写入量增加很快稳定在 1 左右，而连续空间垃圾回收方法的写放大系数逐渐增加并维持在大于 1 的水平。因为负载一次连续写入的量大部分能覆盖一个或者多个 Block，覆盖写的起始位置和大小与第一次写请求往往相同，因此每次覆盖写都会造成大量的整块 Block 中的数据无效。对于贪心垃圾回收，其只需要对相应块进行清除操作而无需重新写入大量脏页；对于连续空间垃圾回收，已写入的所有脏页均需要读出后重新写入，造成写放大系数偏高。对于 MACDRP 负载，其中包含大量小于块大小的写，超级块映射-贪心垃圾回收方法此时不得不进行大量重复写操作导致写放大系数在此种负载下偏高。

## 4.5 负载均衡

这里通过可视化设备上每个 Page/Block 执行写操作/擦除操作的次数比较不同方法的负载均衡情况。



(a) 重放 LAMMPS 时的负载分布



(b) 重放 MACDRP 时的负载分布

图 4.7 负载分布图，每种应用的负载分布图的左上、右上、左下、右下各 2 张图分别代表页映射-贪心垃圾回收方法、页映射-连续空间垃圾回收方法、超级块映射-贪心垃圾回收方法和混合超级块映射-贪心垃圾回收方法

如图4.7，页映射-连续空间垃圾回收方法的负载均衡效果最好，所有位置的写入次数和擦除次数均几乎相同，原因是该方法在设备的几乎所有页被写入后才进行垃圾回收，而垃圾回收会对设备的所有块进行擦除，整个设备不断循环全部写入-全部擦除的过程。改进后的超级块映射-贪心垃圾回收方法强制要求每次覆盖写选择另一位置的超级块而非之前选择的超级块写入，强制改变了映射关系，从而使得擦除的块的选择与写请求的起始位置无关，改善了负载均衡。

## 4.6 设备容量的影响

设备容量的大小可能对映射表维护和垃圾回收的开销产生影响。这里通过改变每个 Lun 含有的 Block 数量改变设备容量，使得设备容量分别为写入的有

效数据量的 2 倍、4 倍和 8 倍，观察这一变化对吞吐量、映射表修改次数、擦除次数和写放大系数等性能指标的影响。

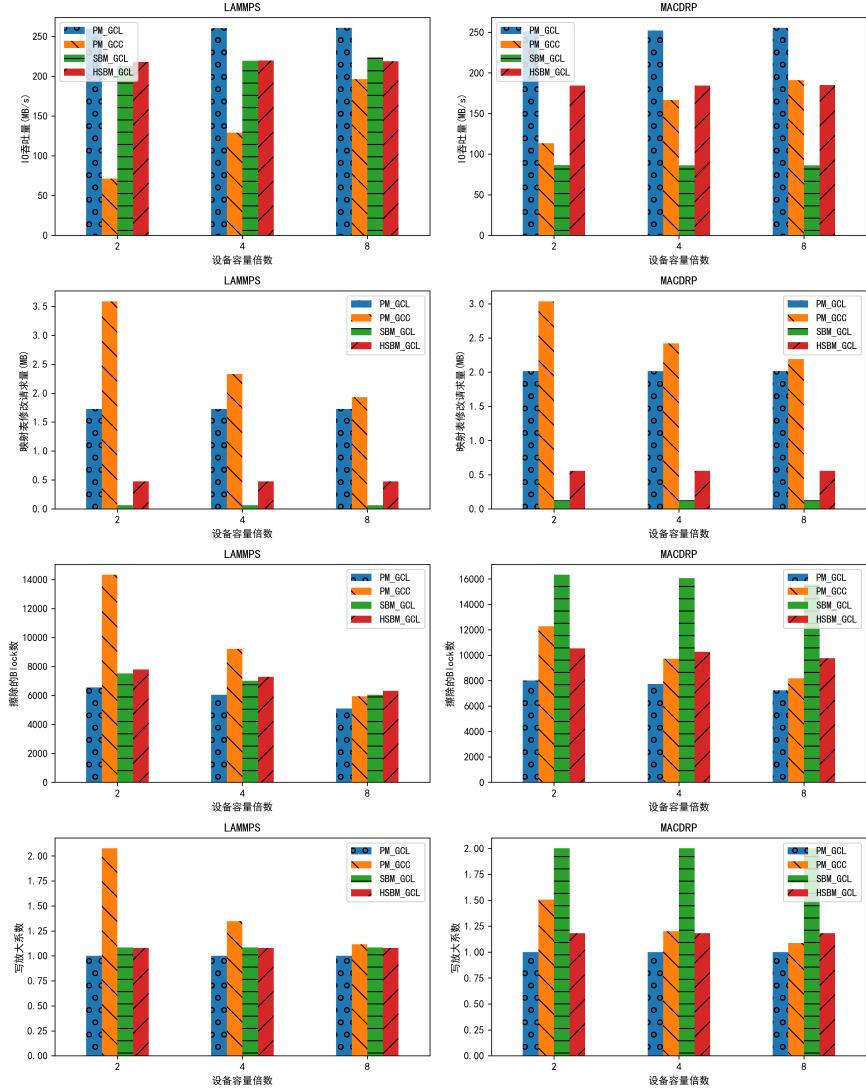


图 4.8 设备容量对各项性能指标的影响

如图4.8，几种采用贪心垃圾回收的方法在设备容量增大时性能提高很小或基本不变，原因是设备容量增大只能延长第一次进行垃圾回收前连续写入的阶段，而这一阶段虽然无需进行垃圾回收，性能最好，但持续时间占整个重放过程的比例很低。达到稳定状态后它们都只在设备上维持很少的可用空间，开销与设备容量无关。对于页映射-连续空间垃圾回收方法，设备容量的增大增加了两次垃圾回收的间隔，尽管一次垃圾回收需要擦除的块数量更多，但垃圾回收次

数减少和写放大系数降低带来的好处更大，故该方法在设备容量增大时性能提高较多。

## 4.7 块内页数的影响

块内页数变少后，负载中块不对齐和小于块大小的写入比例会下降，对不同方法的性能也会产生影响。这里设置每个 Block 含有的 Page 数量分别为 8,16,32 和 64，观察不同设置下各种方法的性能。

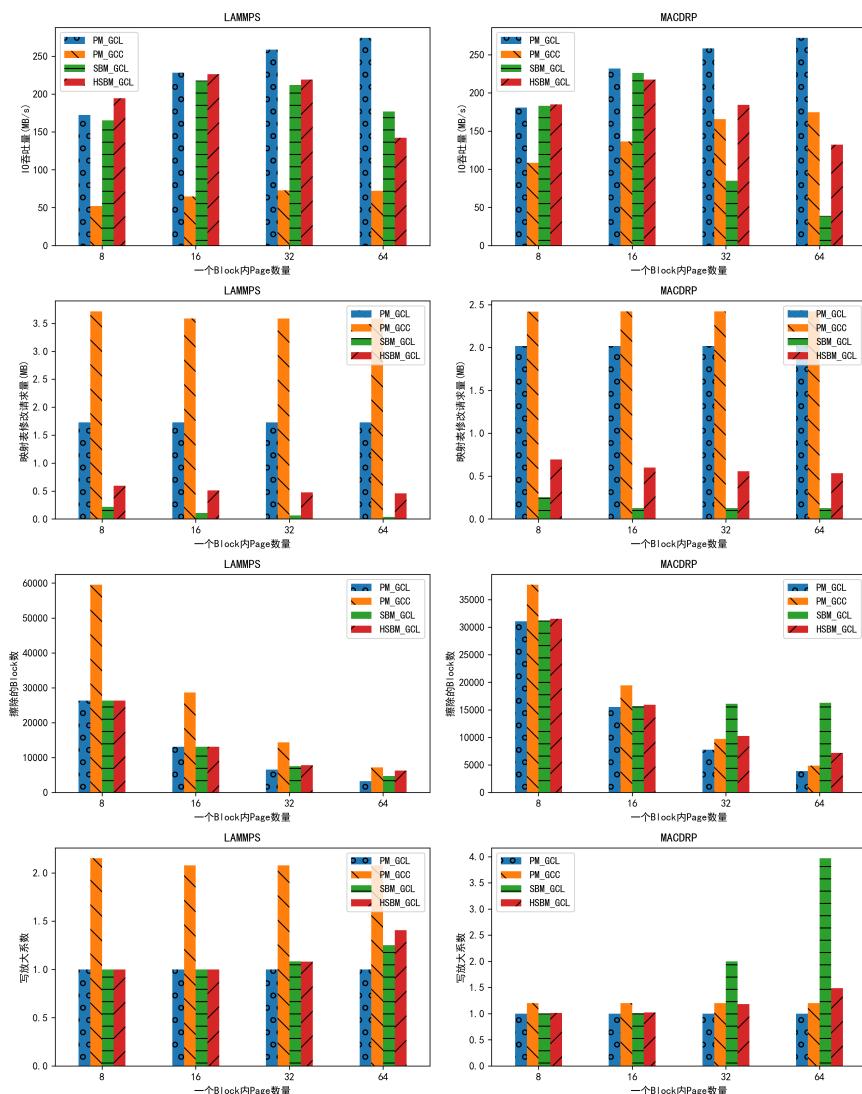


图 4.9 块内页数对各项性能指标的影响

如图4.9，对于页映射-贪心垃圾回收和页映射-连续空间垃圾回收方法，随着

块内页数的增大，两种方法的性能都是单调上升的。原因是页映射下映射的粒度小至页级别，每块内的页数对建立映射影响不大；但擦除成本会随块内页数上升而下降：单次擦除可以清除更多的无效数据，获得更多的可用空间。而对于超级块映射和混合超级块映射，虽然擦除成本同样符合上述规律，但块内页数的增加同时会导致覆盖写大小不足块大小时需要重新写入的数据量增加，带来更大的写放大系数。两种因素综合作用导致这两种方法的性能随着块内页数的增加先上升后下降，实验中块内页数为 16 时两种方法的性能最好。

## 4.8 超级块大小的选择

超级块映射涉及到设置为超级块设置合适的大小。较大的超级块能够显著减小映射表体积和维护成本，但发生覆盖写时需要擦除和重新写入的脏页数量也更多；较小的超级块则反之。这里通过调整超级块的大小为 1、2、4、8 个块，比较不同设置下超级块映射-贪心垃圾回收方法和混合超级块映射-贪心垃圾回收方法的性能指标。这里每个块内的页数设为 8。

