



哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

AquaFS

开发设计文档

参赛队名	RethinkFS
队伍成员	梁鑫嵘、李羿廷
指导老师	夏文、李诗逸

2023 年 6 月 6 日

摘要

ZenFS 由于其简单且与硬件 Zoned Storage SSD 紧密结合等特点，实现了零硬件预留空间、零硬件垃圾回收开销等高负载场景下的高性能特性，但是其仅适合 ZNS 设备、需要软件特殊适配等特点也限制了其应用场景灵活性。

AquaFS 是一个以 ZenFS 为原型的适用于 Zoned Storage SSD 的文件系统，将原来的 ZenFS 模块化，扩展其应用场景，添加 RAID 来平衡数据安全和写放大，同时添加调参模块以提高文件系统的智能性和性能。

初赛阶段进度情况如表 1 所示。在未来，我们将继续实现 AquaFS 的更多功能，如用户态 NVME 驱动加速、更完善的数据恢复、多文件系统融合调参等，进一步扩展应用场景并提升其智能化水平。

表 1: 初赛进度情况

改进内容	完成进度	完成情况
智能调参模块	进度约 70%	✓基于方差的重要参数选择方案 ✓基于高斯回归的调参方案 □将对垃圾回收参数进行进一步测试 □将对融合文件系统后的更多参数进行调优测试
基于 Zone 的智能动态分区 RAID	整体进度约 80%	✓全盘 RAID 模式 RAID0、RAID1、RAID-C ✓分区 RAID 模式 Zone 映射和 RAID 逻辑分配 ✓数据完整性检测和恢复
异步 IO 优化	进度约 60%	✓完成了 <code>io_uring</code> 异步读写优化 □将进一步研究基于 <code>spdk</code> 、 <code>xnvm</code> 等的用户态 <code>nvme</code> 驱动
融合通用文件系统	进度约 50%	✓基于 FUSE 和 Rust 完成一个 Ext2 兼容文件系统 □将进一步研究与 ZenFS 的结合方式 □将进一步研究智能数据请求分类方法

当前测试结果：使用 RocksDB 随机读写测试来测试 RAID 模块实现的数据正确性（表 3），在不同 RAID 配置下进行 RocksDB 数据库读写和校验并且测试通过；使用 `io_uring` 等优化 IO 后，进行单线程读写测试（表 4），在不同 RAID 配置下能够有效提高整体带宽利用率；对智能调参模块进行测试（图 39），在 RocksDB 随机写入测试中能够根据系统运行数据调整文件系统参数，逐步提高数据吞吐量。

目录

摘要	2
1 概述	5
2 需求分析与调研	6
2.1 需求分析	6
2.1.1 往年实现分析	6
2.2 背景调研	10
2.2.1 Flash 的特点和分类	10
2.2.2 Nand Flash	11
2.2.3 Nor Flash	15
2.2.4 FTL	16
2.2.5 SSD 的性能瓶颈	17
2.2.6 文件系统调参	18
2.3 ZenFS	20
2.3.1 ZenFS 的特点	20
2.3.2 ZenFS 源码分析	23
3 系统设计	33
3.1 AquaFS 整体架构	33
3.2 模块设计细节	34
3.3 开发计划	37
4 系统实现	38
4.1 AquaFS 文件系统的 RAID 实现	38
4.1.1 全盘 RAID 的实现	39
4.1.2 智能分区 RAID 的实现	44
4.1.3 分区 RAID 故障处理	46
4.2 AquaFS 的 IO 加速实现	47
4.3 辅助文件系统 ExtFS 的实现	51
4.4 AquaFS 文件系统的智能调参模块实现	53
4.5 磨损均衡模块的实现	54
5 系统测试	56
5.1 智能分区 RAID 模块和 IO 加速测试	56

5.1.1	AquaZFS 数据完整性测试	57
5.1.2	使用 RAID 0 加速文件读写	59
5.1.3	文件系统数据恢复测试	60
5.1.4	文件系统性能测试	61
5.1.5	ExtFS 功能和性能测试	63
5.2	AquaTurnner 智能调参模块测试	65
6	总结与展望	68
6.1	工作总结	68
6.2	创新点和未来	69
	附录 A 图表索引	72
A.1	图目录	72
A.2	表目录	73
	附录 B 参考文献	74
	参考文献	74

1 概述

AquaFS 是一个以 ZenFS 为原型的适用于 Zoned Storage SSD 的智能化的文件系统，旨在为闪存设备提供更好的性能、更长的寿命和更适应实际应用场景的运行参数。本文档将从多个角度介绍 AquaFS 的设计与实现。

AquaFS 与传统 Ext4 等文件系统相比，有以下特点：

- 与 Ext4 等原地更新的文件系统不同，AquaFS 采用了日志结构，以适应 Flash 结构特性，减少写放大。
- 与传统软件文件系统不同，AquaFS 与硬件结合共同设计，降低硬件成本的同时提升性能。
- 与普通的单一文件系统不同，AquaFS 采用了多系统模块结合的方法，以提升系统的灵活性。
- 设计了专门的参数调整逻辑，以动态适应不同的工作负载。
- 设计了智能的读写逻辑，不同的工作负载能够由不同的模块处理或共同处理，以提升系统的性能。
- 优化垃圾回收逻辑，硬件上取消垃圾回收，软件上可调整预留空间与垃圾回收频率，以均衡调整性能与寿命。

而与 ZenFS 相比，AquaFS 有以下特点：

- **适用场景更广泛：**ZenFS 仅适用于软件特殊适配后的 ZNS 设备，而 AquaFS 可以通过融合文件系统和通用文件接口等方式适用于更多的软硬件设备。
- **更加灵活：**ZenFS 由于其简单而参数较少，且都由上层软件适配调整，而 AquaFS 可以通过调整参数、融合文件系统等方式提升其场景适用性，为没有软件适配的应用场景提供支持。
- **更加智能：**ZenFS 许多逻辑都是固定的，而 AquaFS 通过智能调参、智能分类读写等方式提升其智能化水平。此外，AquaFS 还可以通过文件系统级冗余、文件系统读写分离、inode 缓存等方式提升其智能化水平。
- **更加安全：**ZenFS 在写入记录层面进行校验，但是对更常见的整块数据损坏无法有效应对，而 AquaFS 通过 RAID 等方式提升了针对 Zones 的数据安全性。
- **更加高效：**ZenFS 仅有 Direct IO 模式，而 AquaFS 支持 `io_uring` 等更高效的异步 IO 模式。

后文将结合背景调研、需求分析、系统设计、系统实现、系统测试等方面，对 AquaFS 的各种设计特性进行详细介绍。

2 需求分析与调研

2.1 需求分析

根据题目描述，我们需要实现一个“更加智能的 Flash 文件系统”。

Flash 介质因本身的擦除特性，给上层文件系统带来与普通磁盘、内存文件系统不同的数据管理模式。同时，Flash 的写入放大、寿命以及后期稳定性下降等问题也给文件系统的设计带来的一定的挑战。传统的 Flash 文件系统并没有很好地解决这些稳定性相关问题。这里希望寻找一种更智能、更合适的 Flash 文件系统设计，来更好地平衡 Flash 的性能与稳定性。可以思考的方向包括但不限于：数据压缩算法（可以是自适应的压缩算法），检错、纠错、纠删码（可以是联合信源信道编码），数据块选择、擦除策略，Cache 机制。

结合赛题内容，我们需要完成的文件系统需要具备以下特性：

- **适合 Flash 介质：**文件系统需要能够适应并缓解 Flash 的写入放大、寿命、后期稳定性等诸多 Flash 存储介质特有的问题
- **更智能：**文件系统需要使用多种策略，在软件算法层面平衡性能与寿命

2.1.1 往年实现分析

本题在去年已经有队伍选题并完成^[1]，在选题之时，我们对前辈的项目进行了调研。他们队伍完成了一个基于 UBIFS 的适用于裸 Flash 设备的 YUBIFS 文件系统，从以下几个方面对其进行了优化：

- **数据压缩模块：**使用了预压缩和自适应压缩结合的方法，均衡压缩比和写入速度
- **纠错编码模块：**实现自适应生命周期、CRC+RAID5 两种纠错方案
- **Cache 机制：**添加读写缓冲
- **冷热数据识别：**使用了冷热识别的纠错以提高 I/O 速度

YUBIFS 的系统架构如图 1 所示。看起来 YUBIFS 的实现已经非常满足题目的需求，能够很好地回答题目中提出的几个问题。但是我们发现，YUBIFS 的系统实现也存在一些不足之处：

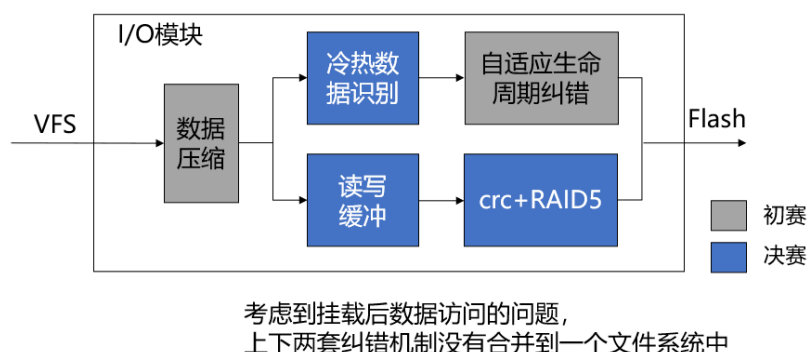


图 1: YOUBIFS 系统架构

• Nor Flash 应用场景

YOUBIFS 的实现中，文件系统的实现是针对于 Nor Flash 的。Nor Flash 在实现上与 Nand Flash 有很大的不同，因此 YOUBIFS 的实现并不能直接应用于 Nand Flash 上，应用场景受到了限制。

同时，Nor Flash 一般用于嵌入式设备，而 Nand Flash 一般用于移动设备或高性能设备，因此 YOUBIFS 的实现也不能直接应用于移动设备或固态硬盘上。YOUBIFS 针对于 Nor Flash 的多种写入情况做了优化，但是由于 Nor Flash 的寿命、速度、稳定性等特性，其并不适合在写入负载较高的情况下工作。

Nor Flash 的寿命一般在 10 万次左右，而 Nand Flash 的寿命一般在 100 万次左右^[2]，因此 Nor Flash 广泛用于 BIOS 存储、固件存储等写入负载较低的场景。在这些应用场景下，由于存储的数据都是非常重要的系统关键数据，如 Bootloader、系统固件等，因此对于数据的稳定性要求非常高。如果系统软件直接在 Nor Flash 上频繁读写，很可能会导致 Nor Flash 的寿命过早耗尽，从而导致系统无法正常启动，造成非常严重的后果，所以 Nor Flash 很少用于存储程序运行过程中产生的数据，常见的写入场景为固件更新、系统升级等。

因此，YOUBIFS 的实现并不能直接应用于移动设备或固态硬盘上，所针对的 Nor Flash 高压写入的应用场景也非常有限。这与 YOUBIFS 的各种优化策略相矛盾，因此我们需要重新设计一个适用于 Nand Flash 的文件系统，以扩展更多的应用场景。

• “智能”，但是还不够智能

YOUBIFS 的实现中，使用了多种策略来平衡性能与寿命，但是这些策略都是相对固定的，只是能够根据不同的情况进行策略切换。其中能够体现“智能”的实现有：

1. 通过判断文件名的方式来判断使用的压缩算法和参数，即压缩算法和等级的自适应

2. 通过检查压缩效果是否合适来判断是否继续压缩，压缩效果不好可能其数据本身就是压缩文件，则再次不压缩
3. 利用文件系统的 `node`，判别冷热数据文件

为什么说我们认为这些“智能”策略其实还不够智能？

这些策略的优化依据，都是文件系统的当前状态、当前负载请求分类等，没有充分考虑 Flash 的介质特性，如后期稳定性下降和延迟升高等，也没有充分考虑工作负载、实际应用和文件系统的互相配合。也就是说，考虑的因素还是不够多，对 Flash 的针对性优化也不够深入。

同时，这些策略都是非常简单固定的逻辑。虽然通过参数上的调整，在性能受限制的嵌入式系统中也能够在指定的负载上取得比较好的效果，但是在性能限制不大、更复杂的系统中就很难有出色的表现。这是由策略本身的简单的性质决定的，复杂的系统往往需要更复杂的调整逻辑，而这就需要更加复杂的算法，从简单的逻辑判断到决策树、线性回归、神经网络等，用更加复杂和智能的策略逻辑来适应更加复杂的系统和更高要求的工作环境。

• 测试和展示不够规范

首先是测试环境上的问题。录制的视频中的测试环境是个人计算机，目标设备是计算机上的 SSD 或者内存，而不是项目考虑中的 Nor Flash。

既然是适用于 Flash 的文件系统，就应该统计和 Flash 相关的数据，从而衡量这个文件系统对 Flash 的稳定性、寿命、速度的综合影响，但是测试中只是在本机进行了速度的测试，这难以说明问题。为了方便开发和开发过程中的验证，最好应该建立一个 Flash 仿真程序，真正计算在 Flash 中的读写频率、位置等信息，并模拟 Flash 芯片的具体延迟给出具体读写速度。

其次，对 Flash 芯片的检错、纠错测试也应该基于 Flash 仿真，而不是在内存中写入值。在内存中改变一个或多个字节或比特，可能并不符合实际情况中的数据损坏场景。

在决赛的文档中，记录了在实际嵌入式场景下在一个 Nor Flash 芯片上运行 YOUBIFS 的相关数据，但是没有体现在录制的展示视频中。演示视频中，不断切换测试脚本和开发环境，并通过切换分支的方式切换功能和版本。这样看视频的人很难看懂在做什么，看懂了也很难说明当前的设计有哪些提升。如果演示的时候就能直观地用自动化脚本生成测试报告，如表格、统计图等，会更加有表现力和说服力。没有使用通用的测试脚本或者软件，而是选择自己写脚本测试，难以说明提升。测试中大部分测试使用 311MiB 大小的连续读写，或者直接使用上百 MiB 的大文件连续读写，这不仅不能说明问题，而且也没有解决实际问题。在复杂的文件读写环境

Language	files	blank	comment	code
C	48	6044	12432	30124
C/C++ Header	13	713	4033	4173
SUM:	61	6757	16465	34297

表 2: YOUBIFS 代码统计

中，大部分读写应该是 4KiB 随机读写，这在其他的文件系统论文中是最重要的指标之一，视频中没有体现。

• 纠错算法和 UBIFS 本身不太兼容

UBIFS 运行在 UBI 层之上，而 UBI 层本身已经做了一层校验和纠错，在上一层遇到一个两个比特级别的错误的可能性并不大，反而是一整块、一整个存储介质完全失效的可能性更高。而 UBIFS 本身的纠错算法是基于比特级别的，这样的纠错算法在 UBI 层已经做了纠错的情况下，就显得多余了。而且，UBIFS 的纠错算法是基于比特级别的，而 UBI 层的纠错算法是基于块级别的，这样的不兼容性也会导致 UBIFS 的纠错算法的效果不好。

此外，YOUBIFS 中有 CRC+RAID5 的逻辑，但是其只适用于单个 Nor Flash 存储设备，在 UBI 层已经实现校验和纠错的情况下，基于 UBI 层的 YOUBIFS 更应该关注以块或设备为单位的 RAID，而不是以比特为单位的 CRC+RAID。

找出这些不足，我们并非为了挑刺，而是为了找到我们的项目前进的方向，摸着石头过河。除了这些不足之处，我们同样研究了前人项目的各种其他特点。

1. **文档量很大**：文档中需要包含整个开发流程，需要列举并详细说明涉及到的技术要点和原理，以及配有丰富的示意图和代码段。在前人队伍的文档实现中，很大部分是分析原版 UBIFS 的实现逻辑和代码理解，另外一部分有基于这些理解对技术进行的调研以及自己的实现逻辑。
2. **总代码量很大**：原版 UBIFS 的代码量就有 1.5 MiB，所以对代码阅读理解能力也有很高的要求。去除重复文件后，使用 CLOC 统计文件行数如表 2 所示，队伍自己实现的部分暂未统计。

在 YOUBIFS 已经能够完成大部分题目要求的情况下，我们要如何在这个基础上进行改进呢？

依照上面的分析，首先我们可以改变我们文件系统的存储介质。YOUBIFS 使用的是 Nor Flash，那么我们可以选择 Flash 的另一种更加常用更加高性能的品类：Nand Flash。我们的研究基于 Nand Flash，就可以更加贴近实际应用场景，扩展项目的实际用途，也可以更加容易地和其他的文件系统进行对比。

其次，当我们选择了 Nand Flash，就多了一个技术实现上的选择。在许多较为低端的嵌入式设备上，主控是直接控制存储设备的绝对地址的。在 Linux Kernel 中，这样直接控制地址的设备被称为 MTD 设备（Memory Technology Device）。除了 MTD 设备，还有一种方式是硬件层面实现一个从逻辑地址到绝对地址的映射，从而完成磨损均衡、垃圾回收等功能。这一层转换层叫做 FTL（Flash Translation Layer，Flash 地址转换层），在许多消费级、商业级服务器或者高端嵌入式设备上都会使用 FTL。在 Linux Kernel 中，这样的设备往往分类为块设备，而不是 MTD 设备。

	MTD (NAND)	FTL
Interface	Direct interface to NAND controller (SoC)	Block device interface (USB mass storage, SD, ...)
NAND flash compatibility	Dependent on NAND controller (SoC)	Dependent on FTL controller
Bad blocks, Bit-flips, Endurance	Not managed →Upper layer must manage them.	Managed by FTL controller →Upper layer does not have to manage them.
File systems	JFFS2, YAFFS2, UBIFS, ...	FAT, EXT3, EXT4, ...

图 2: MTD 设备和有 FTL 的设备的对比

MTD 设备和有 FTL 的设备的对比如图 2 所示。在我们的项目中，我们选择了基于 Nand Flash 的块设备，而不是基于 Nand Flash 的 MTD 设备，这样可以更加贴近实际应用场景，扩展项目的实际用途，也可以更加容易地和其他的文件系统进行对比。

我们选择基于 Nand Flash 与 FTL 的文件系统，题目需求中的许多方面都出现了新的需求。首先，Nand Flash 的读写单位是页，而 Nor Flash 的读写单位是字节，这就要求我们的文件系统在读写时要考虑到页的边界对齐问题。其次，Nand Flash 之上的一层 FTL，向上隔绝了数据块选择、数据擦除、逻辑物理地址映射等与 Flash 存储介质相关的细节，从而提高了在文件系统层面针对 Flash 进行优化的难度。最后，Nand Flash 的读写速度比 Nor Flash 快，但是擦除速度比 Nor Flash 慢，这就要求我们的文件系统在设计时要考虑到这一点，尽量减少擦除操作的次数。

为了解决这些问题，我们需要对 Nand Flash 的特性进行调研，然后在一种基于 Nand Flash 的文件系统的基础上进行改进。

2.2 背景调研

在我们选定赛题后，我们立刻开始了针对 Flash 文件系统实现的调研。

2.2.1 Flash 的特点和分类

为了实现针对 Flash 优化的文件系统，我们必须首先了解 Flash 的各种特性和分类。

闪存（Flash Memory）是由日本的舛冈富士雄（Fujio Muoka）发明的。他分别于 1966 年和 1971 年从日本东北大学（Tohoku University）获得学士和博士学位，博士毕业之后他加入了东芝（Toshiba）公司。在东芝工作期间，他分别于 1980 年和 1988 年发明了 NOR Flash 和 Nand Flash。

目前市面上的闪存主要有两种：NOR Flash 和 Nand Flash，其分类逻辑和原理如图 3 所示。一般而言，并行接口的 Parallel Flash 是基于 NOR Flash 的；而 SSD 硬盘，U 盘，SD 卡，eMMC 等通常是基于 Nand Flash 的。

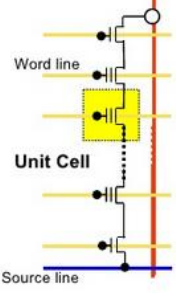
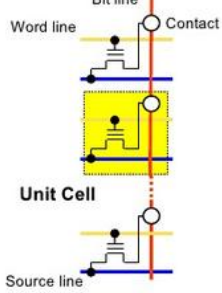
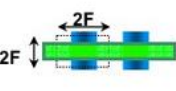
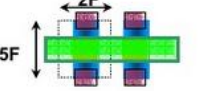
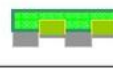
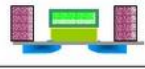
	NAND	NOR
Cell Array		
Layout		
Cross-section		
Cell size	4F²	10F²

图 3: Flash 原理与分类

2.2.2 Nand Flash

Nand Flash 是一种非易失性存储器，具有访问速度快、存储密度高、成本低的特点。Nand Flash 的读写操作都是以页为单位进行的，而且每次写操作都需要先擦除整个块，这使得 Nand Flash 更适合用于存储大量数据，而不适合用于执行代码和固件更新等应用。然而，Nand Flash 的读写速度快，成本低，是目前大多数嵌入式设备或消费级、商业级电子产品的主要存储介质。

为了详细了解 Nand Flash 的多种特性，我们专门针对一款常用的 Nand Flash 进行了调研。这款 Nand Flash 的型号是 K9F1G08U0F-5IB0。这是一款来自三星的 SLC Nand，容量为 1Gb，其主要特性如图 4 所示。

在读写操作失败时，Nand 控制器需要进行一些特殊的处理。将 Datasheet 中的处理方式翻译为中文：在 NAND Flash 存储器中，其使用寿命内可能会出现额外的无效块。请参考资格报告以获取实际数据。应考虑以下可能的故障模式以实现高度可靠的系统。

- Voltage Supply
 - V_{CC} : 3.3V (2.7V ~ 3.6V)
 - Organization
 - Memory Cell Array : (128M + 4M) x 8bit
 - Page Size : (2K + 64)Byte
 - Data Register : (2K + 64) x 8bit
 - Block Erase : (128K + 4K)Byte
 - Automatic Program and Erase
 - Page Program : (2K + 64)Byte
 - Page Read Operation
 - Random Read : 25 μ s(Max.)
 - Serial Access : 25ns(Min.)
 - Data Transfer Rate : SDR 20Mhz(40Mbps)
 - Fast Write Cycle Time
 - Page Program time : 400 μ s(Typ.)
 - Block Erase Time : 4.5ms(Typ.)
 - Command/Address/Data Multiplexed I/O Port
 - Hardware Data Protection
 - Program/Erase Lockout During Power Transitions
 - Command Driven Operation
 - Unique ID for Copyright Protection
 - Package :
 - K9F1G08U0F-Sx¹⁾B0 : Pb-Free, Halogen-Free Package
48 - Pin TSOP1 (12 x 20/0.5 mm pitch)
 - K9F1G08U0F-5x¹⁾B0 : Pb-Free, Halogen-Free Package
63 - FBGA (9 x 11 / 0.8 mm pitch)
- NOTE :**
 1) C : Commercial
 I : Industrial
 F : Automotive - Grade3
 H : Automotive - Grade2

