# 改进 ZNS SSD 性能的策略综述

姚彤[1)]

[1)](华中科技大学 计算机科学与技术学院 武汉市 中国 430074)

**摘 要** ZNS SSD(Zone Namespaces SSD)是一种新型的 SSD，它将整个 SSD 空间划分为多个区域，并且每个区域内只允许顺序写入。ZNS SSD 能有效提升 SSD 的读写吞吐量，减少传统 SSD 垃圾回收带来的写放大，然而它对现有的存储分配策略提出了挑战，同时，由于 Zone 的巨大尺寸，基于 Zone 单元的垃圾回收同样会导致较大的延迟。基于日志结构合并树(LSM-Tree) 的键值存储由于顺序写入特性，具有高 I/O 性能，很适合部署在 ZNS SSD 上。然而，Compaction 操作导致的写放大会缩短 SSD 的使用寿命。本文综述了近几年以来关于 ZNS SSD 的几个经典的研究，总结了能够减小 ZNS SSD 架构中因 Compaction 操作带来的开销的策略。

关键词 Zone Namespace,Log-Sturctured Merge Tree,Data placement,Garbage collection,Lifetime

## A Survey of Strategies to Improve ZNS SSD Performance

Tong Yao[1)]

[1)](Huazhong University of Science and Technology,Department of Computer Science and Technology,Wuhan, China)

**Abstract** ZNS SSD (Zone Namespaces SSD) is a new type of SSD, which divides the entire SSD space into multiple zones, and only sequential writing is allowed in each zone. ZNS SSD can effectively improve the read and write throughput of SSD and reduce the write amplification caused by traditional SSD garbage collection. However, it poses a challenge to the existing storage allocation strategy. At the same time, due to the huge size of Zone, garbage collection based on Zone unit will also cause a large latency. Key-value stores based on log-structured merge-trees (LSM-Tree) are gaining attention, with high I/O performance due to sequential write characteristics. However, the write amplification caused by the compaction operation will shorten the service life of the SSD. This work summarizes several classic studies on ZNS SSD in recent years, and summarizes strategies that can reduce the overhead caused by compaction operations in the ZNS SSD architecture.

**Key words** Zone Namespace,Log-Sturctured Merge Tree,Data placement,Garbage collection,Lifetime

## 1 引言

几十年来，数据管理系统一直使用相同的块存储接口，同时假设底层存储设备具有相似的特性。近几年新出现的 ZNS SSD (Zone Namespaces SSD) 为创新提供了沃土，它释放了 SSD 的潜力，并具有优于硬盘和传统 SSD 的优势。传统的块接口将一系列逻辑块地址(Logical Block Address)映射到固定地址空间并允许随机读取和写入。ZNS SSD 将存储设备划分为一组大小相等的区域，只支持顺序写入。表面上这是对传统基于块的 SSD 的限制，实际上它使得主机能更好地控制 SSD 内部操作（例如数据放置和擦除）。ZNS SSD 的读取与传统 SSD 相同，支持顺序读取和随机读取。ZNS 要求主机以 Zone 为粒度执行重置(Reset)，从而消除设备上的垃圾回收(Garbage Collection)和设备端的 OP(Over Provisioning)空间。ZNS SSD 可以更好地进行数据放置。例如，将存储介质边界与 Zone 对齐，消除了在使用传统 SSD 块接口时的写入大小和块大小不匹配，Zone 的不变性极大地简化了版本管理、复制和重组等操作。

但消除随机写入和主机控制的垃圾回收为数据和元数据管理带来了新的挑战和机遇。ZNS 仅附加接口可能不适用于数据管理的所有方面。此外，ZNS SSD 对“活动”（可用于写入或追加）的 Zone 数量有限制，管理这些资源限制是主机的额外责任。在存储分配方面，ZNS SSD 必须考虑 Zone 内

只能顺序写入的特点，由于不能随机写入，Zone 内数据只能异地更新，这需要主机定期执行 Zone 的垃圾回收操作。垃圾回收操作涉及整个 Zone 的重置，Zone 中有效数据量大时，数据迁移成本高。因此，如何降低 Zone 垃圾回收的开销，避免垃圾回收对系统性能的稳定性的影响是 ZNS SSD 上数据管理必须考虑的问题。

Choi 等人[1]提出了一种基于 LSM 风格的垃圾回收策略，它从候选区域读取所有数据，识别冷数据，将它们合并到一个区域，同时将剩余数据合并到另一个区域。Bjørling 等人[2]提出了适用于 ZNS SSD 的小型文件系统 ZenFS，其中垃圾回收策略是基于 LSM 的 Lifetime 预测。Han 等人[3]提出了 ZNS+接口，其中的 IZC 策略可以有效减轻主机文件系统进行 Compaction 操作的负担。Lee 等人[4]在 ZenFS 的基础上提出了 CAZA(Compaction-Aware Zone Allocation)算法，改进了 ZenFS 在某些情况下失效的问题。Jung 等人[5]提出了一种为 ZNS SSD 量身定制的生命周期均衡压缩（LL-compaction）算法，使每个区域中的 SST 具有相似的生命周期。