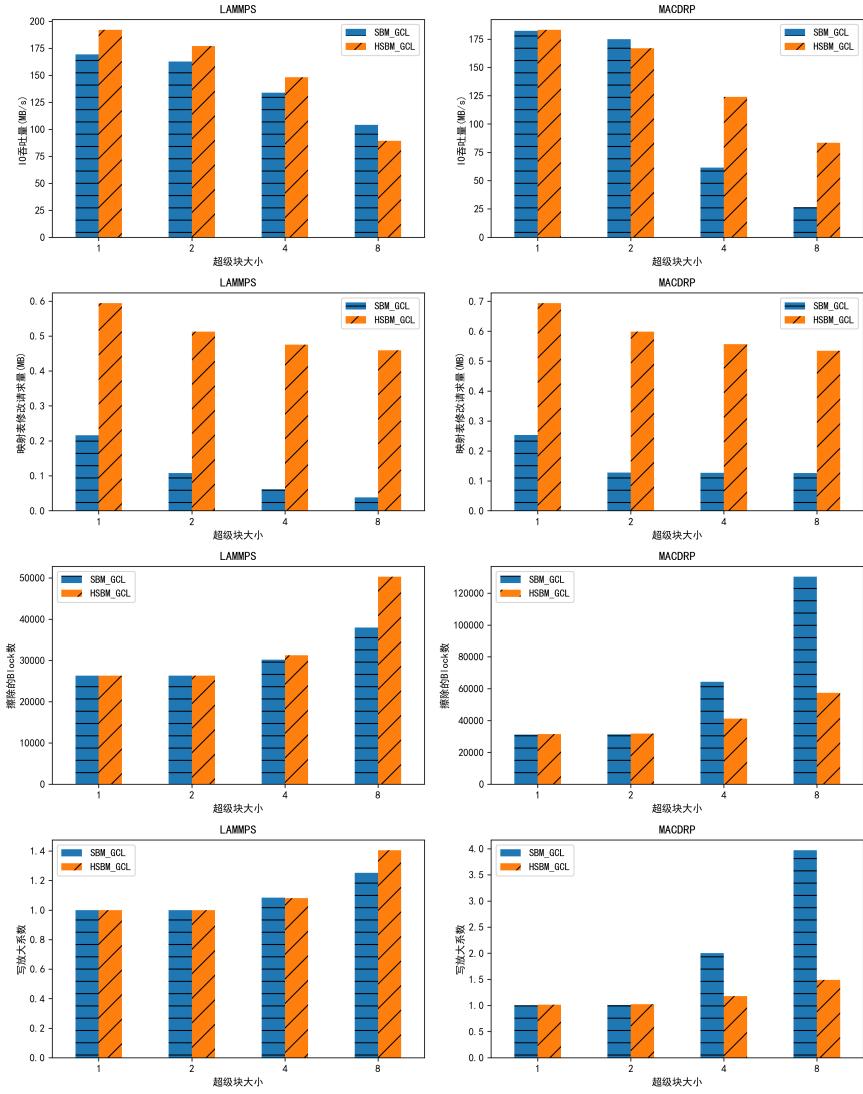


图 4.10 超级块大小对各项性能指标的影响

如图4.10，超级块大小从 1 增加到 2 时，吞吐量有很小的下降，擦除次数和写放大系数几乎不改变，而映射表维护次数下降了接近 50%。超级块大小继续从 2 增加到 4 和 8 时，对于 LAMMPS 应用，吞吐量仍是小幅下降，擦除次数和写放大系数上升很小，映射表维护次数则继续随超级块大小翻倍而减半；对于 MACDRP 应用，超级块映射-贪心垃圾回收方法的吞吐量下降幅度较大，擦除次数和写放大系数大幅上升，映射表维护次数的变化与 LAMMPS 类似。其中的原因是 LAMMPS 一次连续写入的量往往在 8 个 Block 左右，因此增大超级块大小到 8 个 Block 后，每次连续写入后需要回收的块大部分依然不含脏页从而可以直接擦除，且相邻空间的写入仍然能映射到不同的超级块上；而 MACDRP 一次

连续写入的数据量超过 2 个 Block 的比例较低，超级块大小超过 2 个 Block 后，每次连续写入的超级块很可能含有上次写入留下的脏页需要重新写入，造成较大的写放大系数；且相邻空间内的连续写入往往会映射到同一个超级块上，造成更多的映射冲突，进而引发更多的擦除和脏页重新写入。

因此，对于高性能应用，超级块大小应尽量接近其占比最大的单次写入量大小但不能超过，以同时实现最小的映射表维护成本和覆盖写成本。

## 4.9 日志块数量的选择

混合超级块映射可以调节的参数为使用的日志块数量。

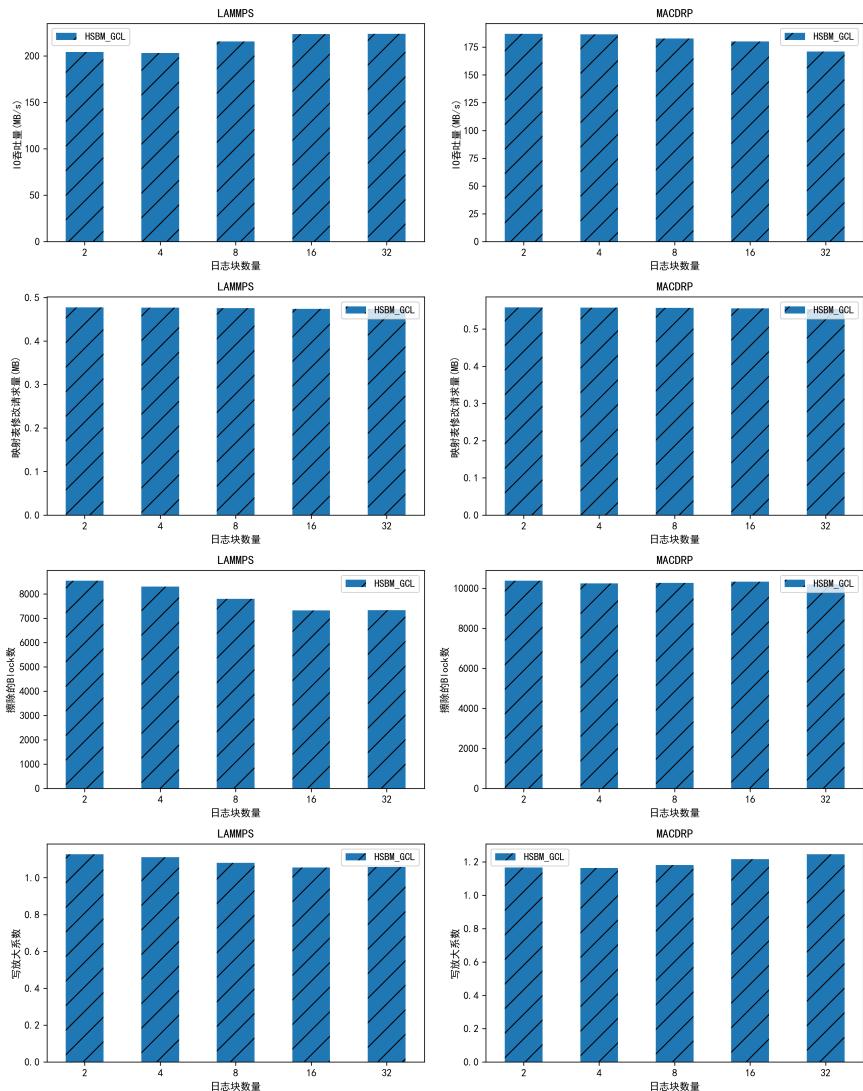


图 4.11 超级块大小对各项性能指标的影响

如图4.11，对于 2 种负载，日志块数量对混合超级块映射性能的影响很小。原因是两种负载随机访问的特征极少，顺序写入占绝大部分，这种情况下一个日志块被执行合并操作的原因不是一段时间内写入涉及到的映射关系过多导致当前写入无日志块可用，而是当前写入的日志块容量耗尽。因此日志块的数量在实验所用负载下并不是影响性能的关键。

## 4.10 本章小结

本章使用吞吐量、映射表维护成本、擦除次数和写放大系数等指标评估了页映射-贪心垃圾回收、页映射-连续空间垃圾回收，超级块映射-贪心垃圾回收和混合超级块映射-贪心垃圾回收三种方法的性能，评估结果表明连续空间垃圾回收与贪心垃圾回收相比劣势明显，不宜采用；超级块映射、混合超级块映射与页映射相比能极大降低映射表体积和维护成本，但负载中含有大量小于块大小的写入时超级块映射的性能下降严重，而混合超级块映射的性能下降相对更小，证明混合超级块映射更适用于高性能应用在开放通道 SSD 上的 IO 优化。之后本章讨论了设备容量和块内页数对不同方法性能的影响。最后本章讨论了超级块映射的重要参数——超级块大小的选取原则：应尽量接近所运行高性能应用占比最大的单次写入量大小但不能超过。

## 第 5 章 结论与展望

开放通道 SSD 因其能够将内部的结构暴露给外部程序从而提供更大的灵活性和更多的优化空间，得到了越来越广泛的应用，尤其是在数据库领域已经有了一些工作成果。但基于开放通道 SSD 优化高性能应用的工作目前还不多见。本文据此从两种高性能应用的负载特征分析入手，针对其特征实现了采用混合超级块映射和贪心垃圾回收方法的 IO 优化策略，相对于已有方法实现了更低的映射表维护开销。

本文利用高性能应用的 IO 负载中覆盖写比例较高的特点，判断其 IO 过程中会在 SSD 上产生大量不含任何脏页的无效数据块，从而将现有方法的映射粒度从页级别提升到超级块级别，同时为了处理其中块不对齐和小于块大小的写入引入日志块实现混合超级块映射，在原有 IO 吞吐性能下降不太大的同时降低了映射表的维护成本。本文同时尝试了模仿日志结构文件系统改进垃圾回收方式的做法，结果表明这一做法在垃圾回收时重新写入脏页的成本过高，写放大系数偏大，因而并不适用于本文所研究的高性能应用负载。

由于时间和精力所限，本文涉及的 IO 优化策略仅覆盖了地址映射、垃圾回收和负载均衡三方面的内容，实际系统中还可以通过写缓存、使用后台线程进行垃圾回收等方式进一步提高系统性能。这部分可以作为后续的研究内容。