图 4: K9F1G08U0F-5IB0 特性

在擦除或编程后状态读取失败的情况下，应进行块替换。因为在页面编程期间的编程状态失败不会影响同一块中其他页面的数据，所以可以通过找到一个已擦除的空块，并重新编程当前目标数据并复制替换块的其余部分，来执行块替换。在读取时必须使用 ECC。为了提高内存空间的效率，建议通过 ECC 回收由于单个位错误而导致的读取或验证失败，而无需进行任何块替换。所述附加块故障率不包括那些被回收的块。写入时错误处理流程如图 5 所示。

Nand 由于其以页面为写入单位的特性，使得其在写入时的错误处理流程较为复杂。在写入时，如果发生错误，需要进行块替换。块替换的过程如图 6 所示。

由于 Nand Flash 只能以页面为单位进行写入，所以在一个块内，页面必须从该块的 LSB（最低有效位）页面到该块的 MSB（最高有效位）页面依次编程，禁止随机页面地址编程。在这种情况下，LSB 页面的定义是要编程的页面中的 LSB。因此，LSB 不需要是页面 0。

这个用于举例的 Flash 芯片已带有 ECC 功能。在编程操作期间，内部 ECC 逻辑生成 ECC。在读取操作期间，设备会自动执行 ECC。读取操作执行后，可以发出读取状态命令以识别读取状态，读取状态保持不变，直到执行其他有效命令。这个 Nand Flash 可以自己纠正 4bit 的错误。

即，这款 Nand 芯片内的控制器能够保证在读取时，能够自动执行 ECC，并且能够自动纠正 4bit 的错误。这样，我们在读取时，基本不需要考虑较少比特位的 ECC 的问题。当 Nand 芯片内的控制器检测到无法纠正的 ECC 错误，将会返回错误信息，然后可以根据这个错误信息，进行块替换。如果块替换失败，那么这个块就会被标记为坏块，不再使用。

由此得知，一般 Nand 芯片向外提供的数据接口可以保证数据的正确性，从而上层软硬件可以依赖于 Nand 上的硬件 ECC 提供的数据正确性保证。但是这是在 Nand 使能了 ECC 功能的情况下，如果 Nand 芯片没有使能 ECC 功能，那么上层软硬件就需要自己实现 ECC 算法，来保证数据的正确性。此外，硬件 ECC 功能需要占用额外的磁盘空

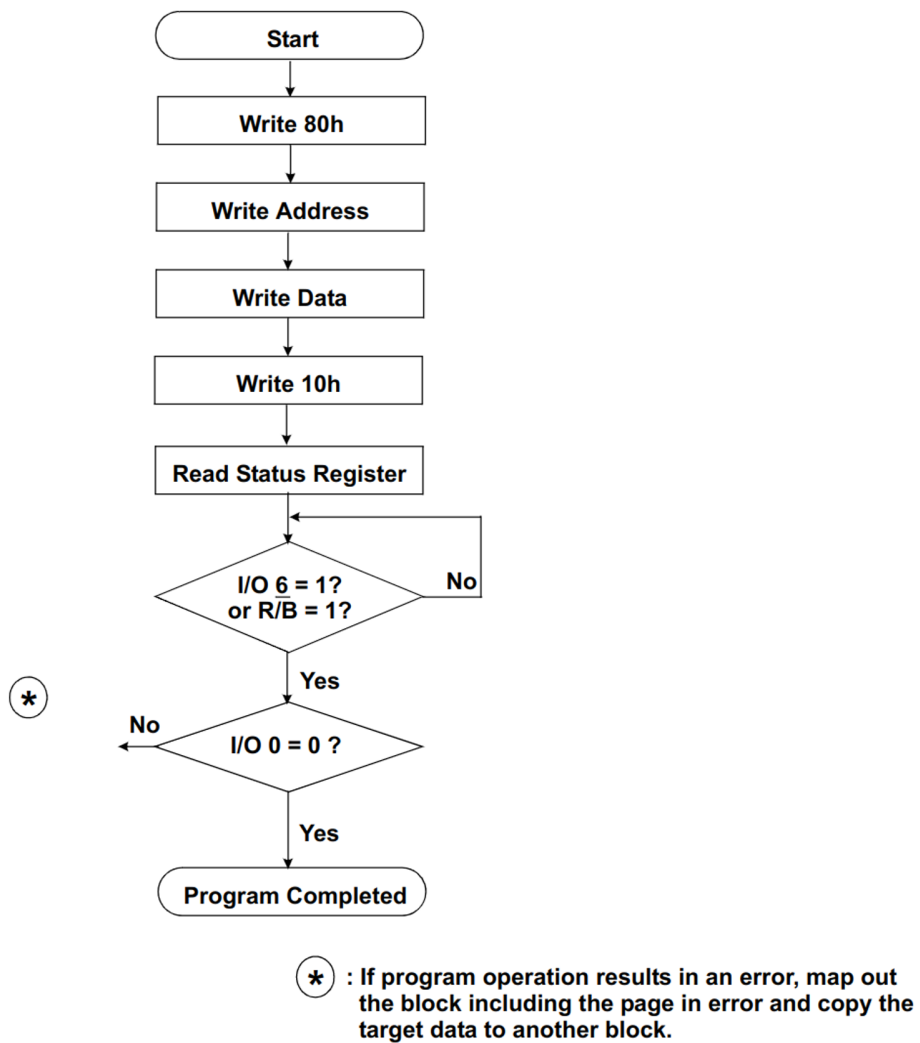


Figure 4. Program Flow Chart

图 5: 写入时错误处理流程

Block Replacement

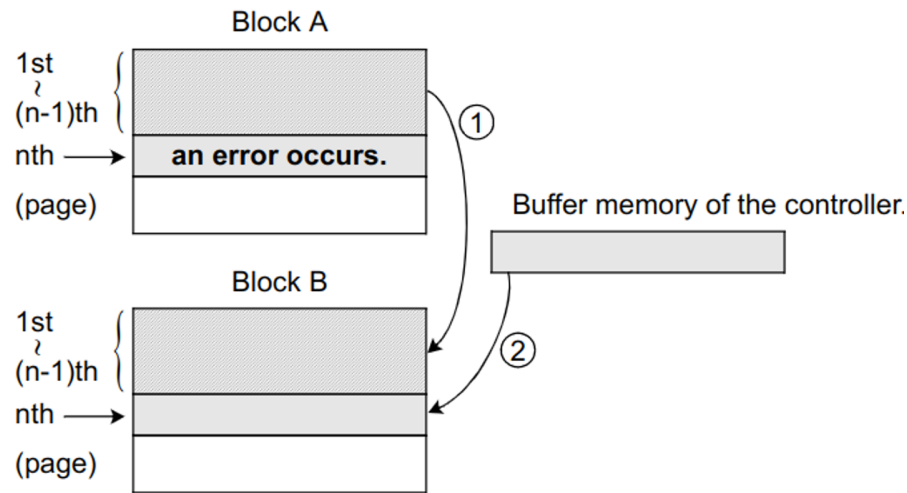


图 6: Nand 块替换流程

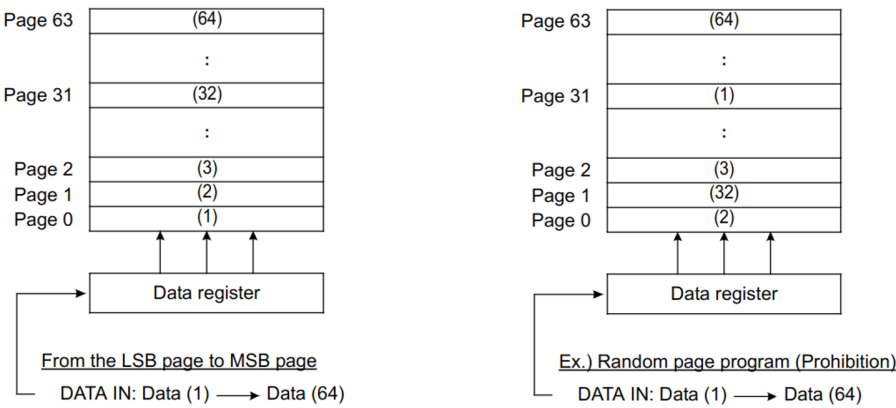


图 7: Nand 写入寻址举例

间，因此，如果对磁盘空间有较高要求，那么可以考虑不使能硬件 ECC 功能。

2.2.3 Nor Flash

NorFlash 是一种非易失性存储器，具有访问速度快、读写寿命长、可靠性高的特点。与 Nand Flash 不同，NorFlash 的读操作可以以随机方式进行，而不需要预先擦除整个块。这使得 NorFlash 更适合用于执行代码和固件更新等应用，可以加速启动时间和提高系统性能。然而，NorFlash 的密度较低，成本较高，不适合用于存储大量数据。

NorFlash 的一些其他特点包括：

- **高度可靠：**Nor Flash 可以实现数据的可靠存储和读取，并且具有高度可靠性。
- **可以执行代码：**Nor Flash 可以作为代码存储器，可以加快启动时间并提高系统性能。此外，Nor Flash 还支持 XIP（执行内存），它允许 CPU 直接从 Nor Flash 中执行代码，而无需将代码加载到 RAM 中。
- **读写寿命：**Nor Flash 的读写寿命比 Nand Flash 长得多，因此它更适合于需要高度可靠性和长寿命的应用程序。
- **错误概率：**Nor Flash 读写操作的错误概率很低，这使得它在数据存储和传输方面更加可靠。

Nand flash 和 Nor flash 的性能比较

flash 闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何 flash 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND 器件执行擦除操作是十分简单的，而 NOR 则要求在进行擦除前先要将目标块内所有的位都写为 0。由于擦除 NOR 器件时是以 64~128KB 的块进行的，执行一个写入/擦除操作的时间为 5s，与此相反，擦除 NAND 器件是以 8~32KB 的块进行的，执行相同的操作最多只需要 4ms。执行擦除时块尺寸的不同进一步拉大了 NOR 和 NADN 之间的性能差距，统计表明，对于给定的一套写入操作（尤其是更新小文件时），更多的擦除操作必须在基于 NOR 的单元中进行。

Nand Flash 与 Nor Flash 的主要区别如下：

- NOR 的读速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的 4ms 擦除速度远比 NOR 的 5s 快。
- 大多数写入操作需要先进行擦除操作。

- NAND 的擦除单元更小，相应的擦除电路更少。

我们最终选定了 Nand Flash 作为我们的存储介质，主要是因为 Nand Flash 的价格更低，而且 Nand Flash 的读写速度更快，更适合于我们的应用场景，同时与其他赛组的工作能保持一定的区分度。

2.2.4 FTL

FTL (Flash Translation Layer, 闪存地址翻译层) 是广泛存在于 Nand Flash 存储层次之上的一层地址翻译逻辑，它的主要作用是将逻辑地址转换为物理地址。

FTL 通常是由 Nand Flash 的厂商提供的，因此，不同厂商的 FTL 实现可能会有所不同，但是它们的主要功能都是相同的，即地址映射。原因是闪存只能异地更新，为了对上支持数据块原地更新则需要通过地址转换实现。由于闪存先擦后写、擦写有次数限制（寿命）、使用过程中会不断出现坏块（块寿命不同）等特性，FTL 还需具备垃圾回收、磨损均衡、坏块管理等十八般武艺。

闪存内部的基本存储单位是 Page (4KB)，N 个 Page 组成一个 Block。FTL 的映射方式主要有：

1. 块级映射

将块映射地址分为两部分：块地址和块内偏移。映射表只保存块的映射关系，块内偏移直接对应。

2. 页级映射

映射表维护每个页的映射关系。

3. 混合映射

主要思路是针对频繁更新的数据采用页级映射，很少更新的数据采用块级映射。其中采用 Log Structured 思想的混合映射将存储分为数据块 (Data Block) 和日志块 (Log Block)。数据块用于存储数据，采用块级映射，日志块用于存储对于数据块更新后的数据，采用页级映射。混合映射是低端 SSD、eMMC、UFS 广泛采用的映射方式。根据日志块和数据块的对应关系又可以分为全相关映射 (FAST)、块相关映射 (BAST)、组相关映射 (SAST) 等等。下图是 SAST 映射的一个示例：2 个日志块对应 4 个数据块，当日志块用完时需要通过搬移有效数据回收日志块。对于顺序写场景，最好情况下日志块对应位置记录了数据块的更新，则可以无需搬移数据，直接将日志块作为新的数据块，数据块进行擦除操作作为新的日志块。对于大量随机写场景，则需要将日志块和数据块中的有效数据搬移到空闲块的对应位置作为新的数据块，然后擦除原日志块和数据块。

由于 FTL 一般实现在 SSD/EMMC 主控中，作为一个黑盒存在，因此上层软件无法得知 FTL 的具体实现，也无法对其进行优化。举个例子，如果上层软件知道 FTL 的映射方式是页级映射，那么它就可以将多个小文件合并成一个大文件，这样就可以减少 FTL 的映射表的大小，从而提高性能。但是上层软件往往并不知道 FTL 的映射级别和映射表的大小，因此无法进行优化；如果软件贸然进行优化，反而可能会触发 FTL 的垃圾回收机制等，从而造成更大的写放大和性能降低。

此外，FTL 的映射逻辑需要占用一定的存储空间，这也是 FTL 的一个缺点。加上主控的垃圾回收功能，一款 SSD 往往有 10% 到 20% 的空间是不可用的，这也是 SSD 的一个缺点。

但是，如果将有一款 SSD 可以将底层尽量展现给上层而不是完全的黑盒，就可以获得控制分配读写的能力，可以用自己的逻辑来实现软 FTL。

2.2.5 SSD 的性能瓶颈

SSD 的性能瓶颈涉及多个方面：

首先，闪存芯片本身的读取和写入速度是一个重要的性能瓶颈。尽管闪存技术不断进步，但相对于传统的机械硬盘，闪存的读取和写入速度仍然有限。特别是对于随机的小块读写操作，闪存的延迟和响应时间可能较高。

其次，闪存芯片的擦写操作是一项耗时的操作，因为擦除一个块通常需要先将其数据复制到其他位置，然后进行擦除操作。这导致了写入操作的额外开销和延迟，尤其在频繁的写入场景下，擦写操作的成本会进一步增加。

另外，闪存芯片的寿命限制也是一个性能瓶颈。闪存芯片具有有限的擦写次数，每次擦写操作都会减少芯片的寿命。当某些块的擦写次数达到上限时，需要进行垃圾回收和数据迁移操作，这可能导致额外的延迟和性能损耗。

存储控制器也对 SSD 的性能起着重要作用。存储控制器负责管理闪存芯片、处理 IO 请求、执行错误校验和纠正（ECC）等功能。较低性能的存储控制器可能成为整个系统的瓶颈，限制了 SSD 的性能表现。

最后，接口和总线的带宽限制也可能影响 SSD 的性能。常见的接口如 SATA、PCIe、NVME 等，其带宽和传输速度对于数据的读取和写入速度有一定影响。如果接口和总线的带宽无法满足 SSD 的性能要求，可能会限制 SSD 的吞吐量和响应时间。

我们选定了需要做基于 Nand Flash 的文件系统，就必须要直面以上的问题，我们需要在文件系统层面上解决这些问题，使得文件系统能够更好的利用 Nand Flash 的特性，提高文件系统的性能。

经过调研，我们发现了许多现有的基于 Nand Flash 的文件系统，如 Btrfs、LogFS、F2FS 等。这些文件系统都是为了解决 Nand Flash 的特性而设计的，它们都有各自的优缺点，但是都是在当下已经有完善的 FTL 映射的情况下去解决这个问题。

如果有一种“文件系统”，能够在上层软件和下层硬件之间沟通协调，让软硬件各自更加各司其职，做更适合做的事情，那么能不能解决上面提到的诸多困难呢？

于是乎，我们找到了 ZenFS。这是一个简单的“文件系统”，但是它却能够做到保持最高性能写入不掉速，能做到 100% 的空间利用率，能做到最高的数据库随机读写速度和最低的读写延迟。对这样一种“文件系统”，我们立刻产生了极大的兴趣。对 ZenFS 的调研，详见第 2.3 章的介绍。

2.2.6 文件系统调参

在数据库和文件系统这些存储系统中实现良好的性能并非易事，因为它们是非常复杂的系统，具有许多可调选项，这些选项几乎控制其运行时操作的许多方面。在这些系统中，配置参数是一个重要的问题。存储系统通常有许多影响其行为的参数，调整这些参数可以显著提高性能。由于大量的参数和可能的配置呈指数级增长，手动和自动调整方法都很费力。

鉴于此，许多组织会聘请昂贵的专家来配置系统的参数。但是随着应用程序在规模和复杂性方面的增长，优化参数以满足应用程序的需求已经超越了人类的能力。这是因为参数的正确配置在很大程度上取决于许多人所无法推理的因素。

于是，我们预期实现一个智能调参模块来使得作为 RocksDB 插件的文件系统 AquaFS 具有自动调整本身参数的能力，以使得文件系统更加智能，减少可能的人力开销。

在智能调参方面主要设计到两个方面，参数选择和参数调整。

在参数选择方面，存储系统通常有许多影响其行为的参数。存储系统是现代计算机系统的关键组件，对应用程序性能和效率有重大影响。大多数存储系统都有许多可配置的参数来控制和影响它们的整体行为。例如，Linux 的 Ext4 提供了大约 60 个参数^[3]，代表超过 1037 种潜在的配置状态。默认设置通常不是最优的；有研究表明，调整存储参数可以将系统性能提高多达 9 倍^[4]。

为了应对大量可能的配置，系统管理员通常专注于使用他们的领域专业知识来调整一些经常使用和经过充分研究的参数，这些参数被认为会显著影响系统性能。然而，面对日益增加的复杂性，这种手动调整方法并不能很好地扩展。现代存储系统使用不同的文件系统类型（EXT4，XFS，Btrfs）、新硬件（SSD、SMR、NVM）、多层和混合存储，和多个虚拟化层（例如 LVM、RAID）。存储系统范围从一个或几个相同的节点到数百个高度异构的节点在参数调整方面。

最近，已存在多种优化方法来自动调整存储系统，在合理的时间范围内实现了良好的性能改进^[5-6]。这些自动调整技术将存储系统建模为黑匣子，反复尝试不同的配置，测量目标函数的值，并根据先前学习的信息选择新的配置进行尝试。然而，许多黑盒自动调整技术难以扩展到高维，并且可能需要很长时间才能收敛到好的解决方案。因此，处理大量存储参数配置的问题在很大程度上仍未解决。

在机器学习和信息论中，降维是处理大型数据集的常用技术。如果它可以应用于存储系统，它将大大减少搜索空间，使人类或算法更容易调整存储系统。以前的研究中提到，并非所有存储参数都具有同样重要的性能影响：一些参数比其他参数具有更大的影响^[5]。这自然促使我们研究如何量化每个存储参数的影响或重要性，以及如何有效地选择重要参数。

现在存在许多方法来解决维数灾难的问题，也即降维方法。降维方法通常可以概括为两类：特征提取和特征选择。

特征提取是指将高维数据投影到低维空间；新构建的特征通常是原始特征的线性或非线性组合。常见的特征提取方法包括主成分分析 (PCA)、独立成分分析和线性判别分析。特征提取的一个主要缺点是每个特征的物理意义都会因投影和许多维度的非线性组合而丢失。因此，常见的特征提取技术与本文中的目标相冲突：即选择一些可以理解和解释的原始存储参数。

相反，特征选择直接从原始特征中选择一个特征子集，目的是只找到重要的特征。特征选择方法可以分为监督或非监督方法。无监督特征选择，例如 Principle Feature Analysis，根据特征之间的关系选择包含大部分基本信息的子集。在选择阶段不考虑特征对优化目标的影响。相反，有监督的特征选择选择一个可以区分或近似目标变量的子集。其中包括基于决策树的算法等等。由于我们有兴趣找到对我们的优化目标有重大影响的参数，例如 I/O 吞吐量，因此监督特征选择最适合我们的需求^[7]。

由于在存储系统中存在许多离散参数，将这些离散参数抽象成离散值的质量决定了特征选择的质量，将具有 N 个值的分类参数转换为 N 个单独的二进制参数会使参数空间呈指数级扩展。因此本文并没有采用 lasso 回归，lasso 回归的计算成本很高。

另一种流行的特征选择方法是建立在信息论基础上的方法。这些方法通常为某些子集中目标变量的同质性定义一个度量。常用的指标包括连续变量的方差等。在本文中，我们采用基于方差的重要参数选择方案，方差在系统中的对比可以凸显出参数的重要性。

在参数调整方面，是对于参数选择中的重要参数进行参数的调整。

在之前的对数据库进行自动调参的研究中^[8]，对数据库的表征负载进行提取，在匹配了相似负载，并用 lasso 回归对参数进行去冗余降维之后，采用了高斯过程回归的方案进行参数的调整。高斯过程回归是一种先进的技术，其功能与神经网络的功能大致相当。高斯过程回归有许多吸引人的特性，使其成为配置空间建模和提出建议的合适选择。最重要的是，一个是高斯过程回归默认提供置信区间。在该研究中采用了两种方案提供推荐的参数，其中一种是直接匹配相似负载中的最好配置，另外一种是通过回归的配置。本文中尝试使用两种方案。

2.3 ZenFS

2.3.1 ZenFS 的特点

ZenFS 是一个 RocksDB 的文件系统插件，它利用 RocksDB 的文件系统接口将文件放置到原始分区块设备上的区域中。通过将文件分成多个区域并利用写入生命周期提示来共同定位具有相似生命周期的数据，与传统的块设备相比，系统写入放大大降低了。ZenFS 确保文件系统或磁盘上没有后台垃圾收集，从而提高吞吐量、尾部延迟和磁盘耐久性方面的性能（图 9）。ZenFS 的整体架构如图 8 所示。^[9]