# 2 Zone Namespace SSD

近几十年中，存储设备都将介质划分为固定大小的数据块以供主机使用。在一个块中储存的数据可以以任意顺序被读、写以及重写。当 SSD 问世时，同样遵循了这种读写规则，它专门设计了复杂的闪存翻译层(Flash Translation Layer)以提供与传统设备相似的块接口，但现阶段产生的性能瓶颈证明了这种规则并不能最大程度地利用闪存介质的价值。

基于闪存的 SSD 中允许数据直接写入一个空的页，但如果重写必须按块擦除，这种不对等的操作对象粒度带来了很大代价。FTL 需要负责地址映射、垃圾回收以及磨损均衡等功能，这意味着主机将数据放置权利完全交给 FTL。块接口不仅会增加 FTL 的压力，其中垃圾回收操作还需要额外的 OP 空间，额外的 DRAM 保存映射表，带来吞吐量降低，写放大，性能不可靠等问题。

在 ZNS 出现前，Stream SSDs 和 Open-Channel SSDs 都为减少块接口带来的代价做出了一定的努力。Stream SSDs 让主机标记数据生命周期，当写请求进入 SSD 时，设备根据标记将数据写入不同块中，这样能极大减少垃圾回收，但如果主机不能很好地分配标记，Stream SSDs 的表现就与块接口 SSDs 相似，而 Stream SSDs 要保证自己有足够的资源和效率来区分标记，就要消耗更多内存以及性能好的主控芯片，这样并不能很好降低 SSD 成本。Open-Channel SSDs 把大部分 FTL 的工作迁移到上层主机端，向主机展示芯片布局细节，主机可以根据 I/O 请求的发起方，将数据写到不同位置，实现不同应用的物理隔离。但 Open-Channel SSDs 需要主机侧软件层面的支持，或者重新增加一个软件层来匹配原来的软件堆栈。目前其软件生态并未完善，有些上层应用需要的改动比较大。基于上述问题，ZNS 被提出了，它既可以做到允许主机侧尽量自由放置数据，同时有标准的软件生态。

## 2.1 ZNS特性

Zoned Namespace 将一个 Namespace 的逻辑地址空间切分成多个 Zone。如图 1 所示，Zone 是 Namespace 内的一种固定大小的子区间，每个 Zone 都有一段 LBA 区间，这段区间只能顺序写，如果要覆盖写，必须进行一次擦除操作。ZNS 让上层应用来调度的 GC 操作，由于 Zone 内顺序写、无覆盖写的特性，消除了 GC 操作，减少了写放大。传统的 SSD 的 GC 时机和耗时对于应用是不可控的，ZNS 读、写、擦除和 GC 完全由主机和应用掌控，相比于传统 SSD 延迟表现更稳定。传统 SSD 会预留一部分 OP 空间用于垃圾回收 GC 和磨损均衡 WL，通常 OP 空间占整个 SSD 容量的 7-28%，对用户是不可见的。ZNS SSD 有更高效的 GC 机制，除了保留极少量 OP 空间用于替换坏块，并没有其他会大量消耗 NAND 空间的情况，并且 ZNS 固件本身所需要的运行空间远低于传统 SSD。ZNS 的架构特点，使得可以使用更少的 DRAM，以及更少的 OP 空间，因此相对传统 SSD 的成本有较大程度的降低。

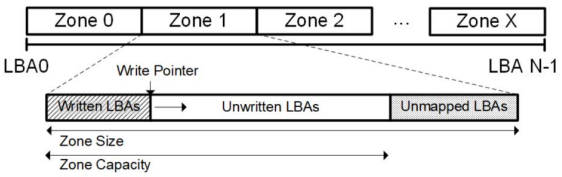


图 1 ZNS 结构

## 2.2 ZNS主控软件设计

### 2.2.1 主机端 FTL(HFTL)

HFTL 充当 ZNS SSD 的写入语义和执行随机写入和就地更新的应用程序之间的中介。HFTL 层有着类似于 SSD FTL 的职责，但 HFTL 层只负责地址

映射和相关的 GC。虽然它的职责比 SSD FTL 少，但 HFTL 必须管理其对 CPU 和 DRAM 资源的利用，因为它与主机应用程序共享这些资源。HFTL 可以更轻松地集成主机端信息并增强对数据放置和垃圾收集的控制，同时还向应用程序公开传统的块接口。

2.2.2 文件系统

为了支持更多的应用来访问 ZNS 设备，文件系统会作为一个更高层的存储接口，就像是 POSIX API 一样为应用提供文件语义。现有的文件系统包括 F2FS, BTRFS 以及 ZFS 等都能够支持顺序写模式，但是这些文件系统的元数据的更新并非是顺序写，比如 Log-Structured Write (文件系统的 WAL)，Superblock 的更新等,都不是顺序写。所以想要为 ZNS 做一个更上层的应用，并且能获取 ZNS 内部的各个 Zone 的状态，使用现有的这些文件系统会带来过于复杂，且改造成本太高的问题，虽然在前期的 Zone Storage 演进的过程中也在 F2FS 这种文件系统上做了适配，但也仅限于在 ZAC/ZBC 这样的设备中使用，不适用于 ZNS，所以后续 ZNS 单独设计了适配于 ZNS 的文件系统来为应用提供文件语义。