## 插图索引

图 2.1	SSD 设备的内部结构 .....	4
图 2.2	LAMMPS 的 IO Trace .....	6
图 2.3	LAMMPS 的 IO 请求按页对齐情况 .....	6
图 2.4	LAMMPS 的 IO 请求按块对齐情况 .....	7
图 2.5	MACDRP 的 IO Trace .....	7
图 2.6	MACDRP 的 IO 请求按页对齐情况 .....	8
图 2.7	MACDRP 的 IO 请求按块对齐情况 .....	8
图 3.1	两种 SSD 的访问流程 .....	10
图 3.2	开放通道 SSD 的物理地址与内部结构的对应关系 .....	12
图 3.3	页映射过程示意图 .....	14
图 3.4	引入超级块概念后物理地址与开放通道 SSD 内部结构的对应关系 ..	18
图 3.5	超级块映射过程示意图 .....	19
图 3.6	日志块映射过程示意图 .....	22
图 4.1	不同优化方法的 IO 吞吐量 .....	26
图 4.2	平均 IO 吞吐量在重放过程中随 IO 请求总量的变化 .....	27
图 4.3	不同优化方法在重放过程中的单次 IO 吞吐量 .....	28
图 4.4	用于更新映射表的请求总量在重放过程中随 IO 请求总量的变化 ..	29
图 4.5	擦除的 Block 数在重放过程中随 IO 请求总量的变化 .....	30
图 4.6	写放大系数在重放过程中随 IO 请求总量的变化 .....	31
图 4.7	负载分布图 .....	32
图 4.8	设备容量对各项性能指标的影响 .....	33
图 4.9	块内页数对各项性能指标的影响 .....	34
图 4.10	超级块大小对各项性能指标的影响 .....	36
图 4.11	超级块大小对各项性能指标的影响 .....	37

## 表格索引

表 3.1	开放通道 SSD 的基本性能 .....	11
表 3.2	页映射-贪心垃圾回收策略（PM_GCL）重要结构描述 .....	13
表 3.3	页映射-贪心垃圾回收策略（PM_GCL）映射过程主要接口 .....	15
表 3.4	页映射-连续空间垃圾回收策略（PM_GCC）重要结构描述 .....	16
表 3.5	超级块映射-贪心垃圾回收策略（SBM_GCL）重要结构描述.....	18
表 3.6	引入超级块概念后物理地址到开放通道 SSD 内部结构的转换方法 ..	19
表 3.7	超级块映射-贪心垃圾回收策略（SBM_GCL）重要结构描述.....	21
表 4.1	开放通道 SSD 的基本信息 .....	25
表 4.2	不同优化方法的 IO 吞吐量 (MB/s).....	26

## 参考文献

- [1] Bjørling M, González J, Bonnet P. LightNVM: The Linux Open-Channel SSD Subsystem[Z]. 17.
- [2] Wang P, Sun G, Jiang S, et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD[C/OL]//ACM Press, 2014: 1-14[2018-05-14]. <http://dl.acm.org/citation.cfm?doid=2592798.2592804>.
- [3] González J. Towards Application Driven Storage - Optimizing RocksDB for Open-Channel SSDs[EB/OL]. [https://events.static.linuxfound.org/sites/events/files/slides/LCE2015\\_RocksDB%2BLightNVM.pdf](https://events.static.linuxfound.org/sites/events/files/slides/LCE2015_RocksDB%2BLightNVM.pdf).
- [4] Shan H, Antypas K, Shalf J. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark[C/OL]//IEEE, 2008: 1-12[2018-05-14]. <http://ieeexplore.ieee.org/document/5222721/>.
- [5] Lu Y, Shu J, Zheng W. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems[Z]. 14.
- [6] Bouganim L, Jónsson B , Bonnet P. uFLIP: Understanding Flash IO Patterns[Z]. 12.
- [7] Picoli I L, Pasco C V, Jónsson B , et al. uFLIP-OC: Understanding Flash I/O Patterns on Open-Channel Solid-State Drives[C/OL]//ACM Press, 2017: 1-7[2018-05-14]. <http://dl.acm.org/citation.cfm?doid=3124680.3124741>.
- [8] Bjørling M. Open-Channel SSDs Then. Now. And Beyond.[Z]. 2017: 25.
- [9] He J, Kannan S, Arpaci-Dusseau A C, et al. The Unwritten Contract of Solid State Drives[C/OL]//ACM Press, 2017: 127-144[2018-05-14]. <http://dl.acm.org/citation.cfm?doid=3064176.3064187>.
- [10] Luu H, Behzad B, Aydt R, et al. A Multi-Level Approach for Understanding I/O Activity in HPC Applications[Z]. 5.
- [11] Luu H, Winslett M, Gropp W, et al. A Multiplatform Study of I/O Behavior on Petascale Supercomputers[C/OL]//ACM Press, 2015: 33-44[2018-05-14]. <http://dl.acm.org/citation.cfm?doid=2749246.2749269>.
- [12] Liu W, Wu K, Liu J, et al. Performance Evaluation and Modeling of HPC I/O on Non-Volatile Memory[J/OL]. arXiv:1705.03598 [cs], 2017[2018-05-14]. <http://arxiv.org/abs/1705.03598>.
- [13] Lee S W, Moon B, Park C, et al. A case for flash memory ssd in enterprise database applications[C/OL]//ACM Press, 2008: 1075[2018-05-14]. <http://portal.acm.org/citation.cfm?doid=1376616.1376723>.

- [14] ROSENBLUM M, OUSTERHOUT J K. The design and implementation of a log-structured file system: volume 10[Z]. 1992; 27.
- [15] Park C, Cheon W, Kang J, et al. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications[J/OL]. ACM Transactions on Embedded Computing Systems, 2008, 7(4): 1-23[2018-06-13]. <http://portal.acm.org/citation.cfm?doid=1376804.1376806>.

## 致 谢

衷心感谢导师舒继武教授和陆游游教授对本人的精心指导。他们的言传身教将使我终生受益。

感谢季旭师兄帮助提供高性能应用的 IO Trace，介绍应用背景和解释 Trace 文件含义。感谢李思阳师兄在实验初期提供各种负载模拟器资源和 IO 性能评测工具。感谢杨者同学在开放通道 SSD 模拟器使用和实验用服务器维护方面提供的帮助。

感谢 L<sup>A</sup>T<sub>E</sub>X 和 THU<sup>H</sup>ESIS，帮我在排版论文时节省了不少时间。

## 声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名: \_\_\_\_\_ 日 期: \_\_\_\_\_

## 附录 A 外文资料的调研阅读报告或书面翻译

书面翻译对应的外文资料原文索引：Lu, Youyou, Jiwu Shu, and Weimin Zheng. "Extending the lifetime of flash-based storage through reducing write amplification from file systems." FAST. Vol. 13. 2013.

从文件系统方面通过减少写入放大来延长基于闪存的存储的生命周期

**摘要：**闪存由于其高性能，低能耗和低成本而作为用于企业和嵌入式系统的存储设备而广受欢迎。然而，闪存的耐用性问题仍然是一个挑战，随着存储密度随着多层单元（MLC）的采用而增加，这种问题正在变得越来越严重。之前的工作已经解决了损耗均衡和数据减少问题，但使用文件系统来提高闪存寿命的工作明显较少。传统文件系统中的一些常见机制（例如日志记录，元数据同步和页面对齐更新）会导致额外的写入操作并加剧闪存的磨损。这个问题被称为文件系统的写入放大。

为了减轻写入放大，我们提出了基于对象的闪存转换层设计（OFTL），其中的机制是与闪存协同设计的。通过利用页面元数据，OFTL 能够实现索引元数据的惰性持久性，并在保持一致性的同时消除日志记录。粗粒度块状态维护减少了持久的空闲空间管理开销。借助字节单元访问接口，OFTL 能够将小型更新与元数据进行压缩和共同定位，以进一步减少更新。实验表明，与最新页级别闪存转换层上的 ext3, ext2 和 btrfs 相比，基于 OFTL 的系统 OFSS 在 SYNC 模式下的写入放大率降低了 47.4%~89.4%，ASYNC 模式下的写入放大率降低了 19.8%~64.0%。

### A.1 引言

近年来，闪存技术有了很大的提高。随着主流设计从单层单元（SLC）转向多层/三层单元（MLC / TLC），基于闪存的存储正在见证容量的增加和每比特成本的降低，导致了在企业和嵌入式存储系统中应用的急速增长。但是，闪存密度的增加要求每个单元内有更精细的电压阶跃，这对泄漏和噪声干扰的耐受性较差。因此，闪存的可靠性和使用寿命大幅下降，产生了耐久问题 [18,12,17]。

损耗均衡和数据缩减是延长闪存存储寿命的两种常用方法。通过磨损均衡，编程/擦除（P / E）操作倾向于分布在闪存块上，以使其均匀磨损 [11,14]。数据缩减用于闪存转换层（FTL）和文件系统。FTL 引入了重复数据消除和压缩技术，以避免冗余数据的更新 [15,33,34]。文件系统试图通过尾部打包 [7] 或数据压缩来减少更新 [2,13]。但是，这些数据缩减技术在减少来自文件系统的写入放大方面效率低下，因为以下两个原因，从文件系统生成的元数据更新难以减少。一个是数据缩减不应损害文件系统机制。例如，在数据缩减中不应删除日志中的重复数据。另一个原因是大多数元数据出于一致性原因频繁同步，阻止了数据减少。

不幸的是，传统文件系统中的一些常见机制加剧了闪存写入放大。首先，通常用于保持更新原子化的日志记录加倍了写入大小，因为它将数据和元数据也复制到了日志。其次，元数据经常同步写入，以避免出现故障时的数据丢失。即使元数据占用的存储空间很小，频繁的写入操作也会产生巨大的写入流量，这对存储磨损产生巨大影响。第三，即使对于几个字节的更新，对存储设备的写入也是页对齐的。系统对来自应用程序的实际访问大小进行隐藏，因此更新强度被页面访问粒度放大。部分页面更新几乎总是需要读取 - 修改 - 写入。更糟糕的是，闪存页面大小的增加使得这个问题更加严重 [17]。最后但并非最不重要的一点是，分层设计原则使得文件系统和设备彼此不透明，原因是窄块接口 [35,29,28]。闪存存储上的文件系统机制无法解决耐久性问题，因此，使用这些机制，闪存会更快磨损。透明性还会阻止文件系统中的机制利用存储设备的特性。

事实上，闪存为更好的系统设计提供了机会。首先，每个页面都有一个页面元数据区域，也是 OOB（带外）区域。保留空间可用于为索引元数据的惰性持久性或提供写入原子性的事务信息保留反向索引。其次，块被擦除后，闪存块的全部区域都变为干净状态。可以通过以擦除块（256 KB）为单位跟踪空闲空间而不是文件系统块（4 KB）来减少元数据开销。第三，与 HDD（硬盘驱动器）相比，随机读取性能显著提高。部分页面更新可以被压缩，而不会引入显着的随机读取惩罚。如上所述，闪存特性可以用来缓解与系统协作时的耐久性问题。