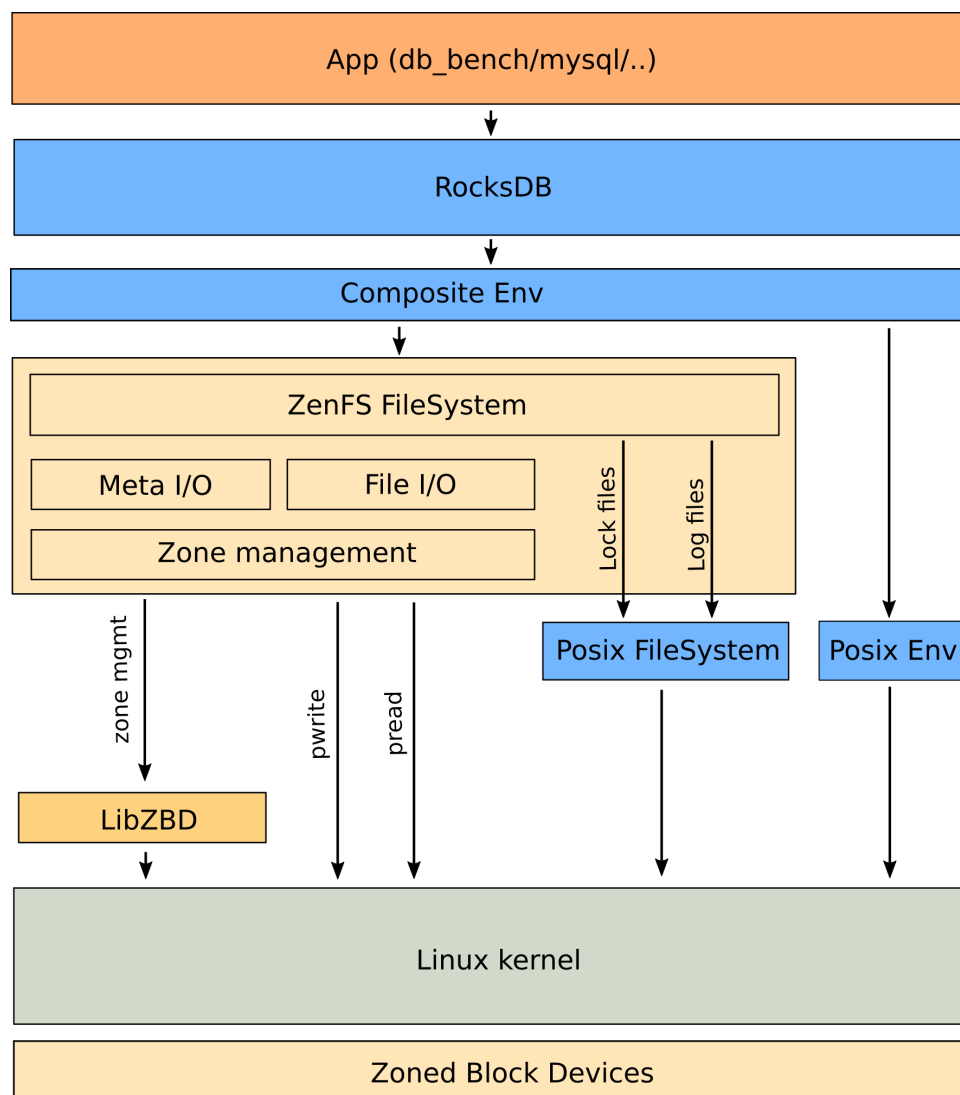


图 8: ZenFS 架构

ZenFS 是一种特殊的“文件系统”。与其说是一种文件系统，不如说是一种全新的软硬件协同的系统的一部分。它通过尽量将原本不可知的 FTL 映射和垃圾回收的过程暴露给上层，使得上层可以更好地利用 SSD 的特性。ZenFS 通过将文件分成多个区

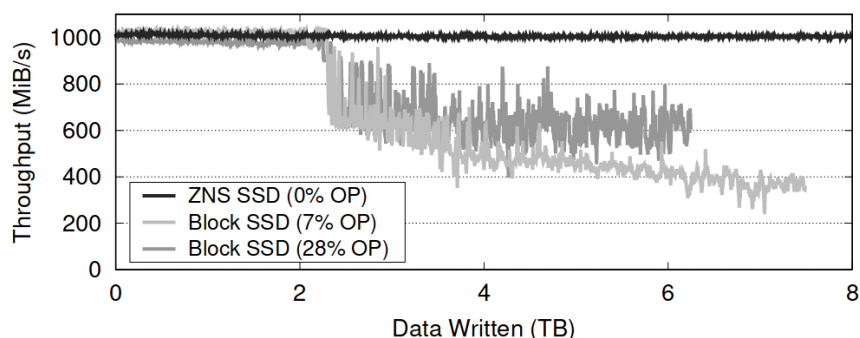


图 9: ZNS 多线程完全覆盖写入负载下吞吐量表现

域并利用写入生命周期提示来共同定位具有相似生命周期的数据，与传统的块设备相比，系统写入放大大降低了。ZenFS 确保文件系统或磁盘上没有后台垃圾收集，从而提高吞吐量、尾部延迟和磁盘耐久性方面的性能。

硬件方面，ZenFS 基于 Zoned Storage SSD 和 ZNS (Zoned Namespace SSD)。

ZNS 是 NVMe 最新标准的一部分，它是专用于特殊 SSD 的通用接口，能够将更多的信息暴露到主机上，利用主机的应用场景信息来优化 SSD 的性能。而 Zoned Storage SSD 就是专门为 ZNS 而设计的 SSD，它将 SSD 的内部划分为多个区域，每个区域都有自己的写入指针，不支持随机写入而是只支持从当前写指针写入，从而尽可能地避免了底层主控上的逻辑映射和垃圾回收。普通 SSD 的主控掌握了逻辑地址到物理地址的映射，并且在主机不可控的情况下执行垃圾回收（尽管有 Trim 命令的支持^[10]），而 Zoned Storage SSD 的垃圾回收机制是由主机来控制的，这样就可以最大程度上地减小写入放大和读写波动，并且能够完全消除硬件预留的空间，从而提高了 SSD 的性能利用率。Zones 的地址空间和写逻辑如图 10 所示。

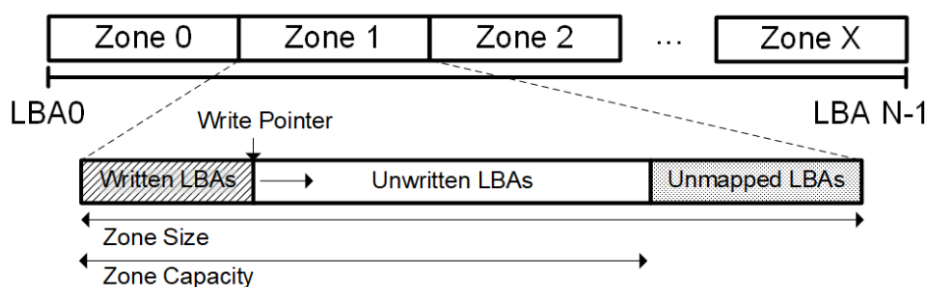


图 10: 单个 Zone 内的写逻辑

硬件方面的尝试其实不止 ZNS 一种。在利用硬件进行 SSD 存储优化的赛道上，有过许多新奇的软硬件技术。Intel 的傲腾就是其中之一，它利用全新的 3D XPoint 相变存储介质，将 SSD 的读写性能提升到了一个新的高度。但是傲腾的缺点也很明显，那就

是成本高昂，而且只有 Intel 自家的 CPU 才能够完全发挥傲腾的性能。而 ZNS 和 Zoned Storage SSD 的优势在于，它们是 NVMe 的标准，任何厂商都可以按照标准来生产。而且 ZNS 和 Zoned Storage SSD 的实际生产成本并不高，硬件上主控的计算压力减少了，也释放了更多原来的预留空间，所以大量生产后成本会比现在的 SSD 还要低，因此在可以遇见的未来，ZNS SSD 还会更加普及。

除了傲腾，还有 Open-Channel SSD (OCSSD)^[11]。它的主要优化点在于，它将 SSD 的主控和垃圾回收的功能都放到了主机上，而 SSD 只负责读写数据。这样做的好处是，主机可以根据自己的应用场景来优化 SSD 的性能，而且 SSD 的成本也会降低。但是这样做的缺点也很明显，那就是主机的计算压力会变大，而且主机的应用场景也会受到限制。它与 Zoned Storage SSD 的区别主要在于，OCSSD 是将 FTL 放置在主机端，从而减小 SSD 上主控的 DRAM 成本等，而 Zoned Storage SSD 则干脆基本放弃了细粒度的 FTL，只在非常大的 Zone 粒度上做了大块逻辑映射和磨损均衡，从而进一步地让 SSD 服务于计算和存储本身。

在硬件实现上，Zoned Storage SSD 可以设置 Zone 的类型。Zone 可以分为两类：Seq Zone 和 Conv Zone。Seq Zone 正如之前介绍的一样，可以随机读取，但是只能从指定的写指针位置写入数据。而 Conv Zone 则基本可以看作是普通的 SSD，主控仍然会维护这部分地址的 FTL，所以这部分 Zones 可以像传统 SSD 一般随机读写。这样分类 Zones，更多地是为了兼容现有的应用场景，因为现有的应用场景中，随机读写的需求还是非常多的。而 Seq Zone 则是为了更好地适应新的应用场景，因为新的应用场景中，随机读写的需求并不多，而且 Seq Zone 的性能也更好。

软件方面，ZenFS 则是利用 RocksDB 的文件系统接口将文件放置到原始分区块设备上的区域中。

RocksDB 是 Facebook 开源的一个 KV 存储引擎，它是 LevelDB 的一个分支，主要用于存储 Facebook 的消息系统的元数据。RocksDB 的特点是，它是一个基于 LSM-Tree 的存储引擎，它的读写性能都非常高，而且它的写入放大也非常低。RocksDB 的写入放大主要是通过将写入的数据放到内存中的 MemTable 中，然后再将 MemTable 中的数据写入到磁盘中的 SSTable 中来实现的。RocksDB 的读写性能和写入放大都非常优秀，所以它在 Facebook 的消息系统中得到了广泛的应用。

RocksDB 的 LST-Tree 有多个层次，每个层次都有自己的 SSTable。在 RocksDB 中，每个 SSTable 都是一个文件，而每个 SSTable 中的数据都是有序的。在 RocksDB 中，每个 SSTable 都有一个大小的限制，当 SSTable 的大小达到限制时，就会将达到大小的 SSTable 进行 Compact 操作。LSM-Tree 的操作逻辑（图 11），让这种数据结构在磁盘上的表现为一种类似于日志的结构，这样结构的数据库在磁盘上的随机写性能非常高，而且写入放大也非常低。

在 ZenFS 中，每个上述提到的 SSTable 就是一个 .sst 文件。ZenFS 对 SST 文件的

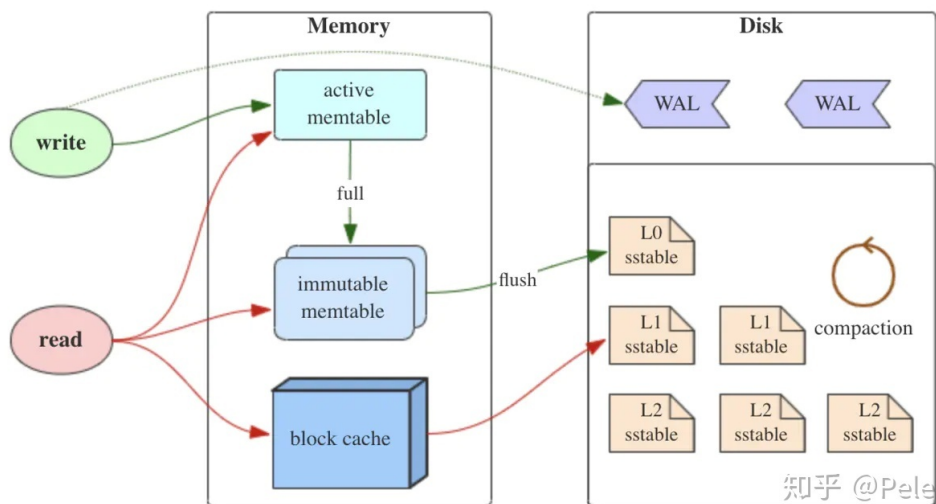


图 11: LSM-Tree 的操作逻辑

管理，也如 SSTable 的完全追加写入如出一辙，即使用 WAL（Write Ahead Log）管理文件的 MetaData 和文件数据，甚至文件的重命名、修改、新增和删除，都是只追加写入的。

2.3.2 ZenFS 源码分析

在调研过程中，我们对 ZenFS 的源代码进行了一次完整的分析，总结出我们理解中的 ZenFS 的系统框图（图 12）。

Zone

```

1  class Zone {
2      ZonedBlockDevice *zbd_;
3      ZonedBlockDeviceBackend *zbd_be_;
4      std::atomic_bool busy_;
5
6      uint64_t start_;//起始物理地址
7      uint64_t capacity_ /* remaining capacity */
8      uint64_t max_capacity_;
9      uint64_t wp_;
10     Env::WriteLifeTimeHint lifetime_;
11     std::atomic<uint64_t> used_capacity_;
12 }

```

这里的 Zone 对应的就是设备上的 Zone，不过在 ZenFS 所管理的 Zones 中，只支持 Seq Zone，所以这里的 Zone 也只会是 Seq Zone。

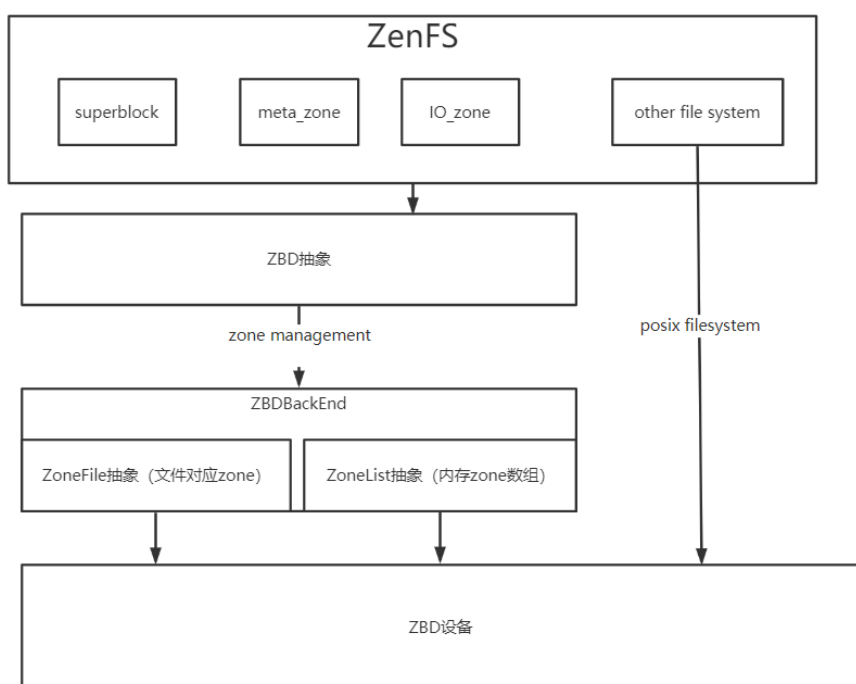


图 12: ZenFS 系统框图

一个 Zone 需要有所属块设备，起始地址和写指针，生命周期和空间相关的维护。Zone 中的方法有重置 Zone，判断是否为空或者是否写满，往写指针后面添加数据等等。

ZoneList

```

1  class ZoneList {
2      private:
3          void *data_;
4          unsigned int zone_count_;
5  }
```

由于一个 Zoned Storage SSD 上的 Zone 一般比较大，Zone 的数量并不多，所以 ZenFS 要求数据后端要能够快速提供一个指向 Zones 信息列表的指针，然后通过这个指针就可以快速地求取 Zones 的信息，例如某个 Zone 的起始地址、容量、写指针、是否下线、是否显式打开等。

ZonedBlockDeviceBackend

ZonedBlockDeviceBackend 是用来和底层设备交互需要经过的接口，这个接口将底层的 ZoneList 全部抽象成内存中连续的 ZoneList 并进行操作。

在 zbdlib_zenfs.h 以及 zbdlib_zenfs.cc 实现接口以方便上层对底层 zbd 的访问。

在 zonefs_zenfs.h 以及 zonefs_zenfs.cc 的实现是将上层对 zbd 的访问通过 posix

的读写文件方式给包装起来，从而能够访问到被 ZoneFS 包装过后的实际的 Zone。

```
1  class ZonedBlockDeviceBackend {
2      public:
3          uint32_t block_sz_ = 0;
4          uint64_t zone_sz_ = 0;
5          uint32_t nr_zones_ = 0;
6      }
```

backend 有两种类型分别是：

```
1  enum class BackendType {
2      kBlockDev,
3      kZoneFS,
4  };
```

当然，在我们实现了 RAID 之后，这里添加了 backend 类型 kRAID，用来表示这个 backend 是一个被 RAID 后的数据后端。

ZonedBlockDevice

```
1  class ZonedBlockDevice {
2      private:
3          std::unique_ptr<ZonedBlockDeviceBackend> zbd_be_;
4          std::vector<Zone *> io_zones; //用于io的zone
5          std::vector<Zone *> meta_zones; //用于保存元信息的zone
6          time_t start_time_;
7          std::shared_ptr<Logger> logger_;
8          uint32_t finish_threshold_ = 0;
9          std::atomic<uint64_t> bytes_written_{0};
10         std::atomic<uint64_t> gc_bytes_written_{0}; // 垃圾回收转移的字节数目
11
12         std::atomic<long> active_io_zones_; //分配io_zone时需要用锁
13         std::atomic<long> open_io_zones_;
14         /* Protects zone_resuorces_ condition variable, used
15            for notifying changes in open_io_zones_ */
16         std::mutex zone_resources_mtx_;
17         std::condition_variable zone_resources_;
18         std::mutex zone_deferred_status_mutex_;
19         IOStatus zone_deferred_status_;
20
21         std::condition_variable migrate_resource_;
22         std::mutex migrate_zone_mtx_;
```

```

23     std::atomic<bool> migrating_{false};
24
25     unsigned int max_nr_active_io_zones_;
26     unsigned int max_nr_open_io_zones_;
27
28     std::shared_ptr<ZenFSMetrics> metrics_;//todo
29 }

```

Open 函数

用来打开一个 ZonedBlockDevice，流程如下：

- 首先获取最大活跃 io_zones 数目和最大可打开 io_zones，这里保留了几个 zone 用于 metadata 存储，一个额外的 Zone 用于 extent migration，即 Extent 合并时使用的辅助空间。
- 从 backend 中获取已有 zones 分配 3 个 meta_zones。将剩余的非离线状态的可获得的 zone 放入 io_zones 中。遍历的过程可统计当前活跃 zone 的数目。

处于离线状态的块，也就是 offline，指的是这些块已经在硬件层面检测到错误，不可用了，需要被替换掉。

Get*Space 函数：顾名思义，用来获取诸如已经使用空间，空闲空间以及可回收空间大小

Log* 函数：用来输出日志，获取 zone 的整体使用情况，每个 zone 单个使用情况以及垃圾空间占比等。

分配 zone 区域方法

ZonedBlockDevice 重要的功能应该是操作 zone，其分配释放相关操作：

- ApplyFinishThreshold 将剩余空间小于预设的块给 finish 掉，finish 操作指的是把一个 zone 的写指针移动到 zone 末尾，剩余容量减为 0。
- FinishCheapestIOZone 将一个最小剩余空间的 zone 给 finish。
- GetBestOpenZoneMatch 将一个当前的文件生命周期和每一个 io_zone 进行生命周期对比，分配一个生命周期大于当前文件的生命周期且最接近的 zone。
- AllocateEmptyZone，顾名思义是分配一块空的 zone。
- ReleaseMigrateZone，顾名思义是释放 migrate_zone (todo migrate zone 是用来干嘛的)。
- TakeMigrateZone，选择一块最优 zone 当作 migrate_zone。

- AllocateIOZone，在保持 max_active_io_zones 的数量的前提下分配一块 zone。
- Read，从偏移 offset 处读 n 个 byte 吧应该是，这里边用了 ZonedDeviceBackend 提供的 Read 接口。

而 ZonedBlockDevice 的读写数据，则基本直接调用 ZonedBlockDeviceBackend 的相关接口，即交给下一层处理。

Snapshot

ZenFS 的 Snapshot 功能，有多种存储结构的 Snapshot，其记录下某一个确定时间的各个结构的基本信息，从而实现 WAL 结构下的数据一致性保持和快照恢复功能。

几种 Snapshot 的存储结构如下：

```

1 // ZBD设备快照记录设备的空闲空间 (free_space)
2 // 已使用空间 (used_space) 和可回收空间 (reclaimable_space) 。
3 class ZBDSnapshot {
4     public:
5         uint64_t free_space; //空闲空间
6         uint64_t used_space; //已使用空间
7         uint64_t reclaimable_space; //可回收空间
8     }
9
10 // Zone快照记录了Zone区域的开始地址 (start)，写指针 (wp)，以及容量相关信息。
11 class ZoneSnapshot {
12     public:
13         uint64_t start; //Zone开始地址
14         uint64_t wp; //写指针
15
16         uint64_t capacity; //容量
17         uint64_t used_capacity; //已用容量
18         uint64_t max_capacity; //最大容量
19     }
20 class ZoneExtentSnapshot {
21     public:
22         uint64_t start; //todo
23         uint64_t length; //Extent长度
24         uint64_t zone_start; //todo
25         std::string filename; //Extent所属文件名
26     }
27 // 论文中提到过一个文件拥有多个extent，extent不会跨Zone存储
28 // 在这些数据结构中可以看出。
29 class ZoneFileSnapshot {

```

```

30     public:
31         uint64_t file_id; // 顾名思义 file 的 id
32         std::string filename; // 文件名
33         std::vector<ZoneExtentSnapshot> extents; // 文件的 extents 快照集合
34     }
35     // 这里的快照便是之前所有快照的组合，记录了整个 ZenFS 设备的重要状态。
36     class ZenFSSnapshot {
37     public:
38         ZBDSnapshot zbd_;
39         std::vector<ZoneSnapshot> zones_;
40         std::vector<ZoneFileSnapshot> zone_files_;
41         std::vector<ZoneExtentSnapshot> extents_;
42     };

```

ZoneExtent

ZoneFs 的文件内容是由多个 extent 组成的，每个 extent 都放在一个 zone 里面，extent 不能跨 zone 存储。

```

1     class ZoneExtent {
2     public:
3         uint64_t start_; // 物理起始地址
4         uint64_t length_; // extent 的长度
5         Zone* zone_; // 所属 zone 指针
6
7         explicit ZoneExtent(uint64_t start, uint64_t length, Zone* zone);
8         Status DecodeFrom(Slice* input);
9         void EncodeTo(std::string* output);
10        void EncodeJson(std::ostream& json_stream);
11    };

```

ZoneFile

```

1     class ZoneFile {
2     private:
3         const uint64_t NO_EXTENT = 0xffffffffffffffff;
4
5         ZonedBlockDevice* zbd_;
6
7         std::vector<ZoneExtent*> extents_;
8         std::vector<std::string> linkfiles_;
9

```

```

10     Zone* active_zone_;
11     uint64_t extent_start_ = NO_EXTENT;
12     uint64_t extent_filepos_ = 0;
13
14     Env::WriteLifeTimeHint lifetime_;
15     IOType io_type_; /* Only used when writing */
16     uint64_t file_size_;
17     uint64_t file_id_;
18
19     uint32_t nr_synced_extents_ = 0;
20     bool open_for_wr_ = false;
21     std::mutex open_for_wr_mtx_;
22
23     time_t m_time_;
24     bool is_sparse_ = false;
25     bool is_deleted_ = false;
26
27     MetadataWriter* metadata_writer_ = NULL;
28
29     std::mutex writer_mtx_; // zonefs的读写锁
30     std::atomic<int> readers_{0};
31
32 public:
33     static const int SPARSE_HEADER_SIZE = 8;
34 }