2.2.3 端到端数据放置

传统的磁盘 I/O 栈需要通过内核文件系统，通用块层，I/O 调度层，块设备驱动层这样的一系列 I/O 子系统才能达到磁盘的主控。这种漫长的链路在无形中拖慢数据的存储效率，间接降低磁盘吞吐，增加了请求的延时。然而 ZNS 内部的设计本身能够极大的降低 SSD 内部的写放大，提升写吞吐量，同时由于 Erase Block 的粒度大，减少了对 LBA 的需求，少了很多的指令和对 LBA 的管理操作。ZNS 做了一个非常大胆的挑战，开发了支持文件系统语义的端到端的访问形态的文件系统 ZenFS，数据的存储绕开像原本块接口一样需要庞大的 I/O 栈，直接与 ZNS SSD 进行交互。

有很多应用广泛的系统一直在 端到端的数据存储上进行探索，而且这一些系统能够支持顺序写模式，同时也支持新提出的 ZNS，包括但不限于：LSM Tree 存储引擎 RocksDB, 基于 Cache 的存储系统 CacheLib,对象存储系统 Ceph 的 Seastore。

# 3 Log Structured Merge Tree(LSM Tree)结构

LSM 树是当前主流 KV 存储系统中被广泛应用的存储引擎，其结构如图 2 所示。LSM 树由两个或以上的存储结构组成，一个存储结构常驻内存中，具体可以是任何方便健值查找的数据结构，比如红黑树、Map，甚至可以是跳表。另外一个存储结构常驻在硬盘中，容量逐级增加。

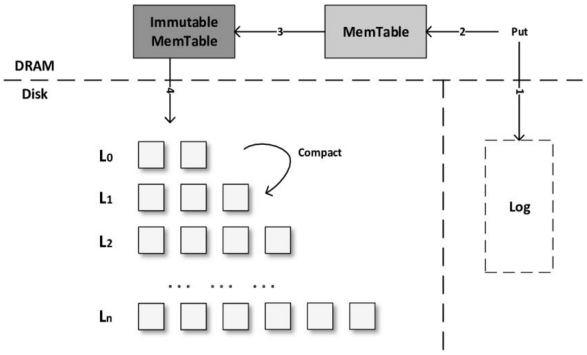


图 2 LSM Tree 结构

LSM 树写性能极高的原理，简单来说就是放弃部分读能力，尽量减少随机写的次数。LSM 树会将所有的数据插入、修改、删除等操作保存在内存中，当此类操作达到一定得数据量后，再批量地写入磁盘当中。而在写磁盘时，会和以前的数据做合并。在合并过程中，并不会像 B+树一样，在原数据的位置上修改，而是直接插入新的数据，从而避免了随机写。

Memtable 是一个在内存中进行数据组织与维护的结构。Memtable 中，所有的数据按用户定义的排序方法排序之后按序存储，等到其存储内容的容量达到阈值时（默认为 4MB），便将其转换成一个不可修改的 Memtable，与此同时创建一个新的 Memtable，供用户继续进行读写操作。Memtable 底层使用了一种跳表数据结构，这种数据结构效率可以比拟二叉查找树，绝大多数操作的时间复杂度为 $O(\log^n)$。

Memtable 的容量到达阈值时，便会转换成一个不可修改的 Memtable，也称为 Immutable Memtable。这两者的结构定义完全一样，区别是 Immutable Memtable 是只读的。当一个 Immutable Memtable 被创建时，后台压缩进程便会将利用其中的内容，创建一个 SSTable，持久化到磁盘文件中。

假设写入到内存的数据还未来得及持久化，系统进程发生了异常，或是宿主机器发生了宕机，会造成用户的写入发生丢失。因此写内存之前会首先将所有的写操作写到日志文件中，也就是 Log 文件。当异常情况发生时，均可以通过日志文件进行恢复。每次日志的写操作都是一次顺序写，因此写效率高，整体写入性能较好。此外，用户写操作的原子性同样通过日志来实现。

虽然采用了先写内存的方式来提高写入效率，但是内存中数据不可能无限增长，且日志中记录的写入操作过多，会导致异常发生时，恢复时间过长。因此内存中的数据达到一定容量，就需要将数据持久化到磁盘中。除了某些元数据文件，LSM 树架构下的数据主要都是通过 SSTable 来进行存储。虽然在内存中，所有的数据都是按序排列的，但是当多个 Memetable 数据持久化到磁盘后，对应的不同的 SSTable 之间是存在交集的，在读操作时，需要对所有的 SSTable 文件进行遍历，严重影响了读取效率。因此后台会定期整合这些 SSTable 文件，该过程也称为 Compaction。随着 Compaction 的进行，SSTable 文件在逻辑上被分成若干层。

Compaction 操作是十分关键的，否则 SSTable 数量会不断膨胀。不同的 Compaction 策略围绕下面三个概念做出权衡和取舍：

1）读放大:读取数据时实际读取的数据量大于真正的数据量。例如在 LSM 树中需要先在 MemTable 查看当前 key 是否存在，不存在继续从 SSTable 中寻找。

2）写放大:写入数据时实际写入的数据量大于真正的数据量。例如在 LSM 树中写入时可能触发 Compaction 操作，导致实际写入的数据量远大于该 key 的数据量。