我们提出了一种名为 OFTL 的基于对象的闪存转换层设计，将存储管理卸载到对象存储，并与闪存共同设计系统以减少写入放大。我们的贡献总结如下：

- 我们提出了 OFTL，一个基于对象的闪存翻译层，以促进对数据的组织方式进行语义感知。
- 为了降低索引元数据持久性和日志记录的成本，我们懒惰地刷新索引并通过

过在页面元数据中保留反向索引和事务信息来消除日志记录。

- 我们还使用粗粒度块状态维护来跟踪页面状态，以降低可用空间管理的成本。
- 使用 OFTL 中的字节单元接口，我们压缩部分页面更新并将它们与元数据共同定位以减少页面更新次数。
- 我们实施基于 OFTL 的系统并在不同类型的工作负载下对其进行评估。结果显示，与现有文件系统相比，写入放大率显着降低。

本文的其余部分安排如下。第 2 部分给出了闪存和基于闪存的存储系统体系结构的背景。然后，我们在第 3 部分中介绍我们的 OFTL 设计，并在第 4 部分中与闪存共同设计系统机制。我们在第 5 部分中描述了实现，并在第 6 部分中评估了设计。相关工作在第 7 部分中介绍，第 8 节给出了结论。

## A.2 背景

### A.2.1 闪存基础

闪存以页为单位读取或写入，并以闪存块单位擦除。页面大小范围从 512B 到 16KB [11,17]。每个 Flash 页面都有页面元数据，这些页面元数据与页面数据一起自动访问。通常，一个 4KB 页面有 128B 页面元数据 [11]。闪存块由闪存页面组成，例如，一个 256KB 闪存块包含 64 页 [11]。闪存块进一步排列在平面和通道中，以便在闪存驱动器内进行并行处理，称为内部并行。

与硬盘驱动器相比，闪存具有两个独特的特征，即不可覆盖属性和耐久性限制，这些都应该被文件系统设计考虑到。不可覆盖属性意味着被编程的页面不能被重新编程，直到它所在的闪存块被擦除。对编程页面的更新被重定向，以不可覆盖的方式清除页面，而编程页面无效以后再擦除。耐久性限制是每个存储器单元具有有限数量的 P/E 操作。存储器单元耗尽更多的 P/E 周期，导致寿命和可靠性的恶化。

### A.2.2 基于闪存的存储系统的体系结构

闪存在嵌入式系统中广泛使用。在嵌入式系统中，闪存不支持块接口，并由文件系统直接管理，在该系统中实现映射，垃圾收集和耗损均衡。文件系统专用于闪存，称为闪存文件系统 [10,32]，如图 1 (a) 所示。随着容量的增加和成本的降低，从笔记本电脑到企业服务器的计算机系统采用闪存设备作为 HDD 的替代品。FTL 在设备固件中实现以提供块接口，如图 1 (b) 所示。

由于嵌入式 FTL 需要来自嵌入式处理器和大型 DRAM 的巨大计算能力来增加设备容量，因此 FusionIO 在称为 VSL（虚拟存储层）的软件中实现了 FTL，共享主机 CPU 周期和主存储器容量 [4]。图 1 (c) 显示了架构。基于软件的 VSL 为文件系统提供了优化的机会。在 VSL 上提出了 DFS 以利用 VSL 中的存储管理 [20]，并且使用日志结构的 FTL 从 VSL 中导出原子写入接口以向系统提供写入原子性 [28]。

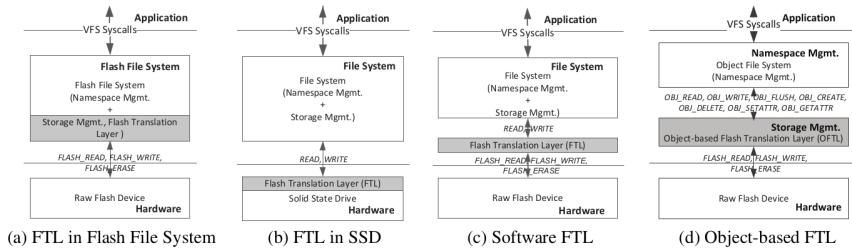


图 1 基于闪存的文件系统结构

尽管基于软件的 FTL 比嵌入式 FTL 具有更好的性能，但文件系统和 FTL 之间的窄块接口阻止文件系统或 FTL 的优化。文件语义隐藏在狭窄的界面之后，阻碍了智能存储管理 [35,26]。此外，闪存属性对文件系统不透明，导致文件系统优化失去了机会 [20,29,28]。与基于对象的 SCM [21] 类似，我们提出了基于对象的 FTL 设计 OFTL，以便与文件系统和闪存更好地配合，如图 1 (d) 所示。在基于 OFTL 的体系结构中，存储管理从文件系统移动到 OFTL 以直接管理闪存，因此可以调查闪存属性以优化文件系统机制设计，如元数据同步的日志删除和频率降低。OFTL 通过读/写/擦除操作管理闪存，并直接访问每个闪存页面的页面元数据。另外，OFTL 将字节单位访问接口导出到文件系统，这是一个基于对象的文件系统，不受存储管理的限制，只能管理名称空间。

### A.3 基于对象的闪存转换层

基于 OFTL 的体系结构将存储空间管理从文件系统卸载到 OFTL，以更好地协同设计文件系统和 FTL。OFTL 访问页面单元接口中的原始闪存设备，同时将字节单元访问接口导出到文件系统。因此，OFTL 将从每个对象的逻辑偏移量映射到 Flash 页面地址。在本节中，我们将描述 OFTL 接口和数据组织。

表 1 对象接口

操作	描述
<b>oread</b> (devid, oid, offset, len, buf)	从编号为 oid 的对象的 offset 位置读取数据到 buf
<b>owrite</b> (devid, oid, offset, len, buf)	向编号为 oid 的对象的 offset 位置写入 buf 中的数据
<b>oflush</b> (devid, oid)	将编号为 oid 的对象的数据和元数据持久化
<b>ocreate</b> (devid, oid)	创建编号为 oid 的对象
<b>odelete</b> (devid, oid)	删除编号为 oid 的对象
<b>ogetattr</b> (devid, oid, buf)	获取编号为 oid 的对象的标签
<b>osetattr</b> (devid, oid, buf)	设置编号为 oid 的对象的标签

**OFTL 接口。** OFTL 将字节单位读/写接口导出到文件系统，以直接将访问大小以字节为单位传递给 OFTL。表 1 显示了对象接口。**oread** 和 **owrite** 接口都将字节单位的偏移和 **len** 传递给 OFTL，而不是扇区单元。因此，OFTL 从应用程序中获得准确的访问大小，从而可以将小更新压缩成更少的页面，这在 4.3 节中讨论。另外，使用对象接口中提供的 **oid** 对每个对象进行操作，这使得 OFTL 知道所访问页面的数据类型。OFTL 利用对象语义对更新关联数据进行聚类。此外，OFTL 使用类型语义区分数据页面的索引页面，以便在页面元数据中保留辅助元数据以实现懒惰索引，这在 4.1 节中讨论。OFTL 以页面单元读取/写入操作和块单位擦除操作访问闪存。页面元数据读/写可以在 NVMe 规范 [6] 之后直接从 OFTL 访问，该规范定义了用于访问 PCI Express 总线上的非易失性存储器的主机控制器接口。

**数据组织。** OFTL 由两个主要部分组成，对象存储和块信息元数据。单个根页面标识每个对象存储和块信息元数据的位置，如图 2 所示。对象存储分为三个级别：对象索引，对象元数据页面和对象数据页面。对象索引使用 B + 树将对象 ID 映射到其对象元数据。对象元数据包含传统上存储在文件系统索引节点中的信息，包括引用对象数据页面中的地址的分配信息。块信息元数据保持每个闪存块的元数据信息，包括闪存块状态，无效页数（其数据过时但未被擦除）以及每个闪存块的擦除次数。每个闪存块有三种状态：FREE，UPDATING 和 USED，每个页面有三种状态：FREE，VALID 和 INVALID，这些在第 4.2 节中有解释。块信息元数据是以日志结构化的方式编写的。每个块信息条目有 32

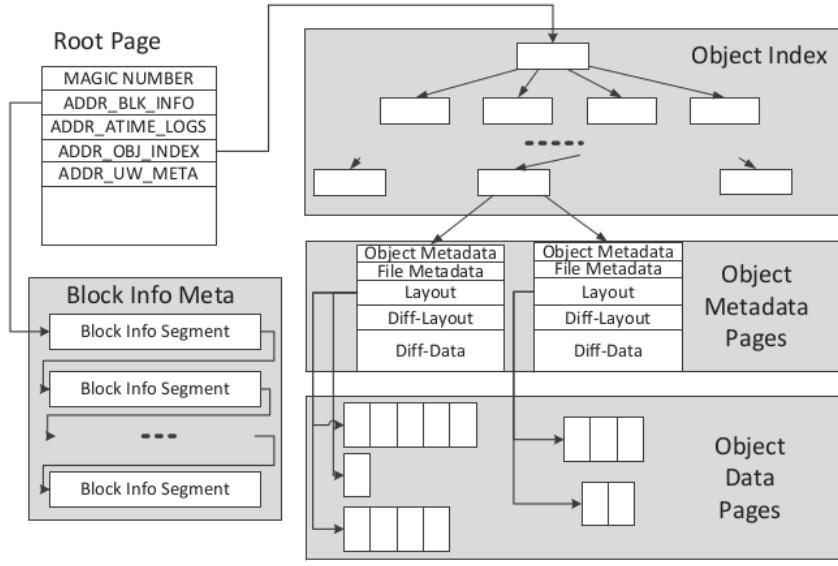


图 2 OFTL 结构

位，其中 20 位用于擦除计数，10 位用于无效页数，2 位用于闪存块状态。块信息不是与每个闪存块一起存储，而是存储在闪存中的一个单独的空间中，这会在垃圾收集期间导致更少的页面更新。

#### A.4 系统与闪存的协同设计

OTFL 使用三种技术来利用底层闪存设备的特性。在 4.1 节中，我们介绍了 Backpointer-Assisted Lazy Indexing，这是一种有效维护数据和元数据之间一致性的机制。在 4.2 节中，我们介绍了我们的粗粒度块状态管理方法，它可以降低状态写入的频率。在第 4.3 节中，我们介绍了我们的压缩更新技术，该技术摊分了跨多个未对齐页面写入的页面写入成本。