```

ZoneFile 是 ZenFS 中的文件抽象，其记录了文件的元数据信息，包括文件名和路径、时间、文件 id、文件大小、文件类型、文件的 extents 等。

zonefile 在操作时只会操作一个 active_zone，同时文件记录了生命周期，读写锁等等。

其中实现的方法有稀疏文件 append，所谓稀疏文件就是除了在 zone 中保存了有效数据信息之外还保存了有效数据的长度信息，如图 13 所示。

sparse 方式的 zone 中会将长度记录计入，而普通的 zone 只会一直 append 文件中的 extent。sparse 文件能够更加灵活地管理文件的空间，但是会比非 sparse 文件多一个长度字段信息。

除此之外，zonefs 还有实现了 RocksDB 的 FSWritableFile 接口以支持顺序写的方法，实现了 FSSequentialFile 接口以实现顺序读，FSRandomAccessFile 接口以实现随机读。

SuperBlock

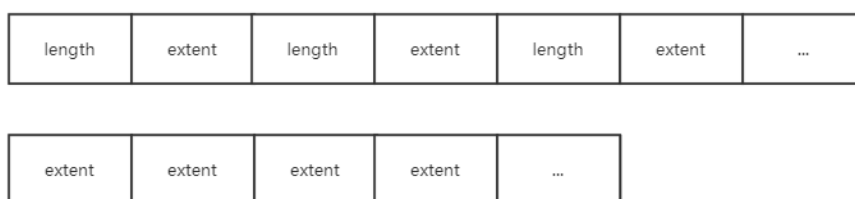


图 13: ZoneFile 稀疏文件

```

1  class Superblock {
2      uint32_t magic_ = 0;
3      char uuid_[37] = {0};
4      uint32_t sequence_ = 0;
5      uint32_t superblock_version_ = 0;
6      uint32_t flags_ = 0;
7      uint32_t block_size_ = 0; /* in bytes */
8      uint32_t zone_size_ = 0; /* in blocks */
9      uint32_t nr_zones_ = 0;
10     char aux_fs_path_[256] = {0};
11     uint32_t finish_treshold_ = 0;
12     char zenfs_version_[64]{0};
13     char reserved_[123] = {0};
14 }

```

ZenFS 的 SuperBlock 超级块相比其他文件系统的超级块要简单很多，其主要记录了一些基本信息，包括魔数、文件系统的版本、block 的大小、zone 的大小、zone 的数量、辅助文件系统路径，之后大部分的信息都是保留字段。

ZenFS 的靠近实际硬件的设计使得它更加简洁高效，但是它的预留修改空间也留足了，为我们今后的修改优化提供了便利。我们全盘 RAID 的实现就是利用了它的超级块的预留空间。

ZenMetaLog

```

1  class ZenMetaLog {
2      uint64_t read_pos_;
3      Zone* zone_;
4      ZonedBlockDevice* zbd_;
5      size_t bs_;
6  }

```

ZenMetaLog 是 ZenFS 的元数据日志格式，是 ZenFS 非常重要的数据存储结构。

zenmetalog 加入的 record 是如图14形式，首先存储的是由长度和数据形成的冗余码，然后是数据长度，再是实际的数据。



图 14: ZenMetaLog Record

每一次 roll 的操作都会重新开一个 zone 来记录 metadata 信息，时机是当这个 zone 写满了 todo，而一个新的 metazone 的内容首先会加入下面内容，此时记录的是文件系统的瞬时状态也即 snapshot，当然这些 snapshot 是用 record 的形式加入的（图15）。

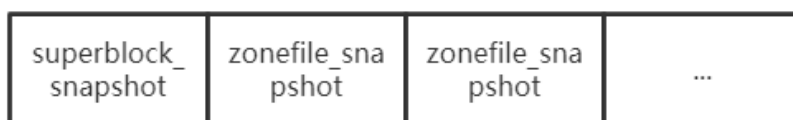


图 15: ZenMetaLog Roll

在 update, replace 或是 delete 文件时，都会按照如图16格式包装成 record 加入 metalog。

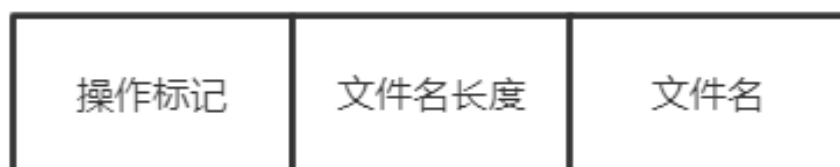


图 16: ZenMetaLog 格式

ZenFS

```

1  class ZenFS : public FileSystemWrapper {
2      ZonedBlockDevice* zbd_;
3      std::map<std::string, std::shared_ptr<ZoneFile>> files_;// ZoneFile
4      std::mutex files_mtx_;
5      std::shared_ptr<Logger> logger_;
6      std::atomic<uint64_t> next_file_id_;

```

```

7
8     Zone* cur_meta_zone_ = nullptr;
9     std::unique_ptr<ZenMetaLog> meta_log_;// 元信息
10    std::mutex metadata_sync_mtx_;
11    std::unique_ptr<Superblock> superblock_;// 超级块
12
13    std::shared_ptr<Logger> GetLogger() { return logger_; }
14
15    std::unique_ptr<std::thread> gc_worker_ = nullptr;// 垃圾回收
16    bool run_gc_worker_ = false;
17 }

```

GC_WORKER 垃圾回收

当全局 zone 里边的 free 空间小于一定比例之后，gc_worker 便开始了垃圾回收工作，核心思想便是：

收集需要垃圾回收的 zone，即 zone 的剩余空间满足一定的条件就进行回收，之后将 zone 中对应的 extent 移动到与当前 zonefile 生命周期相匹配的 zone，这个 zone 是顺序写的，这个移动 extent 的操作叫做 migrate 操作。

ZenFS 内判断当前 zone 是否需要回收的条件还是比较简单的，是一个简单的容量判断。当文件系统 GC 时，找到超过容量的 Zone，再找到一个使用较少的 Zone，将超过容量的 Zone 中的数据迁移到使用较少的 Zone 中，就完成了一次 GC。这里 ZenFS 简单的实现为我们之后的优化提供了便利。

MOUNT 逻辑：

- 读入所有 metazone 并且读出 superblock，选择 seq 序号最大的 superblock 的 meta 作为恢复 zone。
- 若是 readonly 的，则从磁盘同步一次数据。
- 若是可写的，并且用一个新的 metazone 记录当前文件系统的瞬时状态（superblock 以及各个 zonefile 的编码）。同时要将系统的未用 zone 重置一下。最后开启垃圾回收线程。

MKFS 逻辑：

- 选择一个 metazone 作为 log 记录的 zone。
- 写入 superblock 和各个 zonfile 编码到 metazone 中。

以上基本基于源码结构对 ZenFS 做了一个简单的分析。有了以上的理解分析，我们就可以开始对 ZenFS 进行修改优化了。

3 系统设计

3.1 AquaFS 整体架构

AquaFS 是一个模块化的文件系统，以下是 AquaFS 的整体架构图：

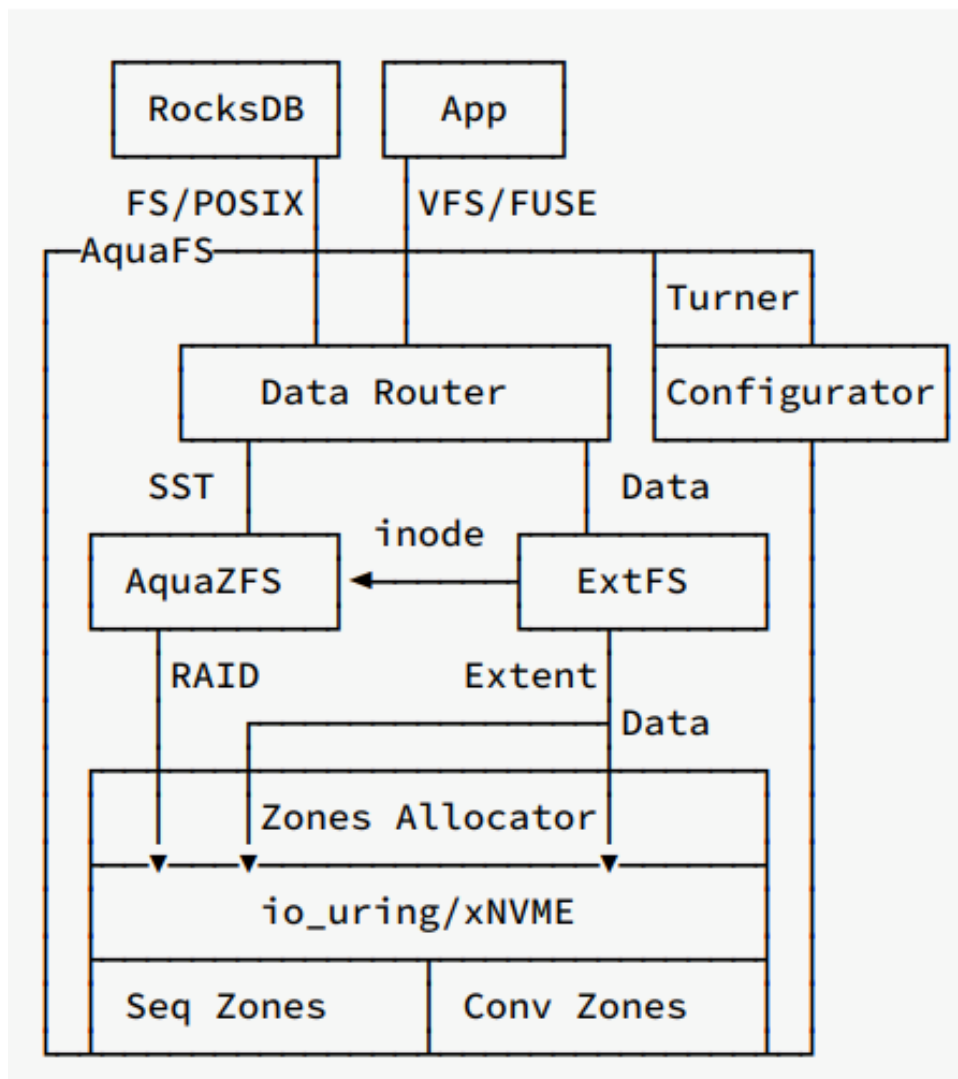


图 17: AquaFS 整体架构图

架构图 17 中的模块简略说明如下：

1. App：文件系统请求负载
2. RocksDB：数据库请求
3. Data Router：FileSystem 请求路由器，需要判断当前请求是否适合 WAL 优化
4. Turner：动态调整运行过程中的参数

5. Configurator：静态调整文件系统参数，建立文件系统时给出建议参数
6. AquaZFS：经过修改和优化的 ZenFS，支持 RAID 等功能
7. ExtFS：有 inode 系统的运行于 Seq/Conv Zones 上的通用文件系统
 - (a) 对普通请求，直接使用 Conv Zones
 - (b) 对 AquaZFS 的新文件，提供 inode 索引等读写优化
 - (c) 对较大的冷数据文件，分配到 Seq Zones
 - (d) 将一些可以异地更新的数据以 AquaFS::Extent 形式写入 AquaZFS
8. Zones Allocator：为 AquaZFS、ExtFS 提供 Zone 分配服务、同时具备磨损均衡的功能
9. Zones：
 - (a) Seq Zones：只能顺序写的 Zones
 - (b) Conv Zones：可以随机写的 Zones
10. AquaFS：整体文件系统

3.2 模块设计细节

RocksDB 使用 AquaFS

RocksDB 使用 AquaFS，可以走两种数据通路：FileSystem 和 POSIX 接口。

RocksDB 使用 FileSystem 接口使用 AquaFS

将 AquaFS 编译为 RocksDB 插件，Data Router 使用 FileSystem 接口。

Data Router 主要转发 SST 请求到 AquaZFS，其他请求转发到 ExtFS，并对特殊情况做二者的负载均衡。

RocksDB 使用 POSIX 接口使用 AquaFS

AquaFS 的 Data Router 通过 Kernel Module 或 FUSE 提供 POSIX 访问接口，并智能判断数据负载位置。

App 使用 AquaFS

由于假定 App 并未实现 FileSystem 接口，所以 App 数据可以通过 Kernel Module / FUSE 方式经过 Data Router 到下层。

调参模块

AquaFS 中调参模块主要有两个部分：Configurator、Turner。

Configurator 在文件系统创建前评估当前系统更适合的固定参数，并结合需求给出合适的参数选择和预估的性能区间。

Turner 在文件系统使用过程中保持运行，根据系统当前状态动态调整可改变的参数，以获得更加灵活良好的整体表现。

可调参数

1. 固定参数
 - (a) 块大小
 - (b) 固定 RAID 参数
 - (c) 数据后端类型
2. GC
 - (a) GC 容量阈值
 - (b) GC 间隔时间
3. 动态 RAID
 - (a) 分配时间 (GC)
 - (b) 分配参数 (0/1/5...)
4. 文件请求分类
 - (a) 分类为 SST、普通数据
 - (b) 分类冷热文件/数据
5. IO 加速方式：io_uring/xNVME

AquaZFS 和 ExtFS

ExtFS 是主要运行于 Conv Zones 上的针对 ZNS 优化的文件系统。主要特性：

1. 必须原地更新的数据放在 Conv Zones 内，如 Superblock 等（待定）
2. 适合异地更新的数据通过 AquaZFS 保存在 Seq Zones 内，如 MetaData（待定）
3. 如果智能检测到 AquaZFS 内部分数据不适合 Seq Zones 存储，则转发到 ExtFS 内处理

4. 用冗余的 inode 等为 AquaZFS 提供索引，可以动态降低其内存消耗

AquaZFS 是基于 ZenFS 的优化修改，支持以上特性，在保持高性能的同时提升文件系统的灵活性。

RAID

在 AquaZFS 从写盘前到实际写盘之间，存在一层 RAID 逻辑。

在原版 ZenFS 的实现中并没有实现 RAID 的逻辑，其只能保证存储的“记录”的数据正确性，而无法保证“文件”的数据正确性，也不能在遇到磁盘故障的时候自动处理修复数据。

AquaZFS 在 ZenFS 的基础上实现了 RAID 逻辑，可以在磁盘故障时自动修复数据，且能够根据配置自动使用不同的 RAID 策略。

除了保证数据的正确性，AquaZFS 还可以根据不同的 RAID 策略提供不同的性能。当前实现的 RAID0 可以以 N 倍加速单线程数据的读写，RAID1 在提供 N 倍读性能的同时，还可以用简单的冗余策略保证数据的安全性。此处的 N 的大小取决于 ZNS 硬件设计上能够打开的最多的同时读写 Zone 数量的和。

AquaZFS 的 RAID 有两种基本模式：全盘模式和智能动态分区模式。

在全盘模式下，AquaZFS 会以 SSD 设备为单位应用同一种 RAID 策略，其策略配置写入超级块中，可以快速将已经在使用中的 AquaZFS 转换为全盘 RAID 格式，并使用超级块在 Meta Zones 中的追加写入逻辑保证配置的正确性。

在智能动态分区模式下，AquaZFS 以 Zones 为单位配置 RAID 策略，可以在不同的 Zones 中使用不同的 RAID 策略。RAID 策略信息以记录形式写入 Meta Zones 中，并使用 WAL 配合 Snapshot 保证数据的正确性。

在实现的过程中，我们为其他的 RAID 逻辑做了预留，能够快速地实现其他 RAID 逻辑。

1. 可灵活配置为：静态固定参数 RAID、智能动态分区 RAID
2. 可以在用户态驱动 NVME，或者内核态使用 liburing 进行 IO 加速，充分利用多盘优势提升性能
3. 利用 Turner 提供的建议，在 AquaZFS 垃圾回收时或合并 Extent 时调整 RAID 逻辑，使文件系统在安全性、性能上有更好的权衡点

Zones Allocator

为 AquaZFS 和 ExtFS 提供统一的 Zones 分配服务。

1. 让整盘空间得到更加充分的利用，减少由于分开两种子系统造成的空间碎片

2. 根据历史数据，测算不同 Zones 的寿命和速度，来控制 Zones 的分配逻辑，延长磁盘寿命，提高磁盘吞吐

IO 加速

在 AquaFS 向上提供 FileSystem 接口时，由于负载程序对 FileSystem 接口做了适配，所以可以让负载程序和整个 AquaFS 都跑在用户态。

当 AquaFS 整个运行在用户态，可以使用 xNVME 用户态 NVME 协议驱动，降低内核态用户态切换的性能损失，同时也可用 io_uring 加速。

若 AquaFS 使用 POSIX 接口，可以使用 VFS 或者 FUSE 接口，此时也可以用 xNVME 或者 io_uring 进行 IO 加速。

智能化

这个架构的「智能」体现在哪？

1. 相比与 ZenFS，灵活性更强
 - (a) 根据运行状况调整参数
 - (b) 提供 RAID 功能，并可以动态分配
2. 数据安全性更强：RAID 功能
3. 智能分配请求
 - (a) 进行读写请求分离
 - (b) 进行磨损均衡的 RAID 分配
4. 更加高效：RAID 功能、全异步 IO 加速

3.3 开发计划

由于一些原因，我们原来的队长没能继续参与到项目中来，所以我们的开发计划也有所改变。

在初赛阶段，考虑到我们的精力和人数，我们优先实现 AquaFS 文件系统的以下几个功能部分：

1. AquaFS 文件系统的 RAID0，RAID1，智能分配器实现
2. AquaFS 文件的智能调参模块初步实现
3. AquaFS 文件系统相关 RAID 测试，智能分配器测试，调参模块初步测试

在复赛阶段，我们将继续完善 AquaFS，完善的功能主要是下面几个模块：

1. AquaFS 文件系统的 RAID5 实现
2. AquaFS 文件系统 RAID0 的 io uring 加速实现
3. AquaFS 文件系统的智能调参模块完善
4. AquaFS 文件系统磨损均衡模块实现
5. AquaFS 文件系统 RAID5，磨损均衡模块测试

4 系统实现

针对以上设计，我们实现了 AquaFS 文件系统。AquaFS 文件系统的实现基于 ZenFS，我们在 ZenFS 的基础上进行了修改和优化，使其支持 RAID 等功能。

4.1 AquaFS 文件系统的 RAID 实现

RAID 是指独立冗余磁盘阵列 (Redundant Array of Independent Disks)，它是一种数据存储技术，通过将多个硬盘组合起来，实现数据的备份、容错和性能优化。

当我们确定 RAID 方案的时候，首先需要解决的问题是：如何将 ZNS 中的 Zones 组织成 RAID 单位。在传统的 RAID 实现中，RAID 单位是磁盘，而在 ZNS 中，RAID 单位可以是磁盘，也可以是 Zone。我们既可以将 ZNS 中的 Zones 组织成 RAID 单位，也可以将 ZNS 中的磁盘组织成 RAID 单位。我们将 ZNS 中的 Zones 组织成 RAID 单位的方式称为分区 RAID，将 ZNS 中的磁盘组织成 RAID 单位的方式称为全盘 RAID。

那么我们是否能将所有的 Zones 都组织成 RAID 单位呢？答案是否定的。在 Zoned Storage SSD 中，Zone 分为了两种：Seq Zone 和 Conv Zone。更详细的说明在 2.3.1。这两种 Zones 由于其读写方式的区别，是不能混为一谈的。如果将 Seq Zone 和 Conv Zone 混用，则浪费了主控上专门对 Conv Zones 配置的 DRAM Cache，也会降低 Seq Zone 的读写性能。所以，我们不能将所有的 Zones 都组织成 RAID 单位，而是需要将 Seq Zone 和 Conv Zone 分开组织成 RAID 单位。

除了 Zone 类型的问题，基于 Zone 的 RAID 对磁盘之间的参数也提出了要求。在相同参数的 Zoned Storage SSD 中，不同设备上的 Zone 由于 Zone 大小、块大小等是相同的，可以很方便地组合为 RAID 单位。但是，如果不同设备上的 Zone 参数不同，那么就不能直接组合为 RAID 单位。所以，我们首先实现的是将相同参数的 Zone 组合为 RAID 单位，对不同参数的 Zone 组合为 RAID 单位的实现将在复赛阶段进行探索，例如进行参数转换、分区兼容等尝试。

除此之外，在 ZNS 上实现 RAID 还需要解决 Seq Zones 上的 RAID 算法问题。RAID 0 与 RAID 1 的逻辑相对简单，而 RAID 5、RAID 6 等则需要在 Seq Zone 上实现更复杂的 RAID 算法。这些算法需要文件系统进一步实现各个 Zones 的读写缓存、原子同步写入等功能。我们尝试了 RAID 5 的算法，但是运行效果暂时不理想。在 ZNS 上实现 RAID 5、RAID 6 等算法的难度较大，我们决定在复赛阶段进行探索。

当前 AquaFS 的 RAID 实现主要分为两种：全盘 RAID 和分区 RAID。这两种 RAID 实现的区别在于，全盘 RAID 的 RAID 单位是磁盘，而分区 RAID 的 RAID 单位是 Zone。

4.1.1 全盘 RAID 的实现

在传统的 RAID 实现中，基本都是以磁盘为单位进行数据 RAID 逻辑。以磁盘为单位的 RAID 能够充分运用各个磁盘的数据吞吐，并基于 Linux 的块设备等抽象层提供软件上的 RAID 功能。我们也首先实现了全盘 RAID 的功能，能够将多个 ZNS 配置为 RAID 0、RAID 1 模式。

由于 ZenFS 是一个没有一般 POSIX 接口的文件系统，其仅向上服务于 RocksDB，所以我们并不能简单地使用 Linux 上常用的磁盘软件 RAID 来实现 ZNS 的 RAID 功能。更何况，ZNS 并不是 Linux 兼容的块设备，内核中并没有对 ZNS 的 RAID 支持。经过调研和评估，我们认为在 Linux Kernel 内实现对 ZNS 的 RAID 支持并不现实。于是，我们需要在 ZenFS 的基础上实现 RAID 功能。

ZenFS 的数据读写将会经过以下几个层次：

1. RocksDB 的 SSTable
2. RocksDB 的 MemTable
3. RocksDB 的 WAL
4. RocksDB 的 FileSystem
5. ZenFS 的 ZoneFile
6. ZenFS 的 ZoneExtent
7. ZenFS 的 Record
8. ZenFS 的 ZonedBlockDeviceBackend
9. libzbd 的 ZbdlbBackend
10. Linux Kernel 相关系统调用

其中，RocksDB 的 SSTable、MemTable、WAL、FileSystem 都是 RocksDB 的内部实现，ZenFS 的 ZoneFile、ZoneExtent、Record、ZonedBlockDeviceBackend 都是 ZenFS 的内部实现，libzbd 的 ZbdlbBackend 是 ZenFS 内对 libzbd 的接口适配，Linux Kernel 相关系统调用是 libzbd 作为用户态程序调用 Linux Kernel 内的设备驱动程序的接口，最终还是通过 Linux 的系统调用来实现数据的传输。

ZenFS 和 RocksDB 还支持了另一种数据读写方式，通过 ZoneFS 将 ZNS 中的 Zones 以文件映射到文件系统中，然后通过文件系统的接口来读写数据。这种方式的数据读写流程如下：

