SMRstore: A Storage Engine for Cloud Object Storage on HM-SMR Drives

Su Zhou, Erci Xu, Hao Wu, Yu Du, Jiacheng Cui, Wanyu Fu, Chang Liu, Yingni Wang, Wenbo Wang, Shouqu Sun, Xianfei Wang, Bo Feng, Biyun Zhu, Xin Tong, Weikang Kong, Linyan Liu, Zhongjie Wu, Jinbo Wu, Qingchao Luo, and Jiesheng Wu, *Alibaba Group*

https://www.usenix.org/conference/fast23/presentation/zhou

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

SMRSTORE: A Storage Engine for Cloud Object Storage on HM-SMR Drives

Su Zhou, Erci Xu*, Hao Wu, Yu Du, Jiacheng Cui, Wanyu Fu, Chang Liu, Yingni Wang, Wenbo Wang, Shouqu Sun, Xianfei Wang, Bo Feng, Biyun Zhu, Xin Tong, Weikang Kong, Linyan Liu, Zhongjie Wu, Jinbo Wu, Qingchao Luo, Jiesheng Wu

Alibaba Group

Abstract

Cloud object storage vendors are always in pursuit of better cost efficiency. Emerging Shingled Magnetic Recording (SMR) drives are becoming economically favorable in archival storage systems due to significantly improved areal density. However, for standard-class object storage, previous studies and our preliminary exploration revealed that the existing SMR drive solutions can experience severe throughput variations due to garbage collection (GC).

In this paper, we introduce SMRSTORE, an SMR-based storage engine for standard-class object storage without compromising performance or durability. The key features of SMRSTORE include directly implementing chunk store interfaces over SMR drives, using a complete log-structured design, and applying guided data placement to reduce GC for consistent performance. The evaluation shows that SMRSTORE delivers comparable performance as Ext4 on the Conventional Magnetic Recording (CMR) drives, and can be up to 2.16x faster than F2FS on SMR drives. By switching to SMR drives, we have decreased the total cost by up to 15% and provided performance on par with the prior system for customers. Currently, we have deployed SMRSTORE in standard-class Alibaba Cloud Object Storage Service (OSS) to store hundreds of PBs of data. We plan to use SMR drives for all classes of OSS in the near future.

1 Introduction

Object storage is a "killer app" in the cloud era. Users can use the service to persist and retrieve objects with high scalability, elasticity and reliability. Typical usage scenarios of object storage include Binary Large OBjects (BLOBs) storage [10,23], datalake [2] and cloud archive [1]. Object storage systems usually employ a large fleet of HDDs. Therefore, a key challenge of building a competitive cloud object storage is the cost efficiency.

Emerging Shingled Magnetic Recording (SMR) drives [5] are economically attractive [29] but they may not serve as a simple drop-in replacement for traditional CMR drives [12]. SMR drives, via overlapping tracks, have a higher areal density [14] (i.e., 25% more than CMR drives) and hence the better cost efficiency. However, shingling tracks has a byproduct—not allowing random writes [8,15]. This characteristic in return may require the upper-level software stack, such as storage engines, to make the corresponding adaptions.

A possible direction is to use Host Managed SMR (HM-SMR) drives where the host OS manages the I/Os and communicates with HM-SMR drives via the Zoned Block Device (ZBD) subsystem [6,7]. There are mainly three types of approaches in designing HM-SMR-based storage systems. First, Linux kernel can expose HM-SMR drives as standard block devices by employing a shingled translation layer (STL), such as dm-zoned [21]. Second, file systems with a log-structured [28] or copy-on-write design, can directly support HM-SMR drives(e.g., F2FS [17] and Btrfs [27]). Further, developers can modify their applications to accommodate HM-SMR drives (e.g., GearDB [32], SMORE [19], and SM-RDB [26]) or directly employ them in archival-class object storage systems such as Alibaba Archive Storage Service [1] and Huawei Object Store [18].

Unfortunately, these existing HM-SMR solutions can not be applied to standard-class Alibaba Cloud Object Storage Service (OSS) . First, setting the HM-SMR drive as a block device (i.e., via dm-zoned [21]) could suffer a significant throughput drop due to frequent buffer zones reclaiming after random updates (e.g., a 56.1% drop under a sustained write workload [22]). Second, employing log-structured file systems to manage HM-SMR drives can experience throughput variations due to GC in file systems. For example, our evaluation shows that the throughput of F2FS on a HM-SMR drive can drop 61.5% due to frequent F2FS GCs triggered by random deletions. Third, though archival-class and standard-class OSS share the same data abstraction (i.e., object), they have drastically different Service Level Objectives (SLOs). Therefore, a design that works well in the archival class may not deliver satisfying performances in the standard class.