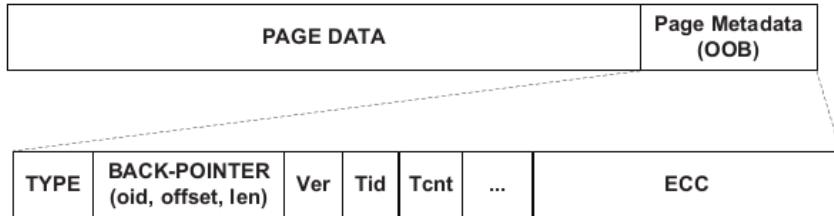


图 3 页元数据

#### A.4.1 反向指针辅助的惰性索引

索引元数据（对象元数据页面中用于指向数据页面的数据布局的指针或者指向对象索引以指向对象元数据页面的指针）应该同步写入以防存储设备损失或不一致。索引元数据的同步也称为索引持久性。虽然典型指针的大小为 8 个字节，但索引持久性会更新整个页面，即 4KB 或更大。因此，频繁的索引持久性导致严重的写入放大。

在索引元数据的惰性持久性中，在每个索引页面的页面元数据中采用了类型特定的反向指针技术，以减少索引持久化的频率，同时保持一致性。如图 2 所示，对象索引，对象元数据页面和对象数据页面在树结构中进行索引并形成三级分层结构。图 3 说明了页面元数据组织。在 OFTL 中，我们有两种类型的后向指示器 (`oid`, `offset`, `len`)。一个用于数据页面来对对象元数据进行逆向索引，`oid` 和 `offset`, `len` 分别表示对象 id, 对象中的逻辑页面偏移量和页面的有效数据长度。另一个用于对象元数据页面来反向索引对象索引，并且只有 `oid` 被设置为表示对象 ID。在写入反指针时，首先设置类型以指示反指针的类型，以便可以针对不同类型正确理解反指针。我们也保留版本以确定最新的恢复页面版本。因此，类型特定的反向指针用作反向索引，并且索引持久化与页面更新解耦。

为了减少反向索引的扫描时间以在系统故障后重建索引元数据，我们使用更新窗口来跟踪最近分配的未完成索引元数据持久化的闪存块。更新窗口由检查点进程维护，检查点进程在更新窗口中的空闲页数低于阈值并且需要扩展窗口时触发。更新窗口描述了在发生故障后需要检查其反向索引的块集，因为索引可能不会引用它们。

更新窗口还提供多个页面的更新原子性。页面元数据保存事务信息 (`tid`, `tcnt`)，其中 `tid` 是每个写入操作的更新窗口中的唯一 ID，`tcnt` 是更新中页面的数量。对于写操作的所有页面，只有一个页面具有 `tcnt` 设置，其他页面的 `tcnt` 值为零。`tcnt` 用于在系统故障后检查操作的完整性。垃圾收集不允许在更新窗口中的闪存块上执行，以便完成原子性检查的事务信息。因此，使用闪存提供的写入原子可以消除日志。

在系统故障后重新启动时，扫描更新窗口中的对象数据页面。首先检查交易信息以确定涉及该页面的写入操作的完整性。如果写入不完整，则写入的所有页面都将被丢弃。这样就保证了原子性。在检查之后，如果在系统失败之前对象布局未被写入稳定，则读出后向指针以更新对象布局。由于所有更新都位于自上次检查点以来的更新窗口中，因此可以通过从当前更新窗口重新构建扫描

页面中的后向指示器，将对象布局更新为最新版本。文件大小元数据也会随着所有有效数据大小的重新计算而更新。虽然其他描述性元数据（如修改时间和访问控制列表）在意外崩溃后可能会丢失，但系统一致性不会受到影响。类似地，可以用对象元数据页面的当前更新窗口的扫描来更新对象索引。通过页面元数据和更新窗口中的辅助信息，可以减少索引持久化频率，并提供写入原子性来消除日志。

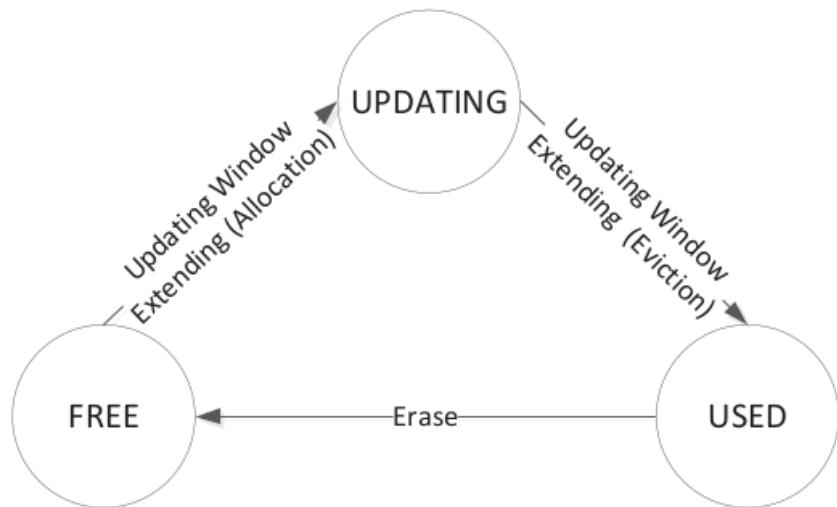


图 4 闪存块的状态转换

**一致性和耐久性讨论。**在懒惰索引技术中，系统崩溃后，除索引指针和大小之外的元数据可能会丢失。这导致元数据的过时版本，这伤害元数据更新的持久性。与耐久性相比，我们将一致性视为更重要的事情，因为索引的腐败会导致文件系统的不一致甚至失败。我们使用懒惰索引技术来保证索引和大小可以在恢复期间重建。但是对于持久性问题，我们选择让文件系统或应用程序确定让其他元数据持久化的时间，并显式同步。

#### A.4.2 粗粒度块状态维护

在闪存中，每个页面都有三种状态：FREE、VALID 和 INVALID。空闲页面是尚未写入的页面。有效页面是已写入的页面，其数据是有效数据。无效页面是已写入但其数据已过时的页面。有效和无效页面也称为 INUSE 页面。每个闪存块也有三种状态：FREE、UPDATING 和 USED。空闲块中的页面全部为空闲页面，已使用块中的页面全部处于使用页面中，而只有更新块同时具有空闲页面或未使用页面。由于页面是按顺序写入闪存块中的，因此最新分配的页码用于

将空闲页面与更新块的使用中分开。因此，OFTL 通过跟踪闪存块状态来区分空闲页面和使用中页面。对于 `inuse` 页面，索引元数据用于进一步区分有效页面和无效页面。有效页面在索引元数据中编入索引，而无效则不在索引中。因此，可以通过跟踪闪存块的状态来识别空闲，有效的无效页面，并且通过闪存块单元中的状态维护而不是页面单元来减少空闲空间管理成本。

OFTL 通过降低元数据持久性的频率来进一步降低成本。闪存块状态的持久性仅在闪存块被分配给或从更新窗口驱逐出时执行，如图 4 的顶部所示。状态持久性被放宽，但是满足以下两个条件：（1）持久性空闲块是实际空闲块集的子集；（2）持续无效页面的数量不超过无效页面的实际数量。

第一个条件意味着一个空闲块可以被认为是非空闲的，这可能导致分配丢失。但是一个非空闲块不能被视为空闲，否则写入失败。它要求从 `FREE` 到 `UPDATING` 的状态转换立即刷新，但将状态持久性从 `USED` 释放到 `FREE`。第二个条件放松了无效页面的数量持久性。无效页面的数量用于选择驱逐的闪存块并检查垃圾回收期间移动的有效页面的数量。通过检查索引元数据来区分有效页面和无效页面。在达到有效页面的数量之前，剩余的所有页面都是无效的，并且不再需要无效的页面检查，从而使被驱逐块的页面移动停止。在最坏的情况下，如果无效页面的持续数量少于实际数量，则必须检查所有页面。由于系统崩溃很少出现，所以对垃圾收集效率的影响是有限的。类似于无效页面的持续数量，记录的擦除次数可能会保持不变，这也可能接受，因为擦除次数不敏感。

总之，空闲空间管理从粗粒度块状态维护中受益，因为闪存块粒度状态跟踪和状态持久性的降低都减少了元数据成本。

#### A.4.3 压缩更新

通过字节单元访问接口，OFTL 能够识别部分页面更新，这些页面更新只更新一页的一部分，既适用于小于一页的小型更新，也适用于大型更新的正面/反面。压缩更新技术压缩同一对象的部分页面更新，并将它们与其对象元数据页面共同定位以减少更新页面。

**部分页面更新。** OFTL 中的部分页面更新被压缩并插入差异页面，又名差异页面。存储在 `diff-page` 中的每个数据段都称为 `diff-data`。`Diff-data` 使用三元组 `<ooff, len, addr>` 进行 `diff-extent` 索引，其中 `ooff` 和 `len` 分别表示对象偏移量和 `diffdata` 的长度，`addr` 是 `diff`-数据在 `diff`-页。`Diff-extents` 在 `diff-layout` 中按照对象偏移量的升序保持不变。每个对象都有一个 `diff-layout`，它是一个对象的所有

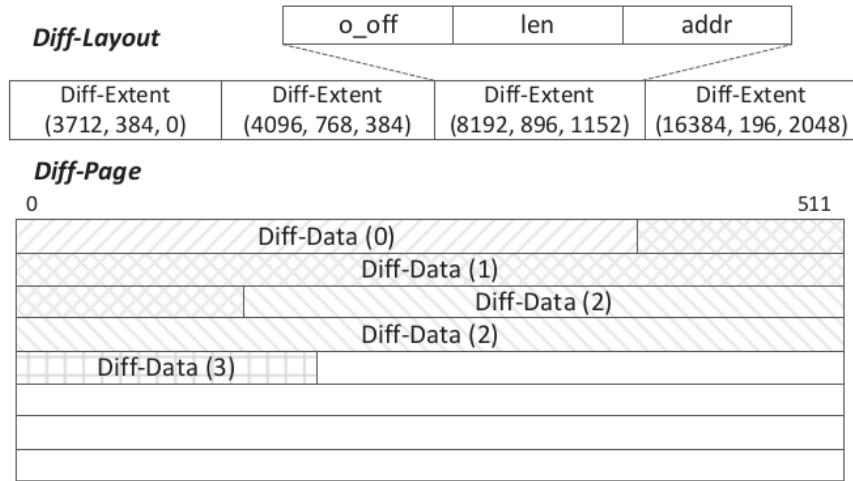


图 5 Differential Layout (Diff-Layout) 数据结构

diff-extents 的集合。图 5 显示了数据结构。

尽管部分页面更新被吸收到差异页面中，而差异页面在差异布局中编入索引，但整个页面更新会直接写入对象页面。对象页面在布局中进行索引，这是一个对象的所有范围的集合，用于保存完整页面的起始地址和长度对。与差异布局一样，布局也记录在对象元数据页面中，如图 2 所示。在压缩更新技术中，读取，写入和合并操作如下所述：