1. RocksDB 的 SSTable
2. RocksDB 的 MemTable
3. RocksDB 的 WAL
4. RocksDB 的 FileSystem
5. ZenFS 的 ZoneFile
6. ZenFS 的 ZoneExtent
7. ZenFS 的 Record
8. ZenFS 的 ZonedBlockDeviceBackend
9. ZoneFS 的 ZoneFSBackend
10. Linux Kernel VFS 接口

虽然这种方式通过 ZoneFS 将 ZNS 中的 Zones 以文件映射到文件系统中，但是 ZoneFS 并不是一个通用的文件系统，它仅仅是一个将 ZNS 中的 Zones 以文件的形式映射到文件系统中的一个文件系统。ZoneFS 并不支持文件的创建、删除、重命名等操作，文件的读写操作也支持不完全，而且还会引入更多的存储 IO 栈，所以我们并没有选择通过 ZoneFS 来实现 ZNS 的 RAID 功能。

我们选择在 ZenFS 的 ZonedBlockDeviceBackend 层实现 ZNS 的 RAID 功能。ZonedBlockDeviceBackend 层是 ZenFS 中管理数据后端的层，它是 ZenFS 与 libzbd 或 ZoneFS 之间的接口层，负责将 ZenFS 的数据读写请求转换为 libzbd 或 ZoneFS 的数据读写请求。

我们通过继承 ZonedBlockDeviceBackend 来实现 ZenFS 内的数据 RAID 功能。ZonedBlockDeviceBackend 的继承类图如图 18 所示。

AbstractRaidZonedBlockDevice 下的 Raid0ZonedBlockDevice、Raid1ZonedBlockDevice 和 RaidCZonedBlockDevice 即为全盘 RAID 的实现。其中，Raid0ZonedBlockDevice 实现

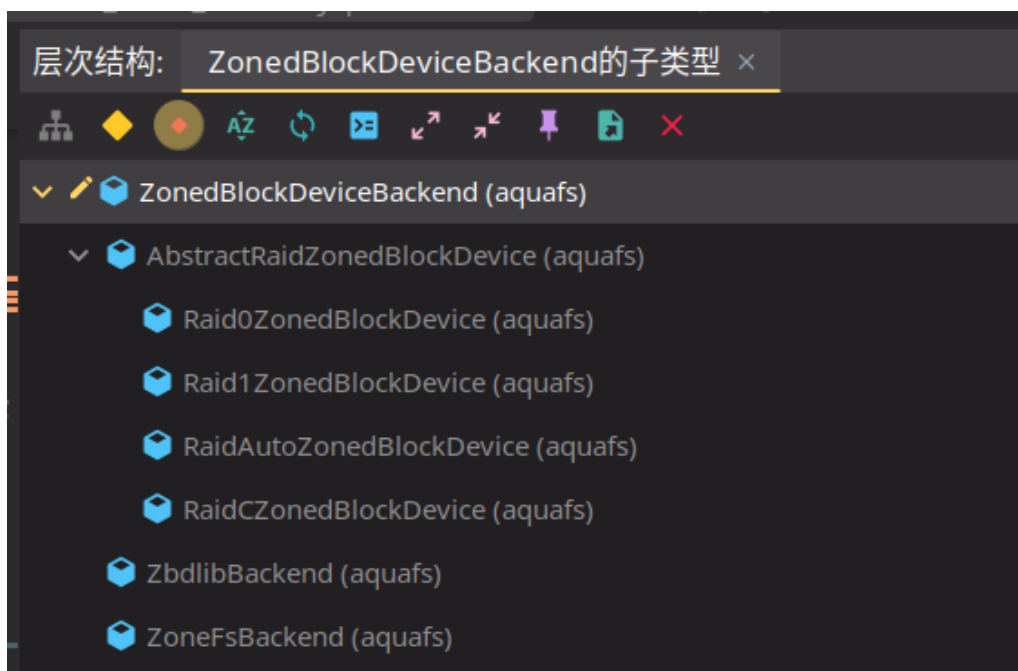


图 18: ZonedBlockDeviceBackend 继承类图

了 RAID 0 的功能，Raid1ZonedBlockDevice 实现了 RAID 1 的功能，RaidCZonedBlockDevice 实现了 RAID C 的功能。

RAID C 是我们自定义的一种简单 RAID 格式，它通过 Zones 的合并来实现简单的数据拼合逻辑，即将多个 Zone 合并为一个 Zone，然后将数据写入到合并后的 Zone 中。RAID C 的实现如图 19 所示。

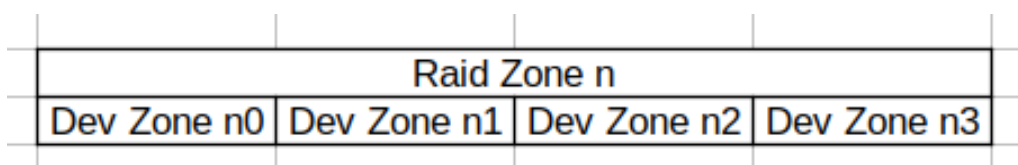


图 19: 全盘 RAID 的数据排布

此时多个 (m 个) 设备上实际存在的 Zones 合并为一个大的逻辑 Raid Zone，数据将会按顺序依次写入 Dev Zone n_i ($0 \leq i < m$)。此 RAID C 模式存在的意义是，形成一个简单的 Zone 合并逻辑，便于后续开发应用。其中，Dev Zone n_i 分别为来自第 i 个设备的第 n 个 Zone。

RAID 0 和 RAID 1 的实现与传统的 RAID 0 和 RAID 1 的实现类似，RAID 0 将数据分散写入多个设备中，RAID 1 将数据写入多个设备中的一个。RAID 0 和 RAID 1 的实现中的数据排布与 RAID C (图 19) 基本一致。

传统块设备上 RAID 0 的工作原理如下：^[12]

1. 数据被分割成固定大小的块。

2. 这些块按照一定的规则，如轮流或按块交替，分配到不同的物理磁盘驱动器上。
3. 每个磁盘驱动器只存储一部分数据，因此所有磁盘驱动器都可以同时读写数据，从而提高了读写速度。
4. 当需要读取数据时，RAID 控制器会从所有驱动器中读取数据块，然后将它们组合成完整的数据块，并将其发送给请求数据的主机。

RAID 0 在写入时，将会将数据分散写入多个设备中。类似于传统块设备的 Block Size，ZNS 也是有最小写入单位的，也是 Block Size。RAID 0 在读写时，可以将读写请求分割为不同的 Block Size 的读写请求，然后对这些请求重新合并排序，再调用底层的读写接口。

当前我们的实现中，RAID 0 的实现基本与传统 RAID 0 一致。区分的点有以下几个：

1. 数据需要以 Zone 为单位做组织，而不是以设备为单位。即与传统 RAID 0 的最后向上层应用提供一个抽象磁盘设备不同，我们需要向上层应用提供多个抽象的 Zones。
2. 数据被分为不同的 Block Size 的块，分得尽量小。
3. 当需要读写数据时，RAID 0 会将读请求分割为不同的 Block Size 的读写请求，然后对这些请求重新合并排序，再调用底层的读接口。
4. 采用了 Direct IO 加速方案，即绕过 Page Cache，直接读写设备。
5. 采用了 io_uring 加速方案，即使用 io_uring 作为异步 IO 框架，提高 IO 吞吐量。

RAID0 的数据排列逻辑如下图所示：

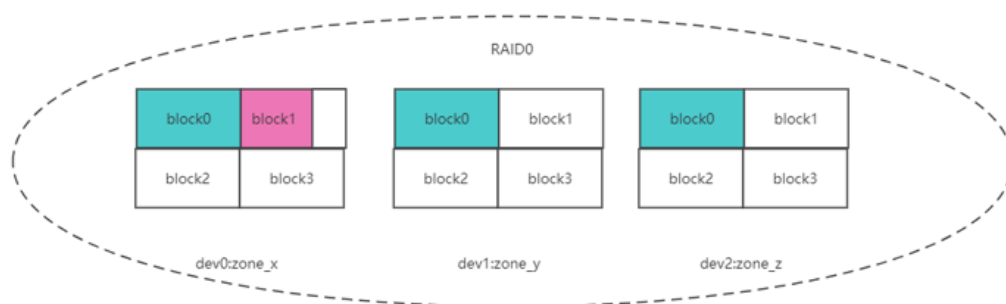


图 20: RAID0 数据排列逻辑

以 RAID 0 读为例，其未经 IO 优化的读流程如下所示：

```

1  int Raid0ZonedBlockDevice::Read(char *buf, int size, uint64_t pos,
2                                     bool direct) {
3  #ifndef AQUAFS_RAID_URING
4      // split read range as blocks
5      int sz_read = 0;
6      int r;
7      while (size > 0) {
8          auto req_size =
9              std::min(size, static_cast<int>(GetBlockSize() - pos % GetBlockSize()));
10         r = devices_[get_idx_dev(pos)]->Read(buf, req_size, req_pos(pos), direct);
11         if (r > 0) {
12             size -= r;
13             sz_read += r;
14             buf += r;
15             pos += r;
16         } else {
17             return r;
18         }
19     }
20     return sz_read;
21 #else
22     // ...
23 #endif
24 }

```

代码逻辑主要为，每次请求最多读取一个 Block Size 的数据，然后将读取的数据拼接到 buf 中，直到读取完毕。其中需要多次重复计算数据分块的设备位置和块位置，于是这里使用了 get_idx_dev 和 req_pos 函数来快速计算设备位置和块位置。这些函数被实现在 AbstractRaidZonedBlockDevice 层，以便其子类可以快速调用其逻辑。

在上述代码中，我们将读请求分割为不同的 Block Size 的读请求，然后调用底层的读写接口。这样做的好处是，可以将读请求分散到多个设备中，从而提高读性能。不过，上述代码中其实并没有体现 RAID 0 的多设备读优化，还是单线程读取。我们在后文中实现了基于 uring 的多设备并行读优化。

除了读写逻辑，RAID 还需要抽象出 Zones 的统一管理。上层软件使用了 RAID 向上提供的抽象 RAID Zones，那么对这些抽象 RAID Zones 的各种 Zone 操作也应当起到作用。例如，对 RAID 0/1 Zone 的重置操作应当扩散到这个 RAID Zone 对应的所有 Dev Zones；对求一个 RAID Zone 的逻辑写指针值，也应该按照 RAID 逻辑计算逻辑 RAID Zone 内的容量，然后计算已用容量，最后计算出逻辑写指针值。对 Zones 的 Close、Finish

等操作的逻辑类似。

当使用全盘 RAID 1 的时候，还有个可以优化的点，即当组成 RAID 的设备的参数相同时，由于这些设备的 Zone 大小相同，因此可以将这些设备的 Zone 一一对应起来，从而利用写指针的对应关系，不需要对请求进行切分，减轻了一些计算上的开销。

4.1.2 智能分区 RAID 的实现

分区 RAID 的实现与全盘 RAID 的实现类似，但是我们在逻辑 Raid Zone 和实际设备 Zone 之间加上了一层基于 Zones 的映射。这些映射是通过 ZenFS 的 Record 写入 MetaZones 内的，将在每次文件系统加载的时候逐步读取加载映射逻辑。

加上这一层映射之后，分区 RAID 的数据排布逻辑可能如图 21 所示。

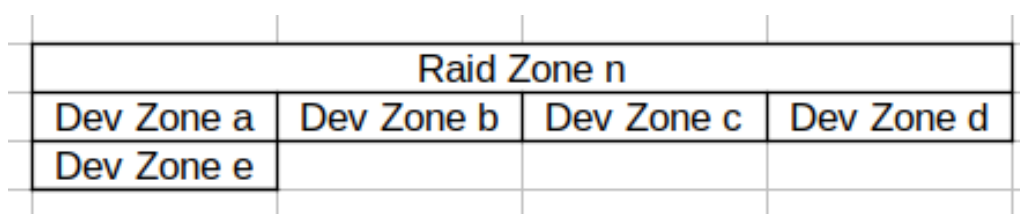


图 21: 分区 RAID 的数据排布

Zone Raid n 为向上层暴露出的可读写数据 Zone 区域，而其中可以存在多种不同的 RAID 逻辑或映射类型。

在示例图 21 中，Dev Zone x 表示来自不同或相同设备的设备上物理存在的 Zone。如果这个 Raid Zone 被配置为 RAID 1，则 Dev Zone a 和 Dev Zone e 将同时以 RAID 1 数据冗余方式为 Raid Zone n 的前四分之一数据提供服务，其他 Dev Zones 由于没有映射，将回退到 RAID C 逻辑提供数据存储服务。若这个 Raid Zone 被配置为 RAID 0，则 Dev Zone a 和 Dev Zone e 将同时以 RAID 0 数据分散方式为 Raid Zone n 的前四分之一数据提供服务，其他 Dev Zones 由于没有映射，将回退到 RAID C 逻辑提供数据存储服务。

实际代码实现上，除了上述映射逻辑处理，还有许多细节需要考虑。

物理块逻辑设备均分问题：在抽象 RAID Zone 里的 Zones 中，这些块可以来自不同的设备，也可以来自同一个设备。但是为了最大化利用 ZNS 的带宽，我们需要尽量将数据分散写入不同的设备中。

跨 Zones 读写问题：由于我们添加的映射逻辑，使得我们的数据排布不再是连续的，而是以 Zones 为单位分散的。这就导致了我们的读写请求可能会跨越多个 Zone。这就需要在读写时，需要将读写请求分割为不同的 Block Size 的读写请求，然后可能对这些请求重新合并排序，再调用底层的读写接口。

在读或写代码中，通过数据段分割并递归调用自身的方式实现跨 Zones 读写的请求分割：

```

1  if (static_cast<decltype(zone_sz_)>(size) > zone_sz_) {
2      // may cross raid zone, split read range as zones
3      int sz_read = 0;
4      int r;
5      while (size > 0) {
6          auto req_size =
7              std::min(size, static_cast<int>(zone_sz_ - pos % zone_sz_));
8          r = Read(buf, req_size, pos, direct);
9          if (r > 0) {
10             buf += r;
11             pos += r;
12             sz_read += r;
13             size -= r;
14         } else {
15             return r;
16         }
17     }
18     // flush_zone_info();
19     return sz_read;
20 } else {
21     // ...

```

数据后端问题：由于我们是基于 ZonedBlockDeviceBackend 来实现的，而 ZonedBlockDeviceBackend 有多种子类，可以是 libzbd、ZoneFS 甚至是原来实现的全盘 RAID。为了进一步简化 IO 调用栈，我们在实现分区 RAID 时，假定数据后端都是 libzbd 提供的读写。这在之后可以进一步优化，以支持更多的数据后端，提升 RAID 逻辑的灵活性，如添加 ZoneFS 支持、添加 SPDK 等 Kernel bypass 方案支持等。

对 ZenFS 的兼容性问题：ZenFS 在加载的过程中，会对固定的 MetaZones 进行扫描，通过 Magic Number 查找到 Meta Zones 中的可用的超级块，并选择最新的超级块进行文件系统初始化。为了兼容 ZenFS 的 Meta Data 管理逻辑，我们不能改变 MetaZones 的排布，也不能改变超级块的存储逻辑。因此，我们在实现分区 RAID 时，需要保证 MetaZones 的排布不变，超级块的存储逻辑不变，以及超级块的存储位置不变。所以，我们在实现分区 RAID 时，将 MetaZones 的排布和超级块的存储位置都固定在了第一个设备上，进行映射的连续逻辑预分配，这样就可以保证 ZenFS 的兼容性，使得 ZenFS 在加载分区 RAID 时，可以正常加载。

在创建或读取文件系统时的预分配映射：

```

1  // create temporal device map: AQUAFS_META_ZONES in the first device is used

```

```

2  // as meta zones, and marked as RAID_NONE; others are marked as RAID_C
3  for (idx_t idx = 0; idx < AQUAFS_META_ZONES; idx++) {
4      for (size_t i = 0; i < nr_dev(); i++)
5          allocator.addMapping(idx * nr_dev() + i, 0, idx * nr_dev() + i);
6      allocator.setMappingMode(idx, RaidMode::RAID_NONE);
7  }

```

为了管理分区之间的映射关系，我们通过组合的方式实现了一个分区分配器 ZoneRaidAllocator。其可以管理分区的映射关系，以及分区的 RAID 逻辑。其主要接口如下：

```

1  Status addMapping(idx_t logical_raid_zone_sub_idx, idx_t physical_device_idx,
2                  idx_t physical_zone_idx);
3  void setMappingMode(idx_t logical_raid_zone_idx, RaidModeItem mode);
4  void setMappingMode(idx_t logical_raid_zone_idx, RaidMode mode);
5
6  int getFreeDeviceZone(idx_t device);
7  int getFreeZoneDevice(idx_t device_zone);
8  Status createMapping(idx_t logical_raid_zone_idx);
9  Status createMappingTwice(idx_t logical_raid_zone_idx);
10 Status createOneMappingAt(idx_t logical_raid_zone_sub_idx, idx_t device,
11                          idx_t &zone);
12 void setOffline(idx_t device, idx_t zone);

```

其可以提供映射关系的查询、增加、删除、修改等功能，以及提供 Raid Zone 所分配的 RAID 逻辑的查询、增加、删除、修改等功能。

同时，它还提供了 setOffline 功能，可以在发现设备故障时，将故障设备的所有分区设置为 Offline 状态，以便后续的故障处理。

4.1.3 分区 RAID 故障处理

在分区 RAID 故障处理方面，我们实现了基于分区 RAID 的故障处理方案。其主要思路是，当发现设备故障时，将故障设备的所有分区设置为 Offline 状态，然后将所有的分区重新映射到其他设备上，以保证数据的可用性。

由于在调研中我们发现，Nand Flash 向上提供的数据一般含有 ECC 校验和纠错（见章节??），且 ZenFS 中的 Record 也有 CRC 校验和，所以我们认为在数据传输过程中，较少比特数据的完整性是可以保证的^[13]。所以，我们在实现分区 RAID 故障处理时，仅考虑大块数据的完整性恢复，而不考虑小块数据的完整性。

```

1  // 在数据操作函数中

```



```

2  if (r < 0) {
3      auto status = ScanAndHandleOffline();
4      if (status.ok()) {
5          // retry this read
6          return Read(buf, size, pos, direct);
7      } else {
8          Error(logger_, "failed to restore data: %s", status.getState());
9          return r;
10     }
11 }

```

ScanAndHandleOffline 函数将扫描当前所有盘的数据状态，如果发现有盘处于 Offline 状态，则将所有的分区重新映射到其他设备上，并根据 RAID 逻辑进行数据恢复。

当前由于仅实现了 RAID 0 和 RAID 1 的数据逻辑，所以暂时仅支持 RAID 1 分区的数据恢复。当发现 RAID 1 分区的数据不一致时，将重新选择一个 Dev Zone，将数据恢复到新的 Dev Zone 上。

当前的故障处理逻辑还相对比较简单，在复赛实现更多 RAID 逻辑后，将进一步改进此部分的故障处理逻辑。

在复赛中我们实现了 RAID5 的故障处理部分，RAID5 的逻辑架构如下图所示：

通过将不同设备的 zone 合并成一个大的 zone，再默认将最后一个设备的 zone 设置成校验 zone，可以实现简单的全盘 RAID5 逻辑，这样在一块 zone 出现错误后，可以利用一块 RAID5 zone 中的其余 zone 进行数据恢复。

4.2 AquaFS 的 IO 加速实现

在上文中，我们已经介绍了 AquaFS 的 RAID 方案，但是由于是比较初级的版本，很多代码并没有进行并行优化，仍然是单线程的。所以，我们在实现 IO 加速方面，主要是对 RAID 下的 IO 操作进行并行优化。

在我们实现之前，ZenFS 的 IO 优化主要是从 RocksDB 中继承而来。RocksDB 中可以使用 Direct IO^[14] 来读写文件，以避免数据在用户态和内核态之间的拷贝。其 Direct IO 优化既可以用于 RocksDB 的 PosixFileSystem 常规文件系统后端，也可以用于 RocksDB 的 ZenFS 文件系统后端。

Direct IO 主要的优化逻辑是，将用户态的数据缓冲区直接映射到内核态的页缓存中，从而避免了数据在用户态和内核态之间的拷贝。通过 Direct IO 打开的设备，从设备中获得的数据将直接写入在用户态的数据缓冲区中，而不是先写入内核态的页缓存中，从而减少了一次数据拷贝。

Direct IO 的使用也非常简单，只需要在打开文件时，将 O_DIRECT 标志传入即可。

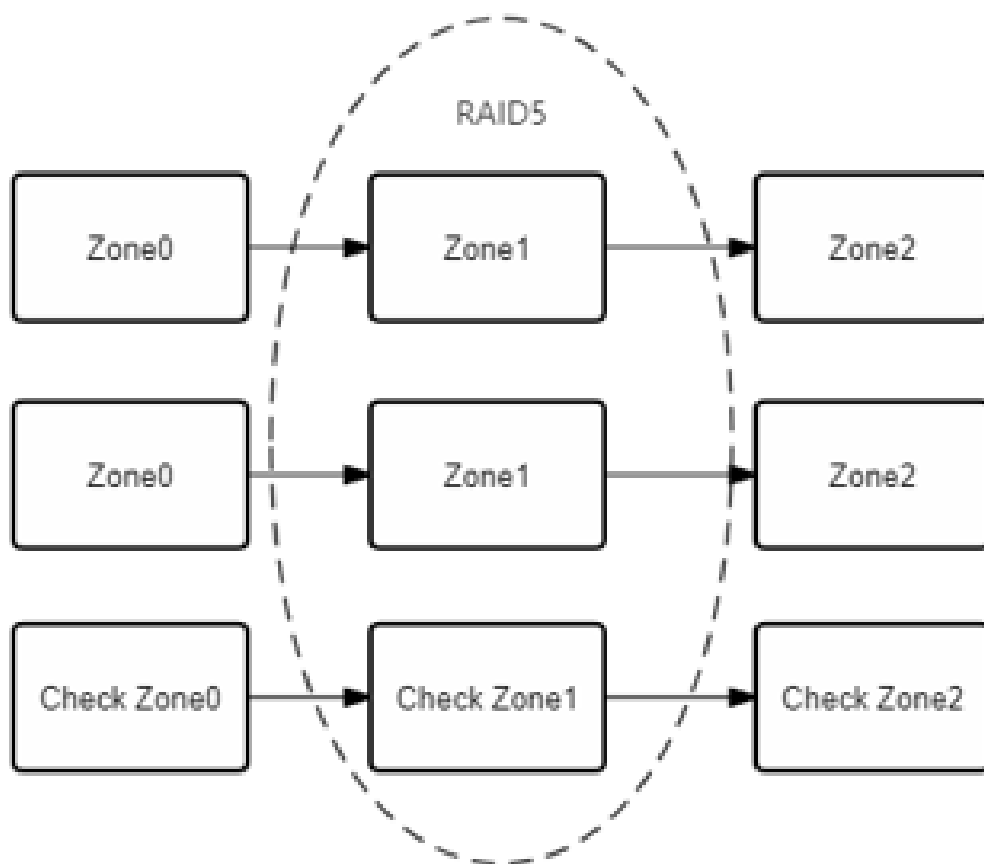


图 22: RAID5 实现方式

但是，由于 Direct IO 的使用需要满足一些条件，如文件的偏移量和长度需要是当前系统内存页大小的整数倍，所以在使用 Direct IO 时，需要对读写对应的内存缓冲区地址和长度进行对齐。具体对齐方式为使用 `posix_memalign` 函数来申请内存，控制参数将申请的内存地址对齐到当前系统内存页大小的整数倍。

```
ret = posix_memalign((void**)&buffer, sysconf(_SC_PAGESIZE), phys_sz);
```