3）空间放大:数据实际占用的磁盘空间比数据的真正大小更多。对于一个 key 来说，只有最新的那条记录是有效的，而之前的记录都是可以被清理回收的。

利用 LSM-Tree 的键值存储系统，由于顺序写入，异地更新的特性可以在基于 NAND 闪存的固态驱动器 SSD 上获得 I/O 性能。然而，在 Compaction 过程中会发生大量写入，这会降低 SSD 的性能和寿命。此外，在传统块接口 SSD 中，闪存转换层的垃圾回收进一步增加了写入放大。而 ZNS SSD 使用一种新的基于 Zone 的接口，将 NAND 闪存区域划分为一定大小的区域，并允许主机直接管理这些区域。因此，ZNS SSD 可以最大限度地减少垃圾回收开销和写入放大，因为数据放置是由主机直接执行的。目前，学者门正在研究如何以最小化使用 ZNS SSD 的 LSM Tree 的写放大。

# 4 ZNS 中垃圾回收策略的研究

## 4.1 LSM_ZGC策略

ZNS SSD 是一把双刃剑，它将不同负载的数据放入不同 Zone 中以改进性能减少写放大，缺点是主机需要直接负责 SSD 的垃圾回收并且受顺序写的约束。一种简单的方法是应用传统方案，例如 LFS（日志结构文件系统）使用的段清理或 FTL 使用的垃圾回收策略。唯一的区别是它基于区域单元而不是段单元应用垃圾收集。然而分析表明，由于区域的大小（例如 0.5GB 或 1GB）远大于段的大小（2 MB 或 4 MB），这种简单的方法会导致较长的垃圾回收延迟。

为了克服这种困难，论文[1]提出了一种新的垃圾回收策略 LSM_ZGC。作者发现传统的段概念对于回收 ZNS SSD 中的 Zone 仍然有价值。具体来说，LSM_ZGC 将一个 Zone 划分为多个 Segment，分别管理它们的有效性位图和利用率等信息。它以 LSM 风格进行垃圾回收，从候选区域读取所有数据，识别冷数据，将它们合并到一个区域，同时将剩余数据合并到另一个区域。这种方法有几个优点：基于段粒度而不是区域粒度的垃圾回收成为热/冷隔离的有效基础，并使工作可以以流水线方式进行；读取段中的所有数据（包括有效和无效数据）可以通过利用区域中的内部并行性来减少垃圾回收开销；根据数据的热/冷特性将数据合并到不同的区域可以增加找到利用率较低的区域的机会。

ZNS SSD 中区域垃圾收集的一种简单方法是选择利用率最低的候选区域，然后它从所选区域读取有效块并将它们写入新区域,最后，发出所选区域的重置命令,将此方案称为 Basic_ZGC。Basic_ZGC 和 LSM_ZGC 策略的对比如图 3 所示。LSM ZGC 策略有三个变化。首先，它以段为单位回收一个区域。这种方法可以很容易地将热数据和冷数据隔离到不同的区域。此外，它允许使用细粒度的段单元执行区域垃圾回收，在该单元中读取、合并和写入段可以以流水线方式完成。

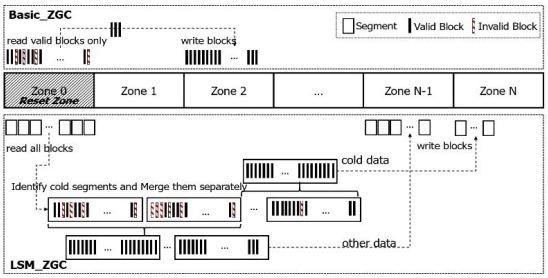


图 3 Basic_ZGC 和 LSM_ZGC 策略

其次，在垃圾回收期间，它不仅读取有效块，还读取 128KB I/O 大小的无效块，而 Basic_ ZGC 只读取有效块。为硬盘设计的原始 LFS 会像本方案一样读取所有数据，然而大多数为 SSD 设计的文件系统和 FTL 仅读取有效数据，因为闪存中没有寻道开销。作者仔细地论证了读取所有块是 ZNS SSD 中的一个可行选项。实际上，作者在设计 LSM_ZGC 时考虑了利用候选段。具体来说，当一个段中的有效块数小于 16 时，LSM_ZGC 只读取有效块。否则，它会读取所有块，因为 16 个 128KB 大小的请求可以覆盖整个 2MB 段数据。第三个区别是 LSM _ZGC 尝试识别冷数据并将它们合并到一个单独的区域中。为此定义一个区域的四种状态，即 C0_Zone、C1C_Zone、C1H_Zone 和 C2_Zone，如图 4 所示。新到达的数据被顺序写入状态为 C0_Zone 的区域，被删除的数据去往图中显示的状态之外。