- **写入：**写入数据分为多页。不开始或结束页面对齐的部分被标识为部分页面更新，而其他部分则被认定为整页更新。然后部分页面更新在 diff-pages 中更新。因为部分更新总是取代整个页面，所以整个页面更新必须使相应的差异数据无效并删除其差异范围，然后更新布局以引用整个页面。
- **合并：**当差异页面已满时需要合并操作。合并时，会扫描 diff-extents 以选择在 diff-pages 中消耗最多空间的驱逐的逻辑页面。然后读取驱逐的逻辑页面的对象页面并与 diff-data 合并。之后，合并的页面被写入新的对象页面，并且相应的 diff-data 和 diff-extents 被删除。
- **读取：**首先以不同的格式检查读取操作。如果它在 diff-data 中被满足，则读缓冲区被 diff-data 填充。否则，读取对象页面并与任何现有的对应 diff-data 合并。

**更新协同定位。**在大多数情况下，每个对象的元数据大小远小于页面大小。OFTL 不是压缩多个对象的元数据，而是与对象元数据页面不同的页面。由于每个数据更新之后都有元数据更新，例如文件大小或修改时间，因此 diff-data 和

元数据页面的共同位置通常会节省一页写入。在协同定位中，diff-page 的大小小于一个页面大小，并且取决于对象元数据的大小更改。diff-page 的大小是通过从 flash 页面大小中减去元数据大小来计算的，flash 页面大小用于检查 diff-page 是否已满。一旦完成，触发合并操作以选择一些 diffdata 并将它们合并到对象数据页面。通过这种方式，部分页面更新的成本通过压缩和协同定位来分摊。

## A.5 实现

我们在 Linux 内核 3.2.9 中实现 OFTL 作为内核模块。该模块由三层组成：转换层，缓存层和存储层。

转换层遵循图 2 所示的设计。使用 Log-Structured Merge Tree (LSM-Tree) [27] 实现闪存对象索引，该索引具有用于快速查找的内存 B + 树并添加记录每个对象索引更新的 `<operation, object ID, phy addr>`。每个对象都有一个内存中对象的元数据数据结构，它记录访问统计信息和两个基于范围的布局，以及图 2 所示的闪存对象元数据页面。diff-layout 和 layout 都链接它们的范围在内存中的列表结构中。在写入请求上，写入数据首先分页对齐。整页更新被写入数据页面，然后进行布局更新，而部分页面更新被插入 diff-data，然后是 diff-layout 更新。对象操作转换为元数据或数据页面上的读/写操作并转发到缓存层。

元数据和数据页面缓存在缓存层中。OFTL 缓存遵循 linux 内核中页面缓冲区的设计，不同之处在于替换是用整个对象完成的。在写操作时，如果设置了 SYNC 标志，则高速缓存检查 SYNC 标志并从存储层调用闪存写入接口。否则，更新缓存层中的对象缓存。页面分配被延迟直到页面需要刷新。然后，缓存层从更新窗口分配空闲页面并调用闪存写入接口以通过存储层写入页面。

存储层从缓存层接收闪存读/写/擦除操作，构建 bio 请求，并将它们发送到通用块层。我们使用系统中的 TRIM / DISCARD 命令和 SATA 协议来执行擦除操作。OOB 的 DMA 传输遵循 NVMe 标准中的设计 [6]。由于硬件限制，目前在存储器中模拟 OOB 操作。

在 OFTL 中，我们使用简单的垃圾收集和磨损平衡策略。当空闲块的百分比下降到 15% 以下时开始垃圾收集，并一直持续到百分比超过该值。在耗损均衡中，擦除次数的上限用于防止闪存块擦除次数过多。在平均擦除次数增加后，这一上限会周期性地上升。我们将在未来纳入更好的战略。

为了评估其他文件系统解决方案，我们还实现了一个简单的目标文件系统

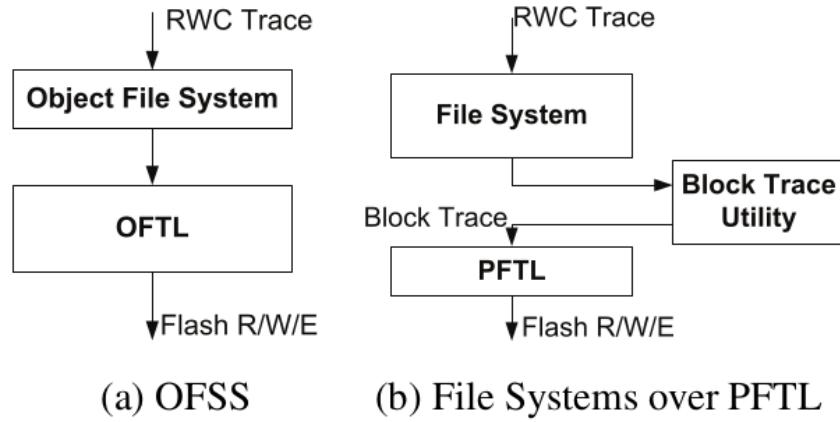


图 6 Trace 驱动的模拟框架

来解析命名空间，如图 1 (d) 所示。目标文件系统使用内存中的哈希表来保持从路径到对象 ID 的映射，并通过将对象 ID 替换为路径将 IO 请求传递给 OFTL。我们在评估部分调用基于 OFTL 的系统对象闪存存储系统 (OFSS)。

## A.6 评估

我们测量写入放大率，写入闪存的总大小/计数，除以应用程序发出的写入总大小/计数，以评估 OFSS 的写入效率以及 ext3, ext2 和 btrfs 最新的页面级 FTL。

在本节中，我们首先评估这四个系统的总体性能，然后分析元数据放大。我们还测量闪存页面大小的影响以及扩展 OFTL 设计带来的更新窗口的开销。

表 2 工作负载特性

负载名称	写次数	写大 小 (KB)	刷新次数	不对齐写的 百分比
iPhoto	496542	6651962	37054	51.2%
iPages	75661	183728	565	99.6%
LASR-1	32249	42600	1714	93.3%
LASR-2	111531	216114	14998	99.0%
LASR-3	21956	24426	3056	96.1%
TPC-C	26144	219689	489	7.1%

### A.6.1 实验设置

我们从系统级 IO 跟踪中提取读取，写入和关闭操作（RWC 跟踪）并在文件系统和 OFSS 上重放它们。图 6 显示了 tracedriven 仿真框架。在重放程序中，关闭操作会在文件系统中执行 fsync 操作或在 OFSS 中执行对象刷新操作。在 OFSS 中，我们将 I/O 的数量和大小收集到 OFTL 存储层的闪存中。在文件系统评估中，我们使用 blktrace 实用程序 [1] 在文件系统评估中捕获存储设备上的 I/O，然后在 PFTL 上重播块跟踪，PFTL 是在 Linux kernel-3.2 中作为内核模块实现的模拟页级 FTL。<sup>9</sup> 我们收集 PFTL 中闪存的 I/O 数量和大小。在仿真中，PFTL 使用类似于主存储器管理的两级页表将逻辑页号转换为物理页号。LazyFLT [23] 功能集成到 PFTL 中以减少映射开销。

我们评估桌面环境中 iBench [19] 的两个工作负载，服务器环境中 LASR [5] 的三个一个月跟踪以及数据库管理系统中的一个 TPCC 跟踪。TPC-C 跟踪是使用 strace 实用程序 [8] 从运行在 PostgreSQL 上的 DBT2 工作负载 [3] 收集的。表 2 列出了六项工作量的特点。

实验在 SUSE Linux 11.1 上进行，Linux 内核 3.2.9 在 4 核 Intel Xeon X5472 3GHz 处理器和 8GB 内存的计算机上运行。除用于操作系统的磁盘驱动器外，还使用 Seagate 7200rpm ST31000524AS 磁盘驱动器进行跟踪收集。在实验中，ext3 和 ext2 使用 noatime, nodiratime 选项进行挂载，而 btrfs 使用 ssd, discard 和 lzo 选项进行挂载。在 OFSS 设置中，默认页面大小为 4KB，闪存块大小为 256KB。对象数据和元数据的更新窗口都设置为 64 个闪存块的大小。默认的 OFTL 缓存大小为 32MB。

### A.6.2 总体比较

在本节中，针对建立在 PFTL 上的 ext2, ext3 和 btrfs，针对 OFSS 评估写入效率和 IO 时间。为了提供数据一致性，ext3 使用数据日志选项装载，而 btrfs 使用写入时复制来更新数据，而 ext2 则不提供一致性。OFSS 通过利用闪存的不覆盖属性并将事务信息保存在页面元数据中来提供数据一致性。这四个系统以 SYNC 和 ASYNC 模式进行评估。在 SYNC 模式下，数据和元数据在请求返回之前需要刷新稳定。Ext2 和 ext3 使用同步选项安装以支持 SYNC 模式。Btrfs 在文件打开操作中使用 O\_SYNC 标志，因为 btrfs 中不支持 sync 安装选项。在 ASYNC 模式下，数据和元数据被缓存在内存中，直到显式同步，时间到期或缓存逐出。三个文件系统的默认挂载选项支持 ASYNC 模式。

表 3 写放大的整体评估

Workload	System	write-cnt-amplification		write-size-amplification		IO-time (unit: s)	
		sync	async	sync	async	sync	async
iPhoto	Ext3	2.7519	0.2024	3.5354	1.8725	487.744	257.356
	Ext2	5.2292	0.1206	2.4030	0.9163	325.571	126.185
	Btrfs	8.9320	0.2602	5.6071	1.0595	770.942	144.671
	OFSS	1.2304	0.1428	<b>1.1786</b>	<b>0.8916</b>	64.146	33.264
iPages	Ext3	2.3711	0.0267	9.8083	2.0491	37.174	7.764
	Ext2	2.7763	0.0182	5.3058	1.0137	20.538	3.833
	Btrfs	6.1918	0.0313	23.1998	1.0914	87.451	4.107
	OFSS	1.7143	0.0097	<b>3.5739</b>	<b>0.9758</b>	6.837	0.732
LASR-1	Ext3	3.7777	0.1127	19.2951	2.5593	16.959	2.245
	Ext2	1.8656	0.0617	6.3874	1.1445	5.633	1.044
	Btrfs	6.1207	0.1992	33.3570	2.0008	29.246	1.744
	OFSS	1.2081	0.0636	<b>4.0123</b>	<b>0.9779</b>	2.884	0.928
LASR-2	Ext3	3.7964	0.2147	13.5078	3.2976	60.167	14.784
	Ext2	1.7143	0.1892	4.2734	1.5034	19.202	6.761
	Btrfs	7.2232	0.5822	32.9221	3.0874	145.612	13.739
	OFSS	2.1302	0.1719	<b>4.4045</b>	<b>1.1021</b>	13.332	4.065
LASR-3	Ext3	11.5083	0.1602	53.7069	4.7793	27.083	2.413
	Ext2	2.4258	0.1184	9.4100	2.0676	4.768	1.044
	Btrfs	6.0922	0.2823	44.7353	3.4276	2.263	1.715
	OFSS	1.3935	0.1345	<b>5.2492</b>	<b>1.2647</b>	2.095	0.835
TPC-C	Ext3	3.3749	0.0666	4.9863	1.8068	22.532	8.196
	Ext2	3.8604	0.0321	2.3813	0.7050	10.800	3.201
	Btrfs	6.5125	0.0336	10.1339	0.8980	45.711	4.044
	OFSS	1.0696	0.0352	<b>1.0461</b>	<b>0.6822</b>	2.588	1.380