我们发现，在我们的加速方案中，Direct IO 可以与更多的 IO 加速方案并存。如在我们的加速方案中，我们可以使用 `io_uring` 来加速 IO 操作，而 `io_uring` 也可以与 Direct IO 并存。

在加速方案选择上，我们选择了 `io_uring` 方案。当前 Linux 系统上有许多 IO 加速方案，如 AIO、`io_uring`、`libaio` 等，如图 23 所示。

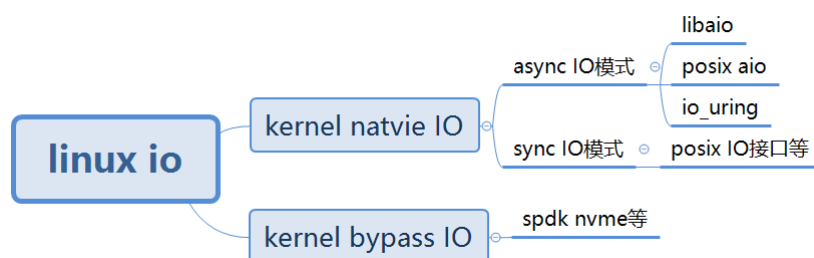


图 23: IO 加速方案

`io_uring` 是 `kernel natvie aio` 的一种，它是 Linux Kernel 5.1 版本加入一个特性。`io_uring` 围绕高效进行设计，其设计了一对共享的 `ring buffer` 用于应用和内核之间的通信，通过该设计实现了如下的三个好处：

1. 避免在提交和完成事件中存在内存拷贝；
2. 避免了 `libaio` 中在提交和完成任务的时候系统调用过程；
3. 该队列采用了无锁的访问模式，通过内存屏障减少了竞争。

`io_uring` 的使用也非常简单，只需要提交 `SQE` 到 `ring buffer` 中，然后等待 `CQE` 即可。由于其简单使用、高效的设计，我们选择了 `io_uring` 作为我们的 IO 加速方案。

其实 `io_uring` 在 RocksDB 的 `PosixFileSystem` 文件系统后端中已经有可选的实现，但是它并没有被实现到 `ZenFS` 后端中，`ZenFS` 的数据读写还是只能使用 Direct IO 进行优化。于是，我们在 `ZenFS` 的文件系统后端中实现了 `io_uring` 的优化，从而进一步加速了 `ZenFS` 的 IO 操作。

为了进一步增大 IO 加速的效果，我们还实现了 IO 操作的批处理。在原版 `ZenFS` 中，IO 操作是以单个系统调用的形式进行的，即每次 IO 操作都是单个请求。但是如果

将多个 IO 操作打包成一个批处理请求，可以进一步提高 IO 操作的效率，尤其是单 IO 请求的效率。

为了实现 IO 操作的批处理，我们在 AquaZFS 的文件系统后端中实现了 IO 请求的分割和合并，并将请求全部打包成一个批处理请求，然后一次性提交给 `io_uring`，从而实现了 IO 操作的批处理。

除了上述优化处理，我们还使用 C++ 20 的协程特性，进一步增大了代码的并行性。在调研过程中，我们找到了一个 C++ 实现的 `io_uring` 绑定库，其向外提供了协程接口，可以在协程中使用 `io_uring`。我们将其移植到了 AquaFS 中，从而实现了在协程中使用 `io_uring`。

有关逻辑可以在数据操作相关的函数中找到，如下以读数据函数 `Read` 为例：

```
1  using req_item_t = std::tuple<int, char *, uint64_t, off_t>;
2  std::vector<req_item_t> requests;
3  std::vector<ZbdlbBackend *> bes(nr_dev());
4  for (decltype(nr_dev()) i = 0; i < nr_dev(); i++) {
5  #ifdef ROCKSDB_USE_RTTI
6      bes[i] = dynamic_cast<ZbdlbBackend *>(devices_[i].get());
7      assert(bes[i] != nullptr);
8  #else
9      bes[i] = (ZbdlbBackend *) (devices_[i].get());
10 #endif
11 }
12 while (size > 0) {
13     // ...
14     requests.emplace_back(fd, buf, req_size, mapped_pos);
15     // ...
16 }
```

如上所示，我们首先将 IO 请求分割成多个小的 IO 请求，将这些请求收集到 `std::vector` 中。由于 ZNS 支持随机读，所以可直接将 IO 请求加入到 `std::vector` 中，而无需考虑 IO 请求的顺序。

收集到 IO 读请求后，我们将这些请求打包成一个批处理请求，然后一次性提交给 `io_uring`，如下所示：

```
1  uio::io_service service;
2  // ...
3  service.run([&]() -> uio::task<> {
4      std::vector<uio::task<int>> futures;
5      for (auto &&req : requests) {
6          uint8_t flags = 0;
```

```

7      // read do not need order
8      // if (req != *req_list.second.cend()) flags |= IOSQE_IO_LINK;
9      futures.emplace_back(service.read(std::get<0>(req), std::get<1>(req),
10                                     std::get<2>(req), std::get<3>(req),
11                                     flags) |
12                                uio::panic_on_err("failed to read!", true));
13    }
14    for (auto &&fut : futures) co_await fut;
15 }();

```

在作为举例的 Read 函数中，我们首先创建了一个 service 对象，然后使用 run 方法启动协程，在协程中逐个将请求提交到 SQE 中，然后等待所有的 CQE 即可。这里的 flags 参数用于指定 IO 请求的属性，如 IOSQE_IO_LINK 表示该 IO 请求是一个批处理请求中的一部分。由于我们的 IO 请求是无序的，所以我们不需要使用 IOSQE_IO_LINK。

在写操作相关的函数中，相比于读操作，还需要基于 ZNS 的顺序写特性考虑更多的因素，如下所示：

```

1  using req_item_t = std::tuple<char *, uint64_t, off_t>;
2  // <dev, zone> -> vec<ordered req>
3  std::map<std::pair<int, idx_t>, std::vector<req_item_t>> requests;

```

我们首先将 IO 请求按照设备和 Zone 序号进行分组，然后将每个 Zone 内的 IO 请求按照顺序进行排序，最后将这些请求收集到 std::vector 中。而在提交 IO 请求时，我们需要将这些请求按照顺序依次提交。

总之，我们在 AquaFS 中实现了 io_uring 和 C++20 协程的优化，从而进一步加速了 AquaFS 的 IO 操作。

4.3 辅助文件系统 ExtFS 的实现

为了进一步提高 AquaFS 的性能，我们还实现了辅助文件系统 ExtFS。

ExtFS 是一个基于 Ext2 文件系统的辅助文件系统，其主要用于辅助存储 AquaZFS 的元数据，如文件的元数据、目录的元数据、文件的数据块映射表等，同时可以分担 AquaZFS 的处理压力。在 RocksDB 数据量特别大的时候，AquaFS 的元数据量同样较大，甚至可能占用几个 GiB 的内存，所以我们可以选择将其存储在 ExtFS 中，从而一定程度上减小 AquaFS 对主机的内存压力。

由于此文件系统设计的时候，我们暂未选定 ZenFS 作为基础参考，所以当前 ExtFS 的设计是参照 Ext2 实现的，兼容 Ext2 rev0.0 版本。ExtFS 的基本布局如图 24 所示，其 inode 寻址方式如图 25。

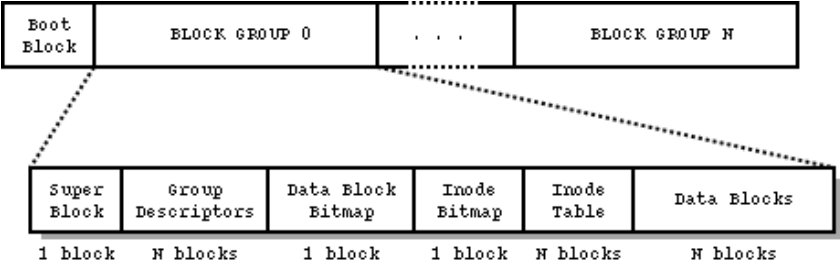


图 24: ExtFS 基本布局

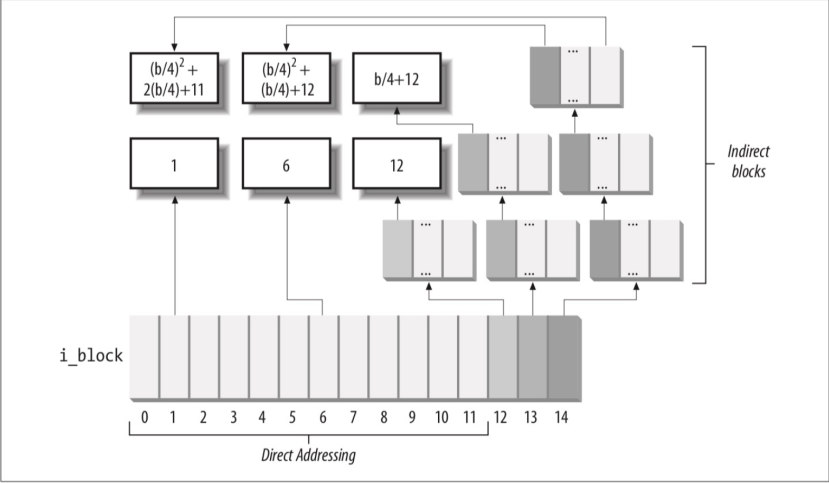


图 25: ExtFS inode 寻址方式

当前，ExtFS 的基本布局与 Ext2 基本一致，主要包括超级块、组描述符表、索引节点表、数据块区域等。其中，超级块用于存储文件系统的基本信息，如文件系统的大小、索引节点表的起始位置、数据块区域的起始位置等。组描述符表用于存储每个组的信息，如索引节点位图的起始位置、数据块位图的起始位置、索引节点表的起始位置等。索引节点表用于存储文件的元数据，如文件的大小、文件的权限、文件的创建时间等。数据块区域用于存储文件的数据块。

由于 ZNS 可以支持 Conv Zones，这些 Zones 与 Seq Zones 共存在同一个设备中，所以我们可以将 ExtFS 的数据块区域、inode 区域等需要原地更新的数据存储在 Conv Zones 中。

Ext4 对于连续大文件存储的针对性优化是它的 Extent 结构。Extent 是一种连续的数据块，它可以存储多个数据块，从而减少索引节点的数量，提高文件系统的性能。在 ExtFS 中，我们可以实现一种存储在 Seq Zones 上的 Extent 结构，从而进一步提高 ExtFS 的性能和磁盘利用率。

当前，ExtFS 的实现还不完善，我们还没有实现 ExtFS 的日志系统、Extent 结构等。现在的 ExtFS 是基于 Rust 语言和 FUSE 接口实现的一个简单的 inode 文件系统，并且有一个内建的块缓存系统。不过经过多种测试，它的实现基本是没有问题的。当前 ExtFS 仍然在建设当中，我们将在后续的工作中进一步完善 ExtFS 的设计。ExtFS 的存在可以促进我们对于融合文件系统的研究，同时也可以为后续的工作提供一个基础。

4.4 AquaFS 文件系统的智能调参模块实现

智能调参模块方面，实现了基于方差的重要参数选择方案和基于高斯过程回归的参数调整方案。

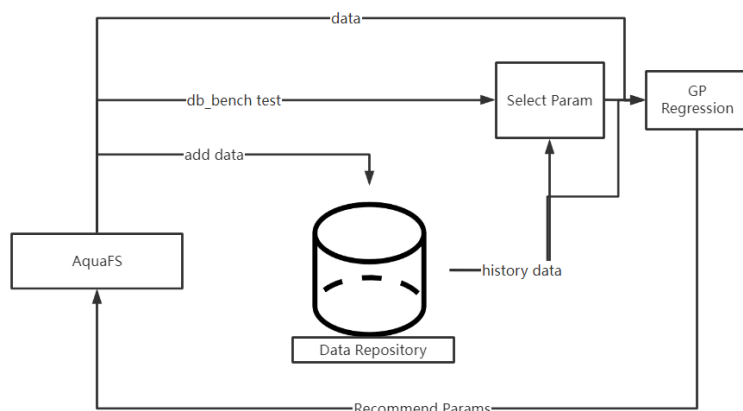


图 26: AquaTurnner 智能调参模块

如图 26 所示，AquaTurnner 智能调参模块主要由三部分组成，被调对象，本文中是

AquaFS，历史数据存储仓库，以及调参事务组成，调参事务又包括选择参数选择模块和高斯过程回归调整参数模块。

首先，AquaFS 需要预热地运行，收集到不同参数配置下的目标指标，本文中使用的是吞吐量指标，将对应的参数配置和目标指标值存入数据仓库中。在初次预热之后，后续不须要预热，除非加入了新的参数指标。

考虑到 AquaFS 作为 RocksDB 的插件存在，在本文中使用 RocksDB 的测试脚本 db_bench 来测试系统的吞吐量，采用 prometheus 作为收集数据的工具。在这个测试中，用收集到的目标指标值和配置参数值，配合历史数据仓库中的数据，来进行参数选择和高斯过程回归。参数选择模块根据方差指标选择最重要的几个参数，高斯过程回归用重要的参数和目标指标值进行拟合以及回归。

首先对于最重要的参数，由方差计算的最大值对应的参数得到：

$$\begin{aligned} Var(S) &= \frac{1}{|S|} \sum_{i=1}^{|S|} (y_i - \mu)^2 \\ PI(P) &= Var(S) - \sum_{i=1}^N \frac{S_{P=P_i}}{S} Var(S_{p=p_i}) \end{aligned} \quad (1)$$

这里的 y_i 是样本中的目标值， μ 是样本目标值均值， PI 系数是基于方差来计算的，即固定某一个参数的值，根据参数的值划分集合，在每个集合中求出集合中目标值的对应方差，再用初始方差 $Var(S)$ 减去这个和，这个 PI 系数越大说明原来的这个参数的影响越大，因为在同一个值的情况下，集合内方差的和很小。

其次由 CPI 指标来选择剩余重要参数：

$$\begin{aligned} CPI(Q|P = p) &= Var(S_{P=p}) - \sum_{j=1}^m \frac{S_{Q=Q_{w_j}, P=p}}{S_{P=p}} Var(S_{Q=q_i|P=p}) \\ CPI(Q|P = p) &= \max_{1 \leq i \leq n} CPI(Q|P = p_i) \end{aligned} \quad (2)$$

基于方差的重要参数选择算法如算法 1 所示。

对于连续参数，AquaTuner 对于参数指标在参数范围内给出合适的推荐参数值，该合适参数值是在拟合的高斯过程模型曲线上，根据在最优配置参数附近做抖动获得，也即尝试最优配置参数点附近的参数值看目标指标是否有所提升，对于离散参数，AquaTuner 尝试匹配最优目标指标值对应的配置参数的离散值。

AquaTuner 的运行算法流程如算法 2 所示。

4.5 磨损均衡模块的实现

为了进一步提高 AquaFS 的性能，使得软硬件更加协调，我们还实现了磨损均衡模块。

Algorithm 1 AquaTuner 参数选择算法

Input: adjust_param_num, db_bench_data, data in repository

Output: important_params

- 1: important_params = []
 - 2: Select the most important param by $PI(param)$, add to important_params;
 - 3: For i in adjust_param_num - 1:
 - 4: Compute CPI for each param that not in important_params;
 - 5: Select the largest CPI 's corresponding params to important params;
 - 6: **return** important_params
-

闪存都是有寿命的，即闪存块有擦写次数限制。一个闪存块，如果其擦写次数超过一定的值，那么该块就变得不那么可靠了，甚至变成坏块不能用了。如果不做磨损平衡，则有可能出现有些闪存块频繁拿来擦写，这些闪存块很容易就会寿终正寝。随着不断的写入，越来越多的坏块出现，最后导致 SSD 在保质期前就失效。

由于 ZNS 设备也是一种闪存设备，我们决定在其上实现一个磨损均衡模块来模拟 FTL 的功能。

磨损均衡的实现是添加在 RAID 映射和底层 zone 之间，我们为每个设备维护一个树来记录对应 zone 号和使用次数，同时在每次分配 zone 的时候，选择上次使用次数最少的 RAID zone 作为分配对象进行分配。

磨损均衡模块中单个设备为了磨损均衡所维护的数据结构如下图所示：

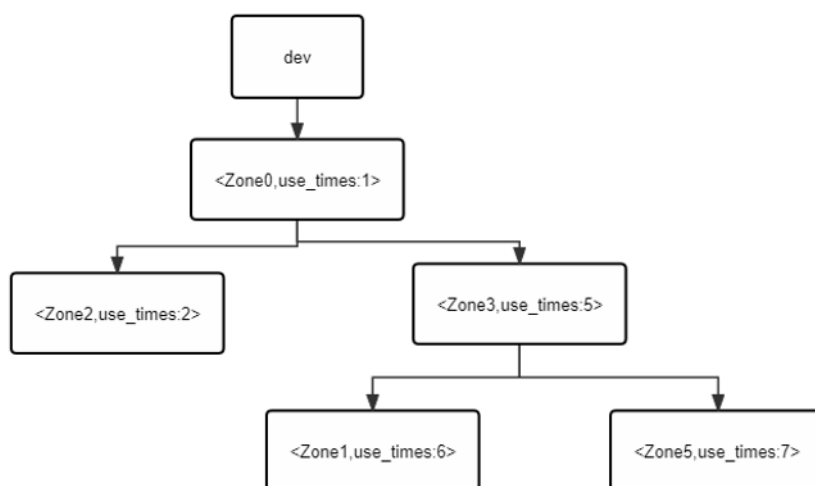


图 27: 磨损均衡

Algorithm 2 AquaTuner 参数调优算法

Input: adjust_param_num**Output:** recommend_param

```
data repository <- warm up system and collect data;
2: start db_bench;
   db_bench_data = collect data from db_bench;
4: add db_bench_data to data repository;
   important_params = Select_Param(adjust_param_num ,db_bench_data, data in repository);
6: GP_model = GP_regression(important_params, db_bench_data, data in repository);
   recommend_param = [];
8: For param in history_best_params and important params:
   If param is continuous:
10:     Try values near the past value, add to recommend_param;
   Else if param is discrete:
12:     Try values in best params, add to recommend_param;
   Target = GP_model.predict(recommend_param);
14: If Target is better:
   return recommend_param;
16: return history_best_param;
```

5 系统测试

5.1 智能分区 RAID 模块和 IO 加速测试

由于 ZNS 当前仅有西数公司的设备支持，我们原本预计使用西数的 ZNS SSD 进行测试，即 SN540 512GB。但是由于西数的 SSD 申请过程非常复杂，国内外沟通流程也比较慢，现在仍在处理流程中，我们无法在短时间内获得设备，只能使用仿真环境进行测试。

在几个月前，FAST23 上有一篇关于 ZNS 的论文 [nvmevirt^{\[15\]}](#)，它类似 SPDK，在用户态实现了 NVME 的协议，也包括了 ZNS 的协议，可以在 Linux 上模拟 ZNS SSD。但是我们在测试过程中发现，它的 ZNS 模拟实现有问题，无法正常使用，我们在 Github 上提了 issue，但是现在项目还在持续更新中，我们测试中遇到的问题只解决了一些，但是又有更多新的问题出现。我们尝试去解决这些问题，但是由于时间有限，我们最终无法在初赛前解决这些问题，只能放弃在初赛中使用这个模拟器。

以下 RAID IO 测试均使用了基于数据的延迟计算仿真，即计算 SSD 的访问延迟和

数据传输延迟，其数据存储后端是内存。由于内存的访问延迟远小于 SSD，因此可以忽略内存的访问延迟，只计算 SSD 的访问延迟和数据传输延迟。

测试环境为：Intel(R) Core(TM) i7-12700 CPU，内存 40GB 3200MHz；系统为 Arch-linux，Linux 内核版本 6.3.5-arhc1-1。由于 libzbd、ZoneFS、nvmevirt 等都需要比较高的内核版本，于是这里直接一步到位地使用了最新的 Linux Kernel mainline。

5.1.1 AquaZFS 数据完整性测试

由于 AquaZFS 是作为一个插件运行在 RocksDB 的文件系统层上的，所以可以使用 RocksDB 来验证其数据完整性。我们使用 RocksDB 的 db_bench 工具进行测试，测试结果如表 3 所示。

表 3: 数据完整性测试

测试项	设备数	测试参数	测试结果
全盘 RAID C	2	aquafs://raidc:dev:nullb0,dev:nullb1	通过
全盘 RAID 0	2	aquafs://raid0:dev:nullb0,dev:nullb1	通过
全盘 RAID 1	2	aquafs://raid1:dev:nullb0,dev:nullb1	通过
全盘 RAID 5	2	aquafs://raid5:dev:nullb0,dev:nullb1	通过
分区 RAID A	2	aquafs://raidA:dev:nullb0,dev:nullb1	通过
分区 RAID A	4	aquafs://raidA:dev:nullb0,dev:nullb1, dev:nullb2,dev:nullb3	通过

db_bench 除了表 3 中的，还需要添加如下参数：

```

1 ./plugin/aquafs/aquafs
2 mkfs # 创建文件系统
3 # --raids= 此选项用于指定数据后端，这里指定使用 AquaFS 的 RAID 功能
4 --aux_path=/tmp/aquafs # 指定 AquaFS 的辅助数据存储路径，如锁文件和日志缓存文件
5 --force # 清除原有的文件系统数据
6
7 ./db_bench
8 # --fs_uri= 此选项用于指定数据后端，可以是文件系统路径，也可以是 AquaFS 的相关配置
9 --benchmarks=fillrandom # 指定测试项，使用随机写并校验
10 --use_direct_io_for_flush_and_compaction # 使用 Direct IO 加速
11 --use_stderr_info_logger # 在标准错误输出日志