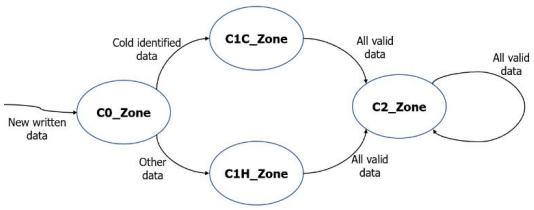


图 4 区域状态转换图

现在假设 LSM_ZGC 选择了一个状态为 C0_Zone 的候选区域。它读取区域中的所有段并尝试识别冷数据段。将利用率高于阈值（称为 thresholdcold）的段定义为冷数据段。这个决策是基于作者的观察得出的，即具有相似生命周期的数据有出强烈的空间局部性。例如，在键值存储中，每个级别显示不同的生命周期，同一级别的 SSTables 以批处理的方式写入。被标记为冷的段中的有效块被合并并写入状态为 C1C_Zone 的区域。其他段的有效块被合并并写入另一个状态为 C1H_Zone 的区域。当候选区域是 C1C_Zone 或 C1H _Zone 时，LSM_ZGC 读取所有段并将所有有效块视为冷块。这是因为这些有效块在两次垃圾回收后仍然存在。它们被合并写入状态为 C2_Zone 的区域。可以进一步扩展，例如写入 C3_Zone 等，但是在本研究中，作者将在此处停止并将从 C2_Zone 幸存下来的有效块写入另一个 C2_Zone。作者希望这种机制能够将冷数据与其他数据隔离开来，从而增加在垃圾回收期间找到利用率较低的候选区域的机会。

实验结果表明，LSM_ZGC 将 Basic_ZGC 的性能平均提高了 1.9 倍。

## 4.2 ZenFS

论文[2]中，作者开发了支持文件系统语义的端到端访问形态的 ZenFS，数据的存储绕开原来块接口使用的庞大的 I/O 栈，直接与 ZNS SSD 进行交互。

### 4.2.1 ZenFS 架构

ZenFS 实现了一个最小的磁盘文件系统，并使用 RocksDB 的文件包装器 API 将其集成。它谨慎地将数据放入区域，同时尊重它们的访问限制，并在写入时与设备端区域元数据协作（例如写指针），从而降低与持久性相关的复杂性。ZenFS 的架构如图 5 所示，下面对架构中的组成部分进行描述。

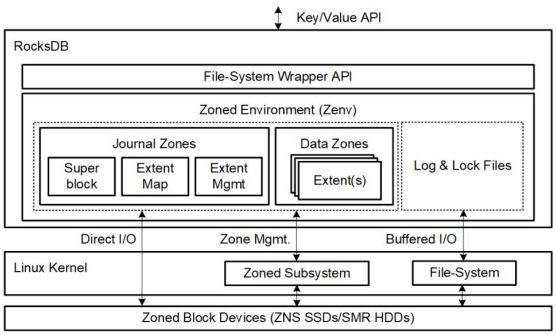


图 5 ZenFS 架构

1）Log&Lock Files。ZenFS 定义了两种类型的区域：日志区域和数据区域。日志区域用于恢复文件系统的状态，并维护超级块数据结构，以及 WAL 和数据文件到区域的映射，而数据区域存储文件内容。

2）Extent。RocksDB 的数据文件被映射到一组 Extent 集合中。一个 Extent 是一个可变大小的、块对齐的、连续的 LBA 地址，Extent 会和特定标识符一起被顺序写入。每个 Zone 可以存储多个 Extent，

但 Extent 不跨区。Extent 分配和解除分配事件记录在内存数据结构中，并在文件关闭或 RocksDB 通过 fsync 调用请求持久化数据时写入日志。内存中的数据结构跟踪 Extent 到 Zone 的映射，一旦删除了 Zone 中分配了 Extent 的所有文件，就可以重置和重新使用 Zone。

3）Super block。超级块数据结构是磁盘初始化和恢复 ZenFS 状态时的初始入口点。超级块包含当前实例的唯一标识符、魔数和用户选项。超级块中的唯一标识符(UUID)允许用户在磁盘的盘符重启或者外部插拔发生变化的情况下也可以识别文件系统。

4）Journal。日志的职责是维护超级块、WAL 和数据文件到通过 Extent 到 Zone 映射。日志状态存储在专用日志区域中，并且位于设备的两个不会离线区域中。在任何时间点，其中一个区域被选为活动日志区域，并且是持续更新日志状态的区域。日志区域有一个标头，存储在特定区域的开头。标头存储序列号（每次初始化新日志区域时递增）、超级块数据结构和当前日志状态的快照。要从磁盘恢复日志状态，需要三个步骤：必须读取两个日志区域中每一个的第一个 LBA 以确定每个日志区域的序列号，其中具有最高值的日志区域是当前活动区域；读取活动区的完整标头并初始化初始超级块和日志状态；最后日志更新应用于标头的日志快照。

#### 4.2.2 数据区选择算法

ZenFS 采用尽力而为算法来选择最佳区域存储 RocksDB 数据文件。RocksDB 通过在写入文件之前为文件设置 writelifetime 提示来区分 WAL 和 SST 文件，不同级别 SST 和 WAL 有不同的 writelifetime 值。第一次写入文件时，会分配一个数据区用于存储。ZenFS 首先尝试根据文件的生命周期和区域中存储的数据的最大生命周期找到一个区域。仅当文件的生命周期小于区域中存储的最旧数据时，匹配才有效，以避免延长区域中数据的生命周期，减少写放大。如果找到多个匹配项，则使用最接近的匹配项。如果未找到匹配项，则会分配一个空区域。如果文件填满了已分配区域的剩余容量，则使用相同的算法分配另一个区域。writelifetime 值会提供给任意 RocksDB 存储后端，因此也会传递给其他兼容的文件系统，并且可以与支持流的 SSD 一起使用。通过使用 ZenFS 区域选择算法和用户定义的可写容量限制，未使用的区域空间或空间放大率保持在 10%左右。