表 3 显示了在 SYNC 模式下每个系统的总写入次数和写入大小所测得的写入放大倍数。写入计数在 ext3, ext2, btrfs 和 OFSS 中的平均写入放大倍数分别为 4.60, 2.98, 6.85 和 1.11。写入大小的写入放大显示类似的结果，并且 ext3, ext2, btrfs 和 OFSS 中的平均值分别为 17.47, 5.03, 24.99 和 2.64。差异主要来自与写入操作相关的元数据成本。在 ext2 中，写操作具有数据更新，inode 更新的子操作，并且如果需要空间分配，则有时会更新位图。在使用数据日志的 ext3 中，数据和元数据复制到日志日志中，然后进行提交/异常同步写入，稍后通过日志守护进程对实际位置进行检查点设置。即使写入计数在 ext3 和 ext2 中由于合并写入而在检查点中接近，由于在日志日志中添加了重复的数据和元数据，ext3 中的写入大小仍然远远大于 ext2 中的写入大小。在 SYNC 模式下，btrfs 将更新记录到特殊的日志树中以消除整个系统更新 [30]，从而导致写入放大率高。Btrfs 针对性能进行了优化，它使用“ssd”分配方案进行免寻找分配，并且尚未针对闪存耐久性进行优化 [2]。随着杂志的淘汰，元数据同步频率的降低和压缩更新，OFSS 相对于其他三种传统文件系统的写入放大率降低了 47.4%~89.4%。

如表 3 所示，ASYNC 模式下的写入放大率分别为 0.13, 0.09, 0.23 和 0.09，写入次数分别为 2.73, 1.23, 1.93 和 0.98，写入大小分别为 ext3, ext2, btrfs 和 OFSS。在 ASYNC 模式下，写入放大并不像 SYNC 模式那样糟糕。原因是元数据同步

不频繁，元数据更新在缓冲区以及数据中合并。但是，由于日志记录，ext3 中的写入强度加倍。Btrfs 必须更新用于写入时复制更新机制的索引元数据，消耗更多页面。此外，当页面未对齐时，页面对齐的更新机制会浪费空间，这会对所有三个传统文件系统产生写入放大效果。相比之下，与三个传统文件系统相比，OFSS 性能更好，写入放大率降低了 19.8%~64.0%。

总体 IO 时间也显示在表 3 中以进行性能比较。由于硬件限制，我们通过向 SSD 发出请求而不是原始闪存来评估性能。由于我们设计的重点是写入放大减少而非映射策略的性能导向设计，垃圾收集或耗损均衡，因此 SSD 内部的 FTL 对性能评估影响不大。我们收集和累积每次操作的设备 I/O 时间，这是从操作到 SSD 的问题到 SSD 的确认的度量。如图所示，两种模式下的 OFSS 性能都明显优于其他模式。写入大小的减少不仅延长了闪存存储的使用寿命，而且还提高了性能。此外，性能改进部分来自基于对象的 FTL 设计，该设计使用 OFTL 缓存中的延迟空间分配来实现更好的 I/O 调度。

**工作量讨论。**一般而言，OFSS 为拥有大量页面未对齐更新或频繁数据同步的工作负载带来更大的好处。当同步写入操作时，传统文件系统需要多个页面，包括数据，inode 和位图文件系统块的页面才能更新。元数据开销对小更新很重要，其中只更新了几个字节。平均更新大小小于 1KB 的 LASR-1 和 LASR3 工作负载在 SYNC 模式下暴露了无法接受的写入放大，而 OFSS 极大地减少了元数据开销，并且使得放大率更低。由于压缩更新技术，未对齐的页面更新也可以从 OFSS 中受益。一个反例是 TPC-C 工作负载，它在用户空间中拥有自己的缓冲区，并以页为单位写入数据。OFSS 对 TPC-C 工作负载的改进不如其他工作负载那么重要。

### A.6.3 元数据的写放大

为了进一步了解 OFSS 的改进以及期刊移除和元数据同步缩减的效果，我们仔细研究元数据放大。在评估中，我们通过使用 `dump2fs` 命令的转储信息检查 ext2 和 ext3 的物理布局来识别每个访问的文件系统块类型。日志类型由块跟踪中的“k 日志”标识。我们从此分析中忽略了 btrfs，因为我们无法区分块跟踪中写入地址的元数据和数据更新。图 7 显示了表格中的总元数据放大率以及 SYNC 和 ASYNC 模式中表格上方的各种元数据的百分比。

从图 7 中，ext3 的元数据放大有两个观察。一个是与文件系统元数据的写入放大相比，PFTL 的写入放大可以忽略不计。另一个原因是日志显着地放大了写

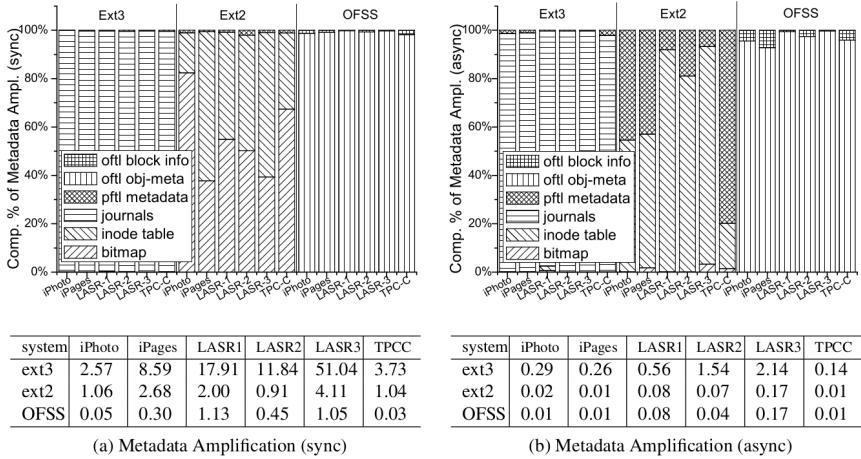


图 7 元数据的写放大

入并且控制了成本。ext2 和 ext3 之间的区别验证了日志删除的好处。在下文中，我们将通过将 OFSS 与 ext2 进行比较来进一步了解减少元数据同步的好处。

在元数据同步减少评估中，我们在计算 OFTL 的元数据放大时，从 OFSS 中的元数据页面中去除差异数据。如图 7 (a) 所示，inode 和 bitmap 更新消耗了 SYNC 模式中大部分的放大。inode 更新的写入放大率从平均值 ext2 中的 0.99 降至 OFTL 元数据更新所测量的 OFSS 中的 0.50。好处不仅来自懒惰索引，而且来自使用压缩更新的已摊销元数据页更新成本。文件系统块位图更新的写入放大率从平均 ext2 中的 0.93 降低到用 OFTL 闪存块信息更新测量的 OFSS 中的 0.0019，其显示了粗粒度块状态维护的益处。由于空闲空间是以闪存块单位进行管理的，实验中为 64 页 (256KB)，比 ext2 中的 4KB 块大得多，因此粗粒度块状态维护极大地有利于自由空间管理。

在如图 7 (b) 所示的 ASYNC 模式下，inode 和元数据更新成本分别是 ext2 和 OFSS 的主要成本。他们两人的写入放大成本平均在 0.05 左右。传统文件系统将多个 inode 压缩到一个 inode 表文件系统块中，与 OFSS 中的压缩更新技术相比，这是一种减少元数据放大的不同方法。这两种方法在 ASYNC 模式下都有类似的效果。ext2 中的文件系统块位图在 ASYNC 模式下进行缓存和合并，使成本接近 OFSS 中的闪存块状态维护成本。

#### A.6.4 闪存页面大小的影响

闪存页面的大小已经暴露出闪存制造的增长趋势 [17]。我们用 4KB, 8KB, 16KB 和 32KB 不同页面大小评估 ext2, ext3, btrfs 和 OFSS 的写入效率。由于 ext2 和 ext3 支持 4KB 的最大文件系统块大小，并且当 lead / node / sector 大小设

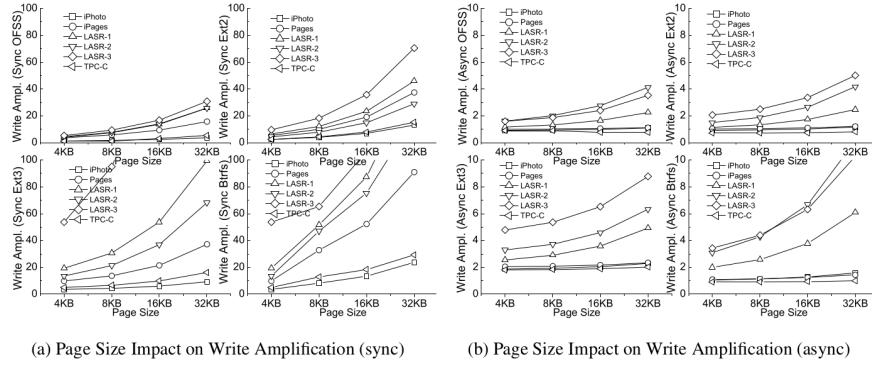


图 8 闪存页大小对写放大的影响

置为 8KB 或更高时, btrfs 不稳定, 所以我们将 4KB 块跟踪访问与 PFTL 中不同的 Flash 页面大小对齐以供评估。如图 8 所示, 随着传统文件系统中页面大小的增加, 写入放大率显着增加, 而 OFSS 显示出显着的改进。LASR-1 和 LASR-3 等小访问大小的工作负载的恶化速度要快得多。另一个观察结果是, 当 Flash 页面大小增加时, 如图 8 (a) 所示的 SYNC 模式下的写入放大比图 8 (b) 所示的 ASYNC 差很多。这是因为在大多数情况下, SYNC 模式下的访问大小远小于页面大小。相比之下, ASYNC 模式下的访问被缓存并合并为更大的请求。OFSS 中的压缩更新技术进一步压缩了部分页面更新并将它们与元数据共同定位, 以减少要更新的页面, 从而提高页面利用率。因此, OFSS 在 SYNC 模式下的写入放大效果更好, 大约为 30, 在 ext3 中接近 9%, ASYNC 模式下的写入放大接近 3, 在 btrfs 中接近 25%。