```

由于输出较多，这里仅展示一部分输出，如图 28、29、30 所示。

```
aquafis mkfs raidc x db_bench aquafis raidc x
/home/chiro/programs/rocksdb/build-release/db_bench --fs_uri=aquafis://raidc:/dev/nullb0,/dev/nullb1 --benchmarks=fillrandom --use_direct_io_for_flush_and_compaction --use_stderr_info_logger
[RAID] RAID Mode: raidc Devices:
[RAID] /dev/nullb0
[RAID] /dev/nullb1
[RAID] Open(readonly=0, exclusive=1)
[RAID] /dev/nullb0 opened, sz=80000000, nr_zones=40, zone_sz=2000000 blk_sz=200 max_active_zones=0, max_open_zones=0
[RAID] /dev/nullb1 opened, sz=80000000, nr_zones=40, zone_sz=2000000 blk_sz=200 max_active_zones=0, max_open_zones=0
[RAID] after Open(): nr_zones=80, zone_sz=2000000 blk_sz=200
AquaFS Initializing
AquaFS parameters: block device: raidc:/dev/nullb0,/dev/nullb1, aux filesystem: PosixFileSystem
AquaFS Initializing
Found OK superBlock in zone 0 seq: 1

Recovered from zone: 0
Rolling to metazone 1

SuperBlock sequence 2
Finish threshold 0
Filesystem mount OK
Resetting unused IO Zones..
Done
Files:

Sum of all files: 0 MB of data

2023/06/06-16:37:23.060267 b397 RocksDB version: 8.2.0

2023/06/06-16:37:23.060298 b397 Git sha e8eb47f98f547f2e40807b5010bc44843dd7e4ba
2023/06/06-16:37:23.060299 b397 Compile date 2023-06-05 13:13:49
2023/06/06-16:37:23.060303 b397 DB SUMMARY

2023/06/06-16:37:23.060308 b397 DB Session ID: C03R3WRNLUZV379W91NIJ

2023/06/06-16:37:23.060312 b397 SST files in rocksdbtest/dbbench dir, Total Num: 0, files:

2023/06/06-16:37:23.060314 b397 Write Ahead Log file in rocksdbtest/dbbench:

2023/06/06-16:37:23.060316 b397 Options.error_if_exists: 0
2023/06/06-16:37:23.060317 b397 Options.create_if_missing: 1
2023/06/06-16:37:23.060319 b397 Options.paramoid_checks: 1
```

图 28: 数据完整性测试 1

```
aquafis mkfs raidc x db_bench aquafis raidc x
** File Read Latency Histogram By Level [default] **

2023/06/06-16:37:26.249751 b397 [db/db_impl/db_impl.cc:489] Shutdown: canceling all background work
Set seed to 16860406463059687 because --seed was 0
Initializing RocksDB Options from the specified file
Initializing RocksDB Options from command-line flags
Integrated BlobDB: blob cache disabled
Keys: 16 bytes each (+ 0 bytes user-defined timestamp)
Values: 100 bytes each (50 bytes after compression)
Entries: 1000000
Prefix: 0 bytes
Keys per prefix: 0
RawSize: 110.6 MB (estimated)
FileSize: 62.9 MB (estimated)
Write rate: 0 bytes/second
Read rate: 0 ops/second
Compression: Snappy
Compression sampling rate: 0
MemtableRep: SkipListFactory
Perf Level: 1
-----
Initializing RocksDB Options from the specified file
Initializing RocksDB Options from command-line flags
Integrated BlobDB: blob cache disabled
DB path: [rocksdbtest/dbbench]
fillrandom : 3.169 micros/op 315530 ops/sec 3.169 seconds 1000000 operations; 34.9 MB/s
2023/06/06-16:37:26.430359 b39a EVENT_LOG_v1 {"time_micros": 1686040646430349, "cf_name": "default", "job": 3, "event": "table_file_creation", "file_number": 11, "file_size": 25312161, "file_
468775, "largest_seqno": 921552, "table_properties": {"data_size": 25174973, "index_size": 222551, "index_partitions": 0, "top_level_index_size": 0, "index_key_is_user_key": 1, "index_value_
"raw_average_key_size": 24, "raw_value_size": 36892400, "raw_average_value_size": 100, "num_data_blocks": 11180, "num_entries": 368924, "num_filter_entries": 0, "num_merg
"fixed_key_len": 0, "filter_policy": "", "column_family_name": "default", "column_family_id": 0, "comparator": "LevelDBBytewiseComparator", "merge_operator": "nullptr", "prefix_extractor_na
"compression_options": {"window_bits": -14, "level": 32767, "strategy": 0; max_dict_bytes: 0; zstd_max_train_bytes: 0; enabled: 0; max_dict_buffer_bytes: 0; use_zstd_dict_trainer: 1; }, "creation_time": 16
1686040646, "slow_compression_estimated_data_size": 0, "fast_compression_estimated_data_size": 0, "db_id": "d04e7ed9-d882-44df-8894-ed7c14b14114", "db_session_id": "C03R3WRNLUZV379W91NIJ", "co
2023/06/06-16:37:26.430384 b39a [db/flush_job.cc:1017] [default] [JOB 3] Flush lasted 380684 microseconds, and 118861 cpu microseconds.

2023/06/06-16:37:26.430399 b39a (Original Log Time 2023/06/06-16:37:26.430376) [db/flush_job.cc:967] [default] [JOB 3] Level-0 flush table #11: 25312161 bytes OK
2023/06/06-16:37:26.430401 b39a (Original Log Time 2023/06/06-16:37:26.430392) EVENT_LOG_v1 {"time_micros": 1686040646430390, "job": 3, "event": "flush_finished", "output_compression": "Snapp
2023/06/06-16:37:26.432227 b397 [File/delete_scheduler.cc:73] Deleted file rocksdbtest/dbbench/000011.sst immediately, rate_bytes_per_sec 0, total_trash_size 0 max_trash_db_ratio 0.250000
2023/06/06-16:37:26.432252 b397 EVENT_LOG_v1 {"time_micros": 1686040646432240, "job": 4, "event": "table_file_deletion", "file_number": 11}
2023/06/06-16:37:26.443809 b397 [db/db_impl/db_impl.cc:692] Shutdown complete
```

图 29: 数据完整性测试 2

```
2023/06/06-16:37:26.430399 b39a [db/flush_job.cc:1017] [default] [JOB 3] Level-0 Flush table #11: 25312161 bytes OK
2023/06/06-16:37:26.430399 b39a (Original Log Time 2023/06/06-16:37:26.430376) [db/flush_job.cc:967] [default] [JOB 3] Level-0 Flush table #11: 25312161 bytes OK
2023/06/06-16:37:26.430401 b39a (Original Log Time 2023/06/06-16:37:26.430392) EVENT_LOG_v1 {"time_micros": 1686040646430398, "job": 3, "event": "flush_finished", "output_compression": "Snappy", "compression_options": {"window_bits": -14; level: 32767; strategy: 0; max_dict_bytes: 0; zstd_max_train_bytes: 0; enabled: 0; max_dict_buffer_bytes: 0; use_zstd_dict_trainer: 1; }, "creation_time": 1686040646, "slow_compression_estimated_data_size": 0, "fast_compression_estimated_data_size": 0, "db_id": "d04e7ed9-d882-44df-8894-ed7c14b14114", "db_session_id": "C03R3JWRNLU2V379W91NT", "db_session_name": "C03R3JWRNLU2V379W91NT"}
2023/06/06-16:37:26.432227 b397 [file/delete_scheduler.cc:73] Deleted file rocksdbtest/dbbench/000011.sst immediately, rate_bytes_per_sec 0, total_trash_size 0 max_trash_db_ratio 0.250000
2023/06/06-16:37:26.432252 b397 EVENT_LOG_v1 {"time_micros": 1686040646432248, "job": 4, "event": "table_file_deletion", "file_number": 11}
2023/06/06-16:37:26.443809 b397 [db/db_impl/db_impl.cc:692] Shutdown complete
AquaFS unmounting...
Files:
/rocksdbtest/dbbench/000008.log sz: 63600556 lh: 2 sparse: 1
  Extent 0 {start=0x1200000, zone=9, len=1048568}
  Extent 1 {start=0x1210000, zone=9, len=1048568}
  Extent 2 {start=0x1220000, zone=9, len=1048568}
  Extent 3 {start=0x1230000, zone=9, len=1048568}
/rocksdbtest/dbbench/000009.sst sz: 25331441 lh: 3 sparse: 0
  Extent 0 {start=0x1000000, zone=8, len=25331441}
/rocksdbtest/dbbench/000010.log sz: 10828068 lh: 2 sparse: 1
  Extent 0 {start=0x1400000, zone=10, len=1048568}
  Extent 1 {start=0x1410000, zone=10, len=1048568}
  Extent 2 {start=0x1420000, zone=10, len=1048568}
  Extent 3 {start=0x1430000, zone=10, len=1048568}
/rocksdbtest/dbbench/CURRENT sz: 16 lh: 0 sparse: 0
  Extent 0 {start=0xa000000, zone=5, len=16}
/rocksdbtest/dbbench/IDENTITY sz: 36 lh: 0 sparse: 0
  Extent 0 {start=0x8000000, zone=4, len=36}
/rocksdbtest/dbbench/MANIFEST-000005 sz: 208 lh: 0 sparse: 0
  Extent 0 {start=0x8000600, zone=4, len=66}
  Extent 1 {start=0x8000800, zone=4, len=142}
/rocksdbtest/dbbench/OPTIONS-000007 sz: 6999 lh: 0 sparse: 0
  Extent 0 {start=0xa000200, zone=5, len=6999}
Sum of all files: 95 MB of data
AquaFS unmounted
进程已结束,退出代码0
```

图 30: 数据完整性测试 3

5.1.2 使用 RAID 0 加速文件读写

由于 ZenFS 并不支持 POSIX 规范的文件系统访问接口, 所以无法将其挂载到 Linux 的 VFS 上, 因此无法使用 Linux 的文件系统测试工具来测试其性能。这里使用的测试逻辑, 与 RocksDB 使用 AquaFS 文件系统插件类似, 都是通过写程序调用 AquaFS 的 API 来测试性能。

使用如下逻辑测试文件读写性能, 并比较 RAID 0 的性能加速:

1. 生成测试参数, 包括文件大小、设备数量、RAID 逻辑、随机种子等
2. 创建文件系统, 设置 RAID 逻辑等
3. 在内存中生成一个指定大小的随机内容文件
4. 重复进行多次:
 - (a) 通过 `aquafs restore` 将这个文件写入 AquaFS 文件系统
 - (b) 通过 `aquafs dump` 将文件从 AquaFS 文件系统读出
 - (c) 通过 `std::hash` 和 `md5sum` 比较读出的文件和原文件是否一致
5. 计算平均耗时, 得到读写平均时间

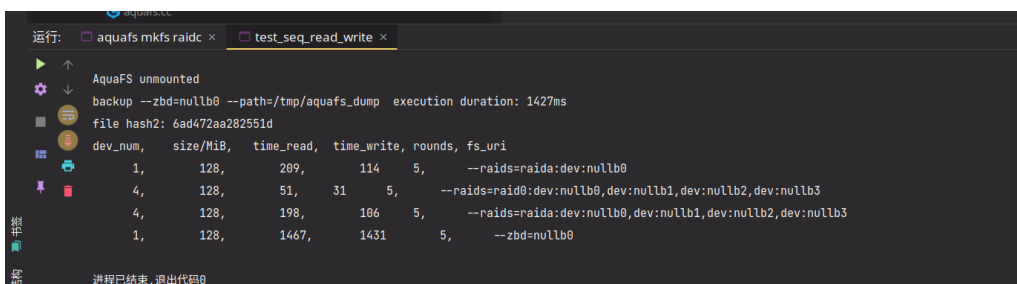


图 31: RAID 0 加速测试

对多组数据进行测试，结果输出如图 31 所示：
其得到的数据如表 4 所示：

表 4: RAID 0 加速测试，5 次运行取平均

设备数量	文件大小 (MiB)	平均读时间 (ms)	平均写时间 (ms)	说明
1	128	209	114	单盘分区 RAID
4	128	51	31	多盘全盘 RAID 0
4	128	198	106	多盘分区 RAID
1	128	1467	1431	ZenFS 默认

可以看到，在单盘分区 RAID 的情况下，读写性能都有至少 N 倍的性能提升，而在多盘全盘 RAID 0 的情况下，读写性能有更大提升。

相比与 ZenFS 的默认读写，单盘分区读写由于使用 io_uring、协程、重新构造读写请求等方式，性能有成倍的提升。具体提升幅度被能同时读写的 Zone 数量限制；而原版 ZenFS 只能单线程进行数据请求，因此性能较低。

相比与单设备的分区 RAID，多盘分区 RAID 有更大的性能提升。多盘意味着系统并行度更大，因此获得了更高的瞬时吞吐量。

而多盘全盘 RAID 0，由于采用的读写加速方式与分区 RAID 一致，其逻辑更加精简，因此性能更高。

5.1.3 文件系统数据恢复测试

测试逻辑：

1. 生成随机大文件
2. 使用 `aquafs restore` 将文件写入 AquaFS 文件系统
3. 在软件层面触发模拟硬件故障，使得一个 Dev Zone 下线
4. 使用 `aquafs dump` 将文件从 AquaFS 文件系统读出，观察是否能够正常读出

5. 校验读出的文件和原文件是否一致

测试流程图如下所示：



图 32: 测试流程图

使用 128 MiB 文件大小进行测试，测试过程如图 33 所示。在测试中，AquaFS 成功地检测到了 Zone 的故障，并且成功地从其他 Zone 中恢复了数据，维护了文件系统的数据完整性。

```
/test_file
visiting offline zone! pos=0, size=1000000, wp-start=1000000, wp=1000000, start=0
[RAID] [WARN] found offline zone: dev 1 zone 0, raid zone sub 10
[RAID] this mp before handle: [mp-before] raid zone %x: dev 1, zone 0; dev 2, zone 0;
[RAID] [WARN] remove mapping for dev 1 zone 0
[RAID] this mp: [mp] raid zone 16: dev 2, zone 0; dev 1, zone 336;
[RAID] fine zone: dev=2, zone=0, wp=1000000, start=0
[RAID] restoring data to dev 1 zone 150, sz=1000000, pos=15000000
AquaFS unmounting...
Files:
```

图 33: 数据恢复测试

其中，测试的 RAID0 和 RAID5 的恢复得到了正确结果：

5.1.4 文件系统性能测试

由于可以使用内存作为存储后端，我们可以很便利地分析 AquaFS 的性能瓶颈，从而快速优化其性能。

我们使用 Linux perf 工具进行性能分析，其中部分分析过程如图 35 所示。

可以看到，AquaFS 的性能瓶颈主要在于：

```

[RAID] [WARN] setting dev 1 zone 0 to offline!
setting idx=0 to offline!

backup --raids=raid0:dev:nullb0,dev:nullb1,dev:nullb2,dev:nullb3 --path=/tmp/aquafs_dump execution duration: 702ms
4c1d946e3e464d63ab9b431f464997fb /tmp/aquafs_test/test_file
4c1d946e3e464d63ab9b431f464997fb /tmp/aquafs_dump/test_file
file hash2: e169cc1594b26464

-

backup --raids=raid5:dev:nullb0,dev:nullb1,dev:nullb2,dev:nullb3 --path=/tmp/aquafs_dump execution duration: 25599ms
82111ba68a257a38827163c5e13e0bd8 /tmp/aquafs_test/test_file
82111ba68a257a38827163c5e13e0bd8 /tmp/aquafs_dump/test_file
file hash2: 6d6b637b26a3f815

```

图 34: RAID0 RAID5 数据恢复测试

```

samples: 55k of event 'cpu_core/cycles/;', Event count (approx.): 21695559409
Children Self Command Shared Object Symbol
- test_seq_read_write
- 17.51% aquafs::aquafs_tools_call
- aquafs::aquafs_tools
- 11.41% aquafs::aquafs_tool_restore
- 11.40% aquafs::aquafs_tool_copy_dir
- 11.39% aquafs::aquafs_tool_copy_file
- 11.37% aquafs::ZonedWritableFile::Append
- 10.85% aquafs::ZonedWritableFile::BufferedWrite
- aquafs::ZonedWritableFile::FlushBuffer
- aquafs::ZoneFile::BufferedAppend
- 10.83% aquafs::Zones::Append
- 9.55% aquafs::Raid0ZonedBlockDevice::Write
- 9.18% 0x7f66ccac6b6f
- 9.18% entry_SYSCALL_64
- do_syscall_64
- 9.00% __do_sys_io_uring_enter
- 7.46% io_submit_sqes
- 6.88% io_issue_sqe
- 6.62% io_write
- 6.17% blkdev_write_iter
- 6.05% __generic_file_write_iter
- 5.78% generic_file_direct_write
- 5.12% __blkdev_direct_IO_async
- 2.23% submit_bio_noacct_nocheck
- 1.95% __submit_bio
- blk_mq_submit_bio
- 1.13% dd_bio_merge
- 0.82% blk_mq_sched_try_merge
- 1.43% bio_iov_iter_get_pages
- 1.18% iov_iter_get_pages
- __iov_iter_get_pages_alloc
- 1.03% internal_get_user_pages_fast
- 1.17% bio_alloc_bioset
- 0.76% mempool_alloc
- 0.67% kmem_cache_alloc
- 1.51% io_run_task_work
- task_work_run
- 1.50% kctx_task_work
- 0.74% __io_submit_flush_completions
- 0.61% io_cqring_event_overflow
- 0.57% __kmalloc
- 0.51% __kmem_cache_alloc_node
- 0.57% io_req_rw_complete
- 0.53% __fsnotify_parent
- 1.24% 0x7f66ccac6b6f
- entry_SYSCALL_64
- do_syscall_64
- 1.11% __do_sys_io_uring_enter
- 0.78% __io_cqring_overflow_flush
- 0.51% 0x7f66cc56ce09
- 6.05% aquafs::aquafs_tool_backup
- 6.03% aquafs::aquafs_tool_copy_dir
- aquafs::aquafs_tool_copy_file
- 4.40% aquafs::ZonedSequentialFile::Read
- aquafs::ZoneFile::PositionedRead
- 4.39% aquafs::ZonedBlockDevice::Read
- 2.48% aquafs::Raid0ZonedBlockDevice::Read
- 1.74% 0x7f66ccac6b6f
- entry_SYSCALL_64
- do_syscall_64
- 1.71% __do_sys_io_uring_enter
- 1.68% io_submit_sqes
- 0.90% __to_alloc_req_refill
- 0.76% kmem_cache_alloc_bulk

```

图 35: perf 性能分析

1. entry_SYSCALL_64: 当前仍然使用内核态的 open、read、write 等系统调用, 导致系统频繁地从用户态切换到内核态, 导致性能瓶颈。
2. io_submit_sqes: 正在使用 io_uring 的软件层面实现, 可能使用方法还有可以优化的地方, 造成 io_uring 中一部分数据的锁争用。

5.1.5 ExtFS 功能和性能测试

为了测试我们 ExtFS 的功能正确性, 我们使用了本校操作系统实验课程中的测试脚本, 对 ExtFS 进行了完整的功能测试, 测试结果如下所示:

```
1  $ ./test.sh
2  开始mount, mkdir, touch, ls, read&write, cp, umount测试测试脚本工程根目录:
3    /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests
4  测试用例: /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/stages/mount.sh
5  测试用例: /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/stages/mkdir.sh
6  测试用例: /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/stages/touch.sh
7  测试用例: /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/stages/ls.sh
8  测试用例: /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/stages/remount.sh
9  测试用例: /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/stages/rw.sh
10 测试用例: /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/stages/cp.sh
11  ===== Finished dev [unoptimized + debuginfo]
    target(s) in 0.05s
12    Running `/home/chiro/os/fuse-ext2/fs/rfs/rfs/target/debug/rfs
    --device=/home/chiro/ddriver
13    -q /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt`
14  pass: case 1 - mount
15  =====pass: case 2.1 - mkdir
    /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0
16  pass: case 2.2 - mkdir /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0/dir0
17  pass: case 2.3 - mkdir
    /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0/dir0/dir0
18  pass: case 2.4 - mkdir /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir1
19  =====pass: case 3.1 - touch
    /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/file0
20  pass: case 3.2 - touch /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/file1
21  pass: case 3.3 - touch /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0/file1
22  pass: case 3.4 - touch /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0/file2
23  pass: case 3.5 - touch /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir1/file3
24  =====pass: case 4.1 - ls
    /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/
```

```

25 pass: case 4.2 - ls /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0
26 pass: case 4.3 - ls /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0/dir1
27 pass: case 4.4 - ls /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/dir0/dir1/dir2
28 ===== Finished dev [unoptimized + debuginfo] target(s)
    in 0.05s
29     Running `/home/chiro/os/fuse-ext2/fs/rfs/rfs/target/debug/rfs
        --device=/home/chiro/ddriver
30     -q /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt`
31 pass: case 5.1 - umount /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt
32 pass: case 5.2 - check bitmap
33 ===== Finished dev [unoptimized + debuginfo]
    target(s) in 0.06s
34     Running `/home/chiro/os/fuse-ext2/fs/rfs/rfs/target/debug/rfs
        --device=/home/chiro/ddriver
35     -q /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt`
36 pass: case 6.1 - write /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/file0
37 pass: case 6.2 - read /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/file0
38 =====pass: case 7.1 - prepare content of
        /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/file9
39 pass: case 7.2 - copy /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/file9 to
40     /home/chiro/os/fuse-ext2/fs/rfs/rfs/tests/mnt/file10
41 =====
42 Score: 34/34
43 pass: 通过所有测试 (34/34)

```

接着使用 dd 工具，在本机的普通 1TiB SSD 上进行了简单的单文件读写测试：

```

1 $ dd if=/dev/random of=/mnt/random bs=1MiB count=64
2 输入了 64+0 块记录输出了 64+0 块记录67108864 字节 (67 MB, 64 MiB) 已复制, 5.88972
    s, 11.4 MB/s
3 $ dd of=/dev/null if=/mnt/random bs=1MiB count=64
4 输入了 64+0 块记录输出了 64+0 块记录67108864 字节 (67 MB, 64 MiB) 已复制, 0.635838
    s, 106 MB/s

```

使用实验中的脚本进行读写性能测试，其测试逻辑为对文件中的某一固定的块反复读写：

```

1 // 关闭模拟磁盘延迟，并且打开了缓存
2 Test loop: 1000000, Cache Blks: 512
3 Finished release [optimized] target(s) in 0.05s
4 Running `target/release/rfs --format -q -c --cache_size 512 /home/chiro/mnt`

```



```

5 Time: 30815.510034561157ms BW: 253.52492920733374MB/s
6 // 关闭模拟磁盘延迟, 并且关闭了缓存
7 Test loop: 100000, Cache Blks: 0
8 Finished release [optimized] target(s) in 0.05s
9 Running `target/release/rfs --format -q /home/chiro/mnt`
10 Time: 8691.662073135376ms BW: 89.88499477156695MB/s
11 // 打开了模拟磁盘延迟, 并且关闭了缓存
12 Test loop: 1000000, Cache Blks: 512
13 Finished release [optimized] target(s) in 0.06s
14 Running `target/release/rfs --format -q --latency -c --cache_size 512
    /home/chiro/mnt`
15 Time: 30927.71863937378ms BW: 252.6051174707074MB/s
16 // 打开了模拟磁盘延迟, 并且关闭了缓存
17 Test loop: 100, Cache Blks: 0
18 Finished release [optimized] target(s) in 0.05s
19 Running `target/release/rfs --format -q --latency /home/chiro/mnt`
20 Time: 9035.56513786316ms BW: 0.08646387780728894MB/s

```

可以看到, 关闭缓存后, 读写性能相比有缓存的大幅下降; 而开启缓存后, 读写性能相比无缓存的大幅提升。ExtFS 上的缓存工作正常且非常有效。在关闭了手动管理的缓存后, 由于 Linux 本身为块设备提供了缓存, 因此在不模拟磁盘延迟时关闭了手动管理的缓存, 读写性能并没有下降太多。