#### 4.2.3 活动区域限制

ZenFS 必须遵守 ZNS 指定的活动区域限制。要运行 ZenFS，至少需要三个活动区域，分别分配给日志、WAL 和压缩进程。为了提高性能，用户可以控制并发压缩的数量。实验表明，通过限制并发压缩的数量，RocksDB 可以使用少至 6 个写入性能受限的活动区域，而超过 12 个活动区域不会增加任何显着的性能优势。

#### 4.2.4 实验结果

实验对比了工作在块接口上的 xfs 和 f2fs, ZNS 上的 f2fs 以及 ZenFS 这四种场景。工作负载分别为 fillrandom 和 overwrite。fill random 的性能差异并不大，当进行 overwrite 时有大量 compaction 进行的时候，ZenFS 的调度策略就很有优势了，写放大在原本 Compaction 策略不变的情况下降到了最低，写吞吐提升了一倍多。

### 4.3 IZC策略

在论文[3]中，作者指出主机端 GC 要比设备端 GC 付出更大的代价，主机在进行数据拷贝时需要处理 I/O 请求，主机到设备端的数据传输，为读取到的数据分配内存，因此不应该单纯地将工作甩给主机端，设备端也要分担一部分工作。由此，作者提出了 ZNS+ 接口，使用 IZC(Internal Zone Compaction)策略，主机可以将数据拷贝操作交由 SSD，从而加速 Compaction，减轻 LFS(Log-Structured File System)的压力。

#### 4.3.1 LFS 传统 Compaction 过程

由块接口 SSD 向 ZNS SSD 转变时，随机写的文件系统要转换为顺序写，例如 LFS。LFS 的一个段可以分配到一个或多个 Zone 内。LFS 要定期进行段的 Cpmpaction，传统 Compaction 过程有四个任务：

1）选取受害段。即选择一个有着最小 Compaction 开销的段。

2）目的块分配。该步骤从目的段中分配连续地址空间。

3）有效数据拷贝。由主机控制，将受害段中全部的有效数据迁移到目的段中，这一步会带来大量主机与设备间的数据传输，同时存储芯片中间会产生大量间隙，不能被完全利用。数据拷贝包含度阶段和写阶段。

4）元数据更新。LFS 需要写若干元数据块来

反映数据的变化。

其工作过程如图 6 所示。

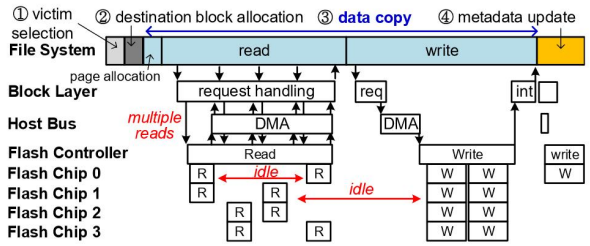


图 6 LFS 传统 Compaction 过程

4.3.2 IZC 策略

IZC 工作过程如图 7 所示。可以看到第 3 步由数据拷贝变为了数据卸载。设备端比主机端更能快速地进行数据拷贝，因为 SSD 可以高效管理并行的闪存芯片，从而消除主机到设备端的数据传输。它发送 zone_compaction 命令来传输数据拷贝信息（即源和目的 LBA）。由于目标数据不需要加载到主机缓存中，所以不需要相应的页面缓存分配。SSD 内部控制器可以高效地安排多个读写操作，同时最大限度地提高闪存芯片的利用率。因此，段压缩延迟可以显著降低。此外，存储设备内部的数据拷贝可以利用 copyback 操作。

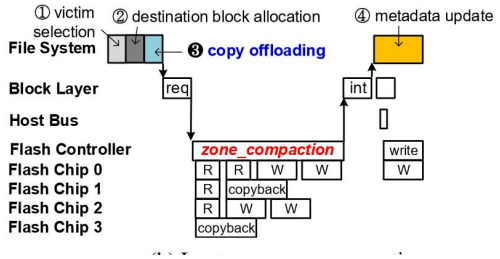


图 7 IZC 过程

ZNS+存储系统中 Compaction 的过程如下：

1) 缓存页面处理。第一步是检查受害者段中每个有效块的相应页面是否缓存在主机 DRAM 上。如果缓存页面是脏的，它必须被写入目标段并且从 IZC 操作中排除。如果缓存页面是干净的，它可以通过写请求写入或通过 zone_compaction 在内部复制。如果它是通过写入请求从主机传输的，则可以跳过相应的闪存读取操作，因为它已经被缓存，也不需要页面分配。但是这会带来数据传输和写入请求处理开销。通过比较闪存读取成本和数据传输成本，我们可以选择合适的方案来重新定位缓存块。一般来说，基于 TLC 或 QLC 技术的高密度闪存比主机到设备的访问延迟更高。

2) 拷贝数据的卸载。为了将数据复制操作卸载到 ZNS+ SSD，生成了 zone_compaction(source LBAs, destination LBAs)命令。ZNS+ SSD 将第 i 个源 LBA 中的数据复制到第 i 个目的 LBA 中。当在 F2FS 启用线程日志记录时，段压缩可以选择一个 TL_opened 段作为目的地，目的 LBA 可以是不连续的。