#### A.6.5 扩展更新窗口的开销

更新窗口用于减少系统崩溃后的扫描时间, 但由于扩展更新窗口时存在索引持久性而导致额外写入。为了评估开销, 我们收集了每个操作的 I/O 延迟的两个数据集, 一个用于正常扩展, 另一个用于没有索引持久性的扩展。两个数据集中潜伏期的累积分布用图 9 中几乎相同的线描绘, 显示了接近的性能。我们还会在扩展期间识别 I/O 并收集每个 I/O 的延迟时间, 以显示对外部 I/O 延迟的影响。扩展期间 I/O 的延迟与所有 I/O 的平均值相差不大。在该图的放大部分中, 即使延迟 I/O 的累积分布在延迟小于 200us 时比平均增长慢, 当延迟大约为 200us 时, 其增长得更快。结果, 扩展的 I/O 中有 99.6% 的延迟小于 400us。此外, 扩展操作很少出现, 只有 0.04% 的 I/O 满足扩展。因此, 我们得出结论: 扩展更新窗口对性能的影响有限。

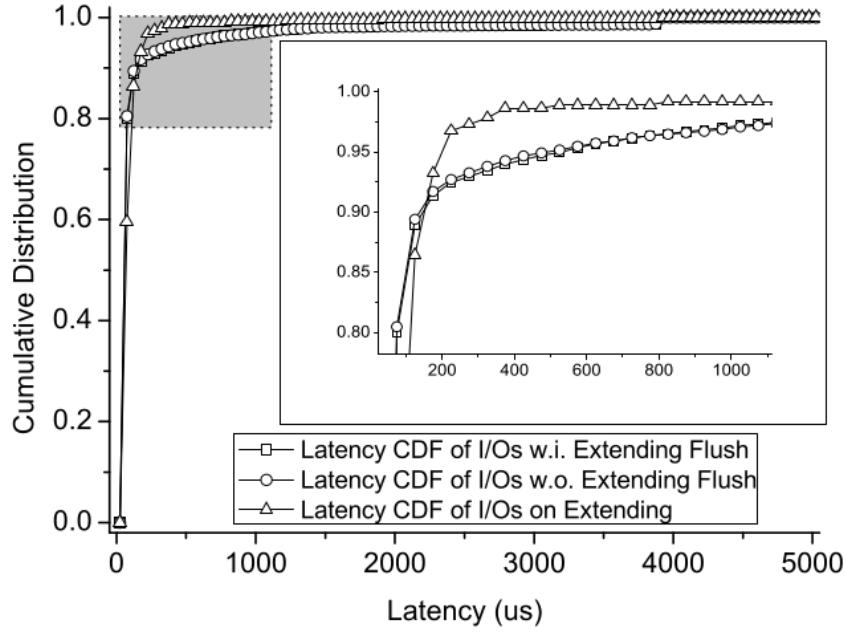


图 9 扩展更新窗口的开销

## A.7 相关工作

Flash 文件系统 [32,10,9] 利用闪存的 nooverwrite 属性进行日志结构更新，通过直接管理闪存来优化性能。但除了磨损平衡之外，他们对闪光耐力没有太大的贡献。无名写入 [35] 和直接文件系统 [20] 建议删除文件系统的映射，以允许 FTL 管理存储空间以提高性能。但是文件语义无法传递给 FTL 进行智能存储空间管理。OFTL 采用基于对象的存储的概念 [25] 将对象接口导出到文件系统，从而轻松地将文件语义传递到设备以进行智能数据布局。

最近的研究还提出消除闪存中日志造成的重复写入。Write Atomic [28] 通过利用 FusionIO ioDrives 中 VSL [4] 的基于日志的结构导出原子写入接口。TxFlash [29] 在页面元数据的帮助下使用循环提交协议来导出相同的接口。但是，循环属性维护起来很复杂，并且该协议在故障后需要进行全驱动扫描。相比之下，OFTL 采用与日志结构文件系统 [31] 中的事务支持类似的简单协议，将事务信息保存在页面元数据中，并在更新窗口中跟踪最近更新的闪存块以便快速恢复。

在磁盘存储系统中使用反向指针来避免在 Backlog 中移动文件系统块时的文件系统元数据更新 [24]，或者通过将后向指针嵌入数据块，目录条目和 Backpointerbased 一致性中的 inode 来提供后续一致性检查 [16]。在基于闪存的存储中，LazyFTL [23] 建议通过在页面元数据中保留逻辑页面编号（LPN）作为反向

指针，并将页面记录在更新闪存块区域中，来懒惰地更新页面级 FTL 的映射条目，以块级 FTL 为代价提供页面级 FTL 的性能。但它不会触及文件系统中的元数据，这对写入放大作出了很大贡献。相反，OFTL 利用页面元数据优化系统设计，并使用反向指针作为索引元数据惰性持久性的反向索引。

CAFTL [15] 和 DeltaFTL [33] 合并冗余数据并压缩 FTL 中的相似页面，而 LFS [13], JFFS2 [32], UBIFS [9] 和 btrfs [2] 压缩文件中的数据页系统。他们都通过利用页面内容相似性来使用压缩，这与使用 OFTL 中访问大小信息的压缩更新技术正交。IPL（页内记录）[22] 采用类似于 OFTL 的方法，并在每个闪存块中保留一个日志区域以吸收小的更新。但是，如果数据和元数据页面分布在多个闪存块中，则会受到同步写入的影响。reiserfs [7] 中的尾部打包与 OFTL 中的压缩更新技术最相关。Reiserfs 将每个文件的尾部包装在其 inode 中。OFTL 与 OFTL 以不覆盖的方式更新不同，因此 OFTL 不仅压缩尾部，而且压缩每个写入操作的头部，包括追加和更新操作。

## A.8 结论

传统文件系统专注于顺序存取优化，而不是写入放大缩小，这对闪存非常重要。系统机制（如日志记录，元数据同步和页面对齐更新）极大地放大了写入强度，而间接性带来的透明性阻止了系统利用闪存特性。在本文中，我们提出了一个基于对象的设计名称 OFTL，其中存储管理从文件系统卸载到 FTL 以直接管理闪存。页面元数据用于保留延迟索引的反向索引和事务信息，以便为日志删除提供写入原子性，借助更新窗口跟踪尚未检查点的最新分配的闪存块。另外，空闲空间管理可以跟踪闪存块状态而不是页面状态，并降低状态持久化的频率以进一步降低元数据成本。使用字节单位访问接口，OFTL 中标识的部分页面更新被压缩并与元数据共存以减少更新。在系统与闪存共同设计的情况下，文件系统的写入放大显著减少。

## 致谢

我们要感谢我们的指导者 Margo Seltzer 和匿名审稿人提出的富有洞察力的意见和详细的建议，这大大改进了本文的内容和陈述。我们也感谢 Shuai Li 在实验装置方面的帮助，Guangyu Sun 在演讲中的帮助。本研究由国家自然科学基金（批准号：60925006），国家自然科学基金重点项目（批准号：61232003），国家

高技术研究发展计划（批准号：2013AA013201），清华腾讯互联网创新技术联合实验室研究基金支持。

## 参考文献

- [1] blktrace(8) - linux man page. <http://linux.die.net/man/8/blktrace>.
- [2] Btrfs. <http://btrfs.wiki.kernel.org>.
- [3] Dbt2 test suite. <http://sourceforge.net/apps/mediawiki/osdldb>.
- [4] Fusionio virtual storage layer. <http://www.fusionio.com/products/vsl>.
- [5] Lasr system call io trace. <http://iotta.snia.org/tracetypes/1>.
- [6] The nvm express standard. <http://www.nvexpress.org>.
- [7] Reiserfs. <http://reiser4.wiki.kernel.org>.
- [8] strace(1) - linux man page. <http://linux.die.net/man/1/strace>.
- [9] Ubifs - ubi file-system. <http://www.linux-mtd.infradead.org/doc/ubifs.html>.
- [10] Yaffs. <http://www.yaffs.net>.
- [11] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In Proceedings of USENIX 2008 Annual Technical Conference, 2008.
- [12] S. Boboila and P. Desnoyers. Write endurance in flash drives: measurements and analysis. In Proceedings of the 8th USENIX conference on File and storage technologies (FAST), 2010.
- [13] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In Proceedings of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS), 1992.
- [14] L.P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In Proceedings of the 2007 ACM symposium on Applied computing, 2007.
- [15] F. Chen, T. Luo, and X. Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST), 2011.
- [16] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST), 2012.
- [17] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of nand flash memory. In Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST), 2012.
- [18] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42),

2009.

- [19] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), 2011.
- [20] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. Dfs: a file system for virtualized flash storage. In Proceedings of the 8th USENIX conference on File and Storage Technologies (FAST), 2010.
- [21] Y. Kang, J. Yang, and E. L. Miller. Object-based scm: An efficient interface for storage class memories. In Proceedings of IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), 2011.
- [22] S. W. Lee and B. 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.
- [23] D. Ma, J. Feng, and G. Li. Lazyftl: A page-level flash translation layer optimized for nand flash memory. In Proceedings of the 2011 International Conference on Management of Data (SIGMOD), 2011.
- [24] P. Macko, M. Seltzer, and K.A. Smith. Tracking back references in a write-anywhere file system. In Proceedings of the 8th USENIX conference on File and storage technologies (FAST), 2010.
- [25] M. Mesnier, G.R. Ganger, and E. Riedel. Object-based storage. IEEE Communications Magazine, 41(8):84–90, 2003.
- [26] D. Nellans, M. Zappe, J. Axboe, and D. Flynn. ptrim ()+ exists (): Exposing new ftl primitives to applications. In 2nd Annual Non-Volatile Memory Workshop, 2011.
- [27] P. O’ Neil, E. Cheng, D. Gawlick, and E. O’ Neil. The log-structured merge-tree (lsm-tree). Acta Informatica, 33(4):351–385, 1996.
- [28] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block i/o: Rethinking traditional storage primitives. In Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA), 2011.
- [29] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI), 2008.
- [30] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. Technical report, IBM Almaden Reserach Center, 2012.
- [31] M. Seltzer. Transaction support in a log-structured file system. In Proceedings of the Ninth International Conference on Data Engineering, 1993.
- [32] David Woodhouse. Jffs2: The journalling flash file system, version 2. <http://sourceware.org/jffs2>.
- [33] G. Wu and X. He. Delta-ftl: improving ssd lifetime via exploiting content locality. In Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys), 2012.
- [34] Q. Yang and J. Ren. I-cash: Intelligently coupled array of ssd and hdd. In Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA),

2011.

- [35] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST), 2012.