由于 ExtFS 完成度还不高, 所以 ExtFS 暂时还不是本项目的重点, 后续会继续完善 ExtFS 的功能。

5.2 AquaTurnner 智能调参模块测试

首先进行 warm_up 得到一组参数和目标指标值的 csv 文件如图 36 所示, 现在由于只调整 AquaFS 参数, 参数空间较小, 主要包括垃圾回收的 gc 相关参数, 块大小以及 zone 的大小参数等, 后续考虑在融合文件系统后加入 inode 相关参数。

设定参数进行 AquaTuner 的参数推荐流程, 离散参数直接指定:

```

1 SECT_SIZE_PARAM = 128
2 ZONE_SIZE_PARAM = 64

```

连续参数可以直接通过脚本 defconfig 获取。

接下来利用 db_bench 进行参数跑分以及收集参数和目标值:

```

1 pre_throughput = 0
2 now_throughput = 0

```

```
gc_start_level,gc_slope,gc_sleep_time,sect_size,zone_size,finish_threshold,ZBD_ABSTRACT_TYPE,RAID_LEVEL,TARGET
20.0,3.0,10000.0,128.0,16.0,0.0,1.0,1.0,24439451.0
20.0,3.0,10000.0,128.0,32.0,0.0,1.0,1.0,19376528.0
20.0,3.0,10000.0,128.0,64.0,0.0,1.0,1.0,11420074.0
20.0,3.0,10000.0,512.0,16.0,0.0,1.0,1.0,12204121.0
20.0,3.0,10000.0,512.0,32.0,0.0,1.0,1.0,12265685.0
20.0,3.0,10000.0,512.0,64.0,0.0,1.0,1.0,12240659.0
20.0,3.0,10000.0,1024.0,16.0,0.0,1.0,1.0,12245765.0
20.0,3.0,10000.0,1024.0,32.0,0.0,1.0,1.0,12330052.0
20.0,3.0,10000.0,1024.0,64.0,0.0,1.0,1.0,12506591.0
20.0,3.0,10000.0,2048.0,16.0,0.0,1.0,1.0,12472544.0
20.0,3.0,10000.0,2048.0,32.0,0.0,1.0,1.0,12483729.0
20.0,3.0,10000.0,2048.0,64.0,0.0,1.0,1.0,12126323.0
20.0,3.0,10000.0,4096.0,16.0,0.0,1.0,1.0,9306903.0
20.0,3.0,10000.0,4096.0,32.0,0.0,1.0,1.0,18367566.0
20.0,3.0,10000.0,4096.0,64.0,0.0,1.0,1.0,18318972.0
```

图 36: 调参测试数据

```
3 for _ in range(1):
4     sect_size = SECT_SIZE_PARAM
5     zone_size = ZONE_SIZE_PARAM
6     total_throughput = []
7     for i in range(2):
8         create_null_blk(sect_size, zone_size, 0, 64)
9         os.system(CREATE_TMP_FILE)
10        throughput_list = execute_adjust_param(2, sect_size, zone_size)
11        total_throughput = total_throughput + throughput_list
12        print("throughput list : {}".format(total_throughput))
13        remove_null_blk()
14    print("sect_size:{}".format(sect_size))
15    print("zone_size:{}".format(zone_size))
16    pre_throughput = now_throughput
17    now_throughput = np.average(total_throughput)
```

代码的整体流程是，首先创建 zone 块，再在 zone 块上创建文件，得到跑分的吞吐量以及推荐参数，再将块 zone 块删除。

创建文件系统如下图所示，创建 AquaFS 的数据模块 nullb0 和作为 log 等文件存储的模块/tmp/aquafs。

```
1 CREATE_TMP_FILE = "mkdir -p /tmp/aquafs ;\
2 sudo ../build/plugin/aquafs/aquafs mkfs --zbd nullb0 --aux-path /tmp/aquafs"
```

之后用 db_bench 进行跑分，并进行智能调参模块的逻辑，按照指定推荐的离散参数跑一遍智能调参模块的结果如图 37 所示。

在上图中推荐了两个参数，最重要的参数是 sect_size, 其次是 zone_size, 这符合现阶段的测试预期，因为我们的垃圾回收也就是 gc 还没有进行触发，，所以能够更改的参数现阶段只有块大小和 zone_size, 而调参模块也根据现阶段的数据仓库中最优的参数

```

select 2 params
12290430054641.78 9603880918309.021 2686549136332.758
Select 3 param, param name: sect_size, var:2686549136332.758
param set: {} ((128.0,)): [0, 1, 2, 15, 16, 17, 18], (512.0,): [3, 4, 5], (1024.0,): [6, 7, 8], (2048.0,): [9, 10, 11], (4096.0,): [12, 13, 14]]
result: [0.0, 0.0, 0.0, 18146151448574.0, 0.0, 0.0, 0.0]
Select 4 param, param name: zone_size, cpi:18146151448574.0
/usr/local/lib/python3.10/dist-packages/sklearn/gaussian_process/kernels.py:430: ConvergenceWarning: The optimal value found for dimension 0 of parameter k1_constant
value is close to the specified upper bound 0.1. Increasing the bound and calling fit again may find a better value.
warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/gaussian_process/kernels.py:430: ConvergenceWarning: The optimal value found for dimension 0 of parameter k2_length_s
cale is close to the specified upper bound 10.0. Increasing the bound and calling fit again may find a better value.
warnings.warn(
recommend params : gc_start_level      20.0
gc_slope      3.0
gc_sleep_time 10000.0
sect_size     512.0
zone_size     32.0
finish_threshold 0.0
ZBD_ABSTRACT_TYPE 1.0
RAID_LEVEL    1.0
Name: 0, dtype: float64
throughput list : [12726448, 15157667, 13340572, 13463982]
Destroyed /dev/nullb0
/home/lyt/Desktop/rocksd2/adjust_param/db_bench_script.py:136: RuntimeWarning: divide by zero encountered in scalar divide
print("Throughput from {} to {}, increase {}".format(pre_throughput, now_throughput, (now_throughput-pre_throughput)/pre_throughput))
Throughput from 0 to 13672167.25, increase inf

```

图 37: 调参测试过程

进行了配置推荐。记录这里的平均吞吐量是 13672167.25。

采用接下来的配置再进行测试：

-
- 1 SECT_SIZE_PARAM = 512
 - 2 ZONE_SIZE_PARAM = 32
-

最终结果如图 38 所示。

```

select 2 params
13517117452533.777 11740265749014.268 1776851703519.5098
Select 3 param, param name: sect_size, var:1776851703519.5098
param set: {} ((128.0,)): [0, 1, 2, 15, 16, 17, 18], (512.0,): [3, 4, 5, 19, 20, 21, 22], (1024.0,): [6, 7, 8], (2048.0,): [9, 10, 11], (4096.0,): [12, 13, 14]]
result: [0.0, 0.0, 0.0, 18146151448574.0, 0.0, 0.0, 0.0]
Select 4 param, param name: zone_size, cpi:18146151448574.0
/usr/local/lib/python3.10/dist-packages/sklearn/gaussian_process/kernels.py:430: ConvergenceWarning: The optimal value found for dimension 0 of parameter k1_constant
value is close to the specified upper bound 0.1. Increasing the bound and calling fit again may find a better value.
warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/gaussian_process/kernels.py:430: ConvergenceWarning: The optimal value found for dimension 0 of parameter k2_length_s
cale is close to the specified upper bound 10.0. Increasing the bound and calling fit again may find a better value.
warnings.warn(
recommend params : gc_start_level      20.0
gc_slope      3.0
gc_sleep_time 10000.0
sect_size     512.0
zone_size     32.0
finish_threshold 0.0
ZBD_ABSTRACT_TYPE 1.0
RAID_LEVEL    1.0
Name: 0, dtype: float64
throughput list : [14137453, 13673814, 17429716, 22502218]
Destroyed /dev/nullb0
/home/lyt/Desktop/rocksd2/adjust_param/db_bench_script.py:136: RuntimeWarning: divide by zero encountered in scalar divide
print("Throughput from {} to {}, increase {}".format(pre_throughput, now_throughput, (now_throughput-pre_throughput)/pre_throughput))
Throughput from 0 to 16935800.25, increase inf

```

图 38: 调参测试结果

平均吞吐量是 16935800.25。相比于上一次的配置参数，吞吐量提升了 23.87%。

将不同轮次的调参结果进行对比如图 39 所示，系统能够根据系统运行数据调整文件系统参数，从而逐步提高数据吞吐量。

后续还将尝试测试触发 gc 来测试 gc 的参数，以及尝试融合文件系统后对整体的 inode 以及其他参数进行自动调整。

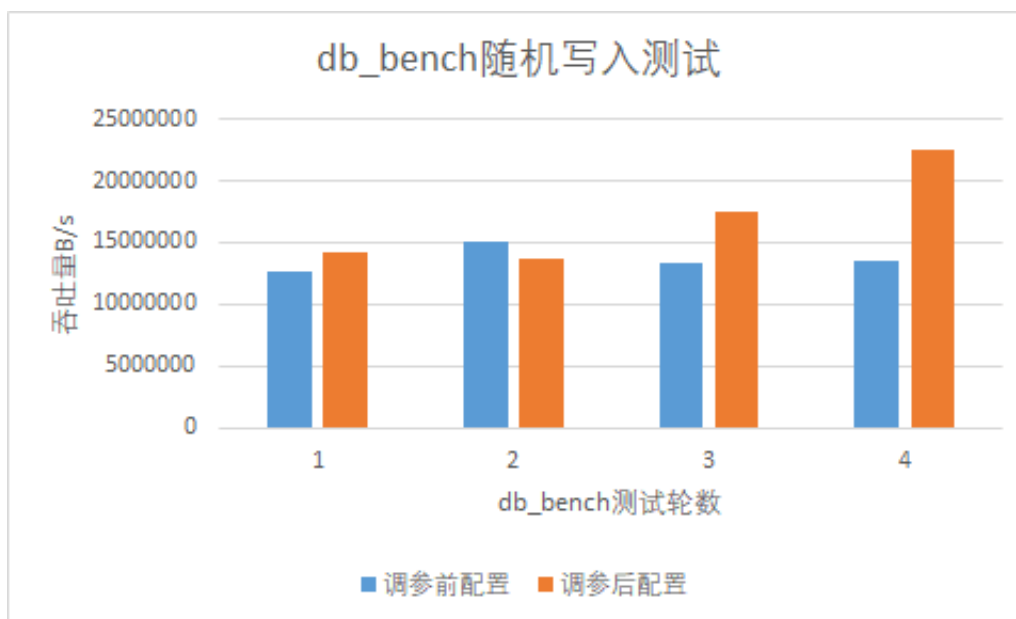


图 39: 调参测试结果对比

6 总结与展望

6.1 工作总结

我们提出了 AquaFS，一个以 ZenFS 为原型的，基于 Zoned Namespace SSD 的高性能、高灵活性、高效，而且更加智能的适用于 Flash 存储介质的文件系统。

AquaFS 采用模块化设计，不同模块各司其职，分别负责不同的功能，使得 AquaFS 的功能更加清晰，易于维护和扩展。

在当前初赛阶段，我们实现了 AquaFS 的一些核心功能，包括：

- 智能调参模块
 1. 参照论文^[7]，实现了高维度参数中重要参数筛选算法
 2. 针对当前轮获得的重要参数，实现了参数调整算法
- 智能动态分区 RAID
 1. 实现了全盘 RAID 模式中的 RAID0、RAID1、RAID-C
 2. 分区 RAID 动态地址映射管理实现
 3. 分区 RAID 动态 RAID 逻辑实现
 4. 分区 RAID 数据块错误处理和数据恢复
- IO 加速实现：实现基于 Direct IO + io_uring + C++ 协程的高性能 IO 加速

- ExtFS 辅助文件系统

总体完成进度约 50%。

工作量统计

这里统计了我们在初赛阶段的工作量，包括代码行数、代码提交数量等，如表5所示。统计的行数不包含非我们队伍成员编写的代码，比如 rocksdb、liburing4cpp 的原始代码。

表 5: 各个代码仓库的代码统计

代码仓库和说明	提交数量	添加行数	删除行数	总修改行数
aquafs-plugin : RocksDB 插件 AquaZFS	620	55,798	21,542	34,256
AquaZFS : 独立 ZenFS 移植	236	33,359	1261	32,098
adjust_param : 智能调参模块	50	1955	135	1820
docs : 在线 Markdown 文档网站	243	3290	694	2596
paper : 初赛技术报告	125	2895	252	2643
AquaFS : 整个系统通过 submodule 的整合	55	1904	126	1778
rocksdb : 配合开发修改参数	92	131	80	51
liburing4cpp : 测试 C++ 协程 io_uring	1	1	1	0
nvmevirt : 测试 ZNS 仿真	2	2	2	0

6.2 创新点和未来

在初赛阶段，我们总结 AquaFS 的已经实现或者部分实现了的创新点主要有以下几点：

- **更加智能：**ZenFS 许多逻辑都是固定的，而 AquaFS 通过智能调参、智能分类读写等方式提升其智能化水平。此外，AquaFS 还可以通过文件系统级冗余、文件系统读写分离、inode 缓存等方式提升其智能化水平。
- **更加安全：**ZenFS 在写入记录层面进行校验，但是对更常见的整块数据损坏无法有效应对，而 AquaFS 通过 RAID 等方式提升了针对 Zones 的数据安全性。
- **更加高效：**ZenFS 仅有 Direct IO 模式，而 AquaFS 支持 io_uring 等更高效的异步 IO 模式。

在概述章节中，我们总结了 AquaFS 相比于原型 ZenFS 的特点，其中还没有实现或者实现得还不够的创新点主要有以下几点：

- **适用场景更广泛：**ZenFS 仅适用于软件特殊适配后的 ZNS 设备，而 AquaFS 可以通过融合文件系统和通用文件接口等方式适用于更多的软硬件设备。
- **更加灵活：**ZenFS 由于其简单而参数较少，且都由上层软件适配调整，而 AquaFS 可以通过调整参数、融合文件系统等方式提升其场景适用性，为没有软件适配的应用场景提供支持。

为了完成这些创新点，我们将在后续的比赛阶段继续努力，完成 AquaFS 的全部功能，包括：

- 完善 AquaFS 的智能调参模块，针对垃圾回收、文件系统融合等算法进行调参
- 完善 Data Router，实现更加智能的数据路由，包括文件系统级冗余、文件系统读写分离、inode 缓存等
- 完善 AquaFS 的 IO 加速模块，实现 SPDK、xnvme 等用户态 NVMe 驱动的支持
- 完善 AquaFS 的 ExtFS 辅助文件系统，实现更加智能的文件系统融合
- 完善 AquaFS 的分区 RAID 功能，加强 RAID 分区分配逻辑的智能化
- 对更多的测试进行量化分析，分析并提升 AquaFS 的性能
- 争取获取到 ZNS SSD 设备，进行更加全面而更有说服力的测试和开发

虽然 AquaFS 在初赛阶段已经完成了部分工作，但是未来的工作仍然艰巨，还有许多问题需要解决：

- **ZNS SSD 设备：**由于暂未申请到 ZNS SSD 设备，我们难以进行更加全面而更有说服力的测试和开发，这是迫切需要解决的问题。
- **鲁棒性：**在 AquaFS 当前已经发挥作用的模块中，许多代码都是初步实现，运行过程可能不稳定或存在隐形 Bug，还需要进一步完善，提升其鲁棒性。
- **量化评估：**对许多工作需要做性能和读写放大上的量化评估，包括智能调参模块、IO 加速模块、分区 RAID 等，这些评估需要大量的测试和分析，需要大量的时间和精力。
- **使用场景：**当前 AquaFS 能够适用的场景还不够广泛，只在 RocksDB 和以 RocksDB 为数据后端的 MySQL 中做了测试，需要进一步扩展其使用场景。
- **性能提升：**当前 AquaFS 的 IO 路径还是相对较长，一些 IO 请求过程还是过于复杂，需要用 Kernel bypass 的方式提升其性能。

总而言之，AquaFS 在初赛阶段已经完成了部分工作，但是还有许多工作需要完成，我们将在后续的比赛阶段继续努力，完成 AquaFS 的全部功能，提升其性能和鲁棒性，扩展其使用场景，为更多的应用场景提供支持。

改进内容	完成进度	完成情况
智能调参模块	进度约 70%	<ul style="list-style-type: none"> ✓ 基于方差的重要参数选择方案 ✓ 基于高斯回归的调参方案 □ 将对垃圾回收参数进行进一步测试 □ 将对融合文件系统后的更多参数进行调优测试
基于 Zone 的智能动态分区 RAID	整体进度约 80%	<ul style="list-style-type: none"> ✓ 全盘 RAID 模式 RAID0、RAID1、RAID-C ✓ 分区 RAID 模式 Zone 映射和 RAID 逻辑分配 ✓ 数据完整性检测和恢复
异步 IO 优化	进度约 60%	<ul style="list-style-type: none"> ✓ 完成了 <code>io_uring</code> 异步读写优化 □ 将进一步研究基于 <code>spdk</code>、<code>xnvm</code> 等的用户态 <code>nvme</code> 驱动
融合通用文件系统	进度约 50%	<ul style="list-style-type: none"> ✓ 基于 FUSE 和 Rust 完成一个 Ext2 兼容文件系统 □ 将进一步研究与 ZenFS 的结合方式 □ 将进一步研究智能数据请求分类方法

附录 A 图表索引

A.1 图目录

插图

1	YOUBIFS 系统架构	7
2	MTD 设备和有 FTL 的设备的对比	10
3	Flash 原理与分类	11
4	K9F1G08U0F-5IB0 特性	12
5	写入时错误处理流程	13
6	Nand 块替换流程	14
7	Nand 写入寻址举例	14
8	ZenFS 架构	20
9	ZNS 多线程完全覆盖写入负载下吞吐量表现	21
10	单个 Zone 内的写逻辑	21
11	LSM-Tree 的操作逻辑	23
12	ZenFS 系统框图	24
13	ZoneFile 稀疏文件	30
14	ZenMetaLog Record	31
15	ZenMetaLog Roll	31
16	ZenMetaLog 格式	31
17	AquaFS 整体架构图	33
18	ZonedBlockDeviceBackend 继承类图	41
19	全盘 RAID 的数据排布	41
20	RAID0 数据排列逻辑	42
21	分区 RAID 的数据排布	44
22	RAID5 实现方式	48
23	IO 加速方案	49
24	ExtFS 基本布局	52
25	ExtFS inode 寻址方式	52
26	AquaTurnner 智能调参模块	53
27	磨损均衡	55
28	数据完整性测试 1	58
29	数据完整性测试 2	58
30	数据完整性测试 3	59

31	RAID 0 加速测试	60
32	测试流程图	61
33	数据恢复测试	61
34	RAID0 RAID5 数据恢复测试	62
35	perf 性能分析	62
36	调参测试数据	66
37	调参测试过程	67
38	调参测试结果	67
39	调参测试结果对比	68

A.2 表目录

表格

1	初赛进度情况	2
2	YOUBIFS 代码统计	9
3	数据完整性测试	57
4	RAID 0 加速测试, 5 次运行取平均	60
5	各个代码仓库的代码统计	69

附录 B 参考文献

参考文献

- [1] 基于 UBIFS 的更智能的文件系统[Z]. <https://gitlab.eduxiji.net/why/project788067-124640>.
- [2] CAI Y, MUTLU O, HARATSCH E F, et al. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation[C/OL]//2013 IEEE 31st International Conference on Computer Design (ICCD). Asheville, NC, USA: IEEE, 2013: 123-130 [2023-04-16]. <http://ieeexplore.ieee.org/document/6657034/>. DOI: 10.1109/ICCD.2013.6657034.
- [3] Ext4 documentation[Z]. <https://www.kernel.org/doc/Documentation/filesystems/ext4.txt>.
- [4] SEHGAL P, TARASOV V, ZADOK E. Evaluating Performance and Energy in File System Server Workloads[C]//USENIX Conference on File and Storage Technologies. 2010.
- [5] CAO Z, TARASOV V, TIWARI S, et al. Towards Better Understanding of Black-Box Auto-Tuning: A Comparative Analysis for Storage Systems[C]//USENIX ATC '18: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference. Boston, MA, USA: USENIX Association, 2018: 893-907.
- [6] LU K, LI G, WAN J, et al. ADSTS: Automatic Distributed Storage Tuning System Using Deep Reinforcement Learning[C/OL]//ICPP '22: Proceedings of the 51st International Conference on Parallel Processing. Bordeaux, France: Association for Computing Machinery, 2023. <https://doi.org/10.1145/3545008.3545012>. DOI: 10.1145/3545008.3545012.
- [7] MAHMUD T, GATLA O R, ZHANG D, et al. CONFD: Analyzing Configuration Dependencies of File Systems for Fun and Profit[J].,
- [8] VAN AKEN D, PAVLO A, GORDON G J, et al. Automatic Database Management System Tuning Through Large-Scale Machine Learning[C/OL]//SIGMOD '17: Proceedings of the 2017 ACM International Conference on Management of Data. Chicago, Illinois, USA: Association for Computing Machinery, 2017: 1009-1024. <https://doi.org/10.1145/3035918.3064029>. DOI: 10.1145/3035918.3064029.
- [9] BJØRLING M, AGHAYEV A, HOLMBERG H, et al. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs[J].,

- [10] LIANG Y, JI C, FU C, et al. ITRIM: I/O-Aware TRIM for Improving User Experience on Mobile Devices[J/OL]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2021, 40(9): 1782-1795 [2023-04-12]. <https://ieeexplore.ieee.org/document/9209165/>. DOI: 10.1109/TCAD.2020.3027656.
- [11] LU Y, SHU J, ZHANG J. Mitigating Synchronous I/O Overhead in File Systems on Open-Channel SSDs[J/OL]. ACM Transactions on Storage, 2019, 15(3): 1-25 [2023-04-12]. <https://dl.acm.org/doi/10.1145/3319369>. DOI: 10.1145/3319369.
- [12] CHEN P M, LEE E K, GIBSON G A, et al. RAID: high-performance, reliable secondary storage[J/OL]. ACM Computing Surveys, 1994, 26(2): 145-185 [2023-04-18]. <https://dl.acm.org/doi/10.1145/176979.176981>. DOI: 10.1145/176979.176981.
- [13] SCHROEDER B, LAGISETTY R, MERCHANT A. Flash Reliability in Production: The Expected and the Unexpected[J].,
- [14] BJØRLING M, AXBOE J, NELLANS D, et al. Linux block IO: introducing multi-queue SSD access on multi-core systems[C/OL]//Proceedings of the 6th International Systems and Storage Conference on - SYSTOR '13. Haifa, Israel: ACM Press, 2013: 1 [2023-04-15]. <http://dl.acm.org/citation.cfm?doid=2485732.2485740>. DOI: 10.1145/2485732.2485740.
- [15] KIM S H, JEONG S, SHIM J, et al. NVMeVirt: A Versatile Software-defined Virtual NVMe Device[J].,