3) IZC 操作。最后，ZNS+ SSD 处理 zone_compaction 命令。它通过检查源和目的 LBA 所在芯片来识别可通过回写操作复制的可回写块。仅当必须复制其中的所有块时，块才可回复制。SSD 固件为不可回拷贝的块发出闪存读写操作。

zone_compaction 命令的处理是异步的。主机发出的 compaction 命令会进入到命令队列中，主机不会等待命令完成。因此，在预先发出的压缩命令完成之前，主机可以立即发出后续 I/O 请求。异步处理可以通过消除后续请求的等待时间来提高性能。由于 LFS 的检查点方案，异步命令处理不会损害文件系统的一致性。

4.3.3 实验结果

实验对比了 ZNS 和 IZC 在各种基准测试中的平均段压缩延迟。ZNS 的压缩时间分为四个阶段：初始化、读取、写入和检查点；IZC 的分为三个阶段：初始化、IZC 和检查点。初始化阶段读取与受害者段相关的所有文件的几个元数据块。这些元数据一般都在页面缓存中，所以初始化阶段很短。与 ZNS 相比，IZC 通过消除主机复制开销并利用回写操作将 Zone 压缩时间减少了约 28.251.7%。

## 4.4 CAZA策略

在论文[4]中，作者对 ZenFS 做了仔细的研究，发现其最致命的限制是缺乏对 SSTables 会在 LSM 树中失效的设计考虑。因此，作者在 ZenFS 数据区域选择算法的基础上提出了 CAZA 算法，该算法是基于对压缩过程如何选择、合并和使 LSM 树中的 SSTables 无效的密切观察而设计的。CAZA 的设计考虑了 LSM-tree 的压缩过程，通过将位于 LSM-tree 中不同级别的重叠键范围的 SSTables 合并到同一区域中，并在区域清理期间将它们一起失效，从而最大限度地提高 ZenFS 的区域清理效率。

作者通过观察 LSM-tree 的数据结构，描述了 SSTables 如何形成 LSM-tree，有足够的线索来推断 SSTables 的可能组合，这些组合将用作压缩输入并

称为无效数据。例如，要在第 L 级放置的一些新 SSTable，通过 LSM-tree 的数据结构，可以推断出第 L-1 级会触发压缩操作，有 SSTable 将被合并；由于 L 级与 L+1 级的 SSTable 有键值重叠，L+1 级的一些 SSTable 将会以同样的方式被压缩。当压缩输入数据包括 SSTables 时，根据 LSM-tree 的数据结构，可以很容易地从相邻的较低级别中选择所有具有与 S 重叠键范围的 SSTable。

基于以上观察，CAZA 通过考虑在 LSM-tree 中如何选择压缩输入，将区域分配给在合并相邻级别具有重叠键范围 d 的 SSTable 后新创建的 SSTable。因此，CAZA 可以解决 ZenFS 的两个局限性，即 SSTables 在不同区域中具有重叠的键范围，以及 SSTables 由于不同的生命周期提示级别而在不同区域之间无效。

在图 8 中，所有在级别 1 和级别 2 上具有重叠键范围的 SSTable（A、B、C）在压缩之前都放置在区域 0 中。压缩后，SSTable A、B、C 在 Zone 0 中一起失效。之后，当触发 Zone 回收时，它会选择区域中无效数据比例最高的区域 0 作为清理的目标区域。可以避免有效数据复制，因为该区域中的所有 SSTables 都已被先前的压缩失效。

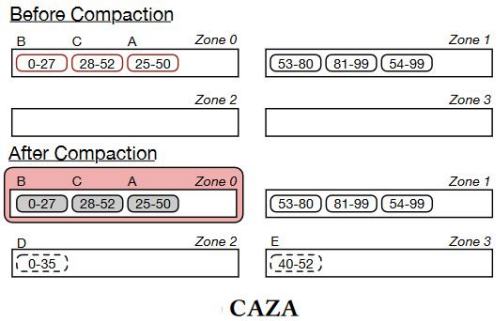


图 8 CAZA 算法过程

CAZA 的算法流程如下：

1）从 L 级开始，即压缩触发的级别。 CAZA 通过在级别 L+1 中搜索与 SSTables 的键范围重叠的 SSTable，构建了一组 SSTable($S_{overlap}$)；

2）CAZA 构建所有包含 $S_{overlap}$ 的区域集合 Z；

3）集合 Z 按照包含的 SSTables 的数量降序排序；

4）CAZA 按顺序从 Z 中分配区域。

CAZA 可能无法找到满足步骤 1-4 中条件的区域，例如以下两种可能触发分区清理的场景：

场景 1：一些 SSTables 可能有全新的键范围。在这种情况下，CAZA 分配一个空区域，以便该区域可以包含新的键范围；

场景 2：CAZA 可能没有足够的空间在具有重叠键范围的 SSTable 所在的所有区域中存储新的 SSTable。在这种情况下，CAZA 将分配一个新的空区域而不是包含不相交键范围的区域。否则，具有非重叠键范围的 SSTable 将位于同一区域中，并且不太可能被压缩在一起。

有意为场景 1-2 分配新的空白区域，这是为了让当前的 SSTable 有机会与未来的 SSTable 一起压缩，这些 SSTable 将被放置在同一个新的空白区域中。为了限制区域回收的成本，CAZA 仅尝试重置没有有效数据的区域，其余的区域清理过程在回退场景中介绍。最后，如果区域清理失败，CAZA 将退回到以下场景：

场景 1：CAZA 在同一级别搜索具有最接近键范围的 SSTable，然后分配包含 SSTable 的区域。寻找最接近的键范围为未来的上层（或下层）SSTable 提供了桥接附近键范围的机会。例如，键范围为(10-20)和(25-35)的 SSTable 可以被上层(15-30)的 SSTable 桥接；

场景 2：CAZA 回退到 ZenFS。

实验结果显示 CAZA 的写放大一直比 ZenFS 低，并且差距越来越大，在要求回收 25%的区域时减少了 2 倍的数据拷贝。

### 4.5 LL-compaction

在论文[5]中，同样为了解决 LSM-tree 在 ZNS 上的写放大问题，作者提出了 LL-compaction(lifetimeleveling compaction)算法，使每个区域中的 SST 具有相似的生命周期。

基于实验，作者观察到两种类型的的 SSTable 会对性能产生影响，分别是长寿命 SST 和短寿命 SST。长寿命 SST 是指一直在 Zone 中，未被压缩的 SST，当 Zone 中其他 SST 均参与压缩并被删除时，长寿命 SST 也要被拷贝从而造成写放大；短寿命 SST 是指刚被写入又要被压缩的 SST。

LL-compaction 算法如图 9 所示，它基于以下原则：

1）它为每级分配专用区域，因为不同级别 SST 具有不同的生命周期，有压缩指针(Compaction Pointer，CP)指向要压缩的 SST；

2）为了避免长寿命 SST 出现，每次压缩都必须涉及下一及的位于当前 $CP_i$ 和下一级 Next $CP_i$ 之

间所有的 SST；

3）短寿命 SST 与正常 SST 分离，并且数量最小化。在中，由 SST1 触发的第一次压缩在 Li+1 级创建了四个新的 SST。最后两个 SST，SST10 和 SST11，被 Next $CP_i$ 隔开。即使 SST10 的容量小于 SST 的最大容量，该 SST 也会在此处结束，并在后面创建另一个 SST。后者会参与 SST2 触发的下一次压缩。如果不这样拆分 SST，SST10 的键范围将包含在下一个压缩窗口中。由于 SST11 是暂时的 SST，因此它被写入 T-Zone 中。然后可以将短寿命 SST 从 Zone10 中的正常 SST 中分离出来。T-Zone 只有暂时的 SST，因此很容易回收。

4）在 $L_{i+1}$ 级中创建 SST 时，创建的 SST 的键范围不能跨越中间的 $CP_{i+1}$。如果不能保证这一点，$CP_{i+1}$ 可以更改为指向 SST 的起始位置的指针，因为无法从 SST 的中间位置开始执行压缩。如果 $CP_{i+1}$ 被 $L_i$-$L_{i+1}$ 的压缩改变，那么在 $L_{i+1}$ 和 $L_{i+2}$ 压缩中 SST 选择的顺序可以被打破。在图 9 中，$CP_{i+1}$ 指向 SST5。由 SST2 触发的第二次压缩在 Li+1 中创建了三个 SST。第一个 SST（SST12）在 $CP_{i+1}$ 之前结束，后面的 SST（SST13）的键范围从 $CP_{i+1}$ 开始。这样就不需要改变 $CP_{i+1}$。

在 LL-compaction 下，由于其 SST 拆分策略，可以创建许多小的 SST 文件。根据实验评估，文件数量在 LL-compaction 中增加了 9%。但是，由于 SST 文件数量增加而产生的额外索引开销可以忽略不计。在执行 Get 操作时，元数据搜索开销仅占总执行时间的 0.6%，大部分开销来自 SSD 访问时间。因此，执行时间增加仅为 0.005%。

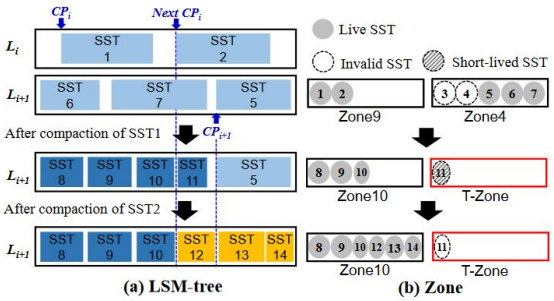


图 9 LL-compaction 过程

## 参 考 文 献

[1] Choi G, Lee K, Oh M, et al. A New {LSM-style} Garbage Collection Scheme for {ZNS}{SSDs}[C]//12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20). 2020.

[2] Bjørling M, Aghayev A, Holmberg H, et al. {ZNS}: Avoiding the Block Interface Tax for Flash-based {SSDs}[C]//2021 USENIX Annual Technical Conference (USENIX ATC 21). 2021: 689-703.

[3] Han K, Gwak H, Shin D, et al. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction[C]//15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21). 2021: 147-162.

[4] Lee H R, Lee C G, Lee S, et al. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs[C]//Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems. 2022: 93-99.

[5] Jung J, Shin D. Lifetime-leveling LSM-tree compaction for ZNS SSD[C]//Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems. 2022: 100-105.