Our benchmarks show that existing SMR translation layers or file systems could result in severe overhead possibly due to garbage collection. Moreover, log-structured design offers direct support to SMR drives and could achieves a high throughput when not affected by GC. Besides, GC, which is inevitable due to the append-only nature of SMR zones, could be alleviated via workload-aware data placement.

In this paper, we describe SMRSTORE, a high-performance HM-SMR storage engine co-designed with Alibaba Cloud

*Corresponding author. erc.xec@alibaba-inc.com

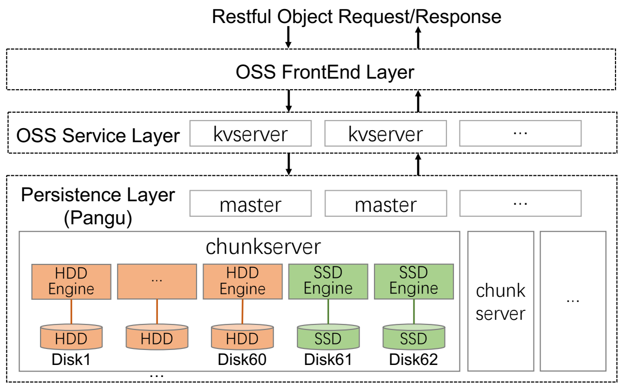

Figure 1: Architecture overview of OSS (§2.1). *The red shaded HHD engines refer to traditional ext4-based storage engines. The green shaded SSD engines refer to user-space storage engines.*

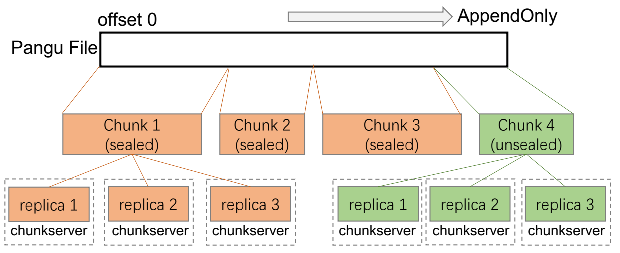

Figure 2: Semantics of PANGU file and chunk. *This figure shows a PANGU file consists of four chunks. Only chunk 4 (the last chunk) is not sealed (writable). PANGU keeps multiple replicas for each chunk across chunkservers to protect data against any failures.*

standard-class OSS. There are three key features in SMRSTORE. First, SMRSTORE is a user-space storage engine that does not require local file system support and directly implements chunk interfaces of PANGU distributed file system.

Second, SMRSTORE strictly follows a log-structured design to organize HM-SMR on-disk layout. In SMRSTORE, the basic building block is a variable-length customized log format called *Record*. We use records to persist data, and form various metadata structures (e.g., checkpoint and journal).

Third, we design a series of workload-aware zone allocation strategies to reduce the interleaving of different types of OSS data & metadata in zones. These effort help us to effectively lower the overhead of GC in HM-SMR drives.

We extensively evaluate SMRSTORE under various scenarios. The results show that PANGU chunkserver with SMRSTORE achieves more than 110MB/s throughput in high concurrent write workloads, 30% higher than the previous generation design (i.e., chunkserver with Ext4 on CMR drives). Moreover, on a storage server (60 HDDs and 2 cache SSDs), chunkserver with SMRSTORE provides steady 4GB/s write throughput in macro benchmarks, 2.16x higher than F2FS. Third, in OSS deployment, the performances of the HM-SMR cluster are comparable to the CMR cluster in all aspects.

The rest of the paper is organized as follows. We describe standard-class OSS in Alibaba and the HM-SMR drives in §2. Then, we analyze the pros and cons of existing solutions (§3). Further, we demonstrate the design choices (§4), the detail implementation of SMRSTORE (§5) and the evaluation (§6). We conclude with discussions on the limitation of SMRSTORE (§7), the related work (§8) and a short conclusion (§9).

2 Background

2.1 Alibaba Cloud OSS

Alibaba Cloud OSS offers four classes of services, including standard, infrequent access, archive, and cold archive (prices in descending order and retrieval time in ascending order). Standard-class OSS, usually for hosting hot data, offers the fastest SLOs with the highest economical cost.

Architecture. Figure 1 illustrates the three layers in Alibaba Cloud standard-class OSS stack, including an OSS frontend layer, an OSS service layer, and a persistence layer. The frontend layer pre-processes users' http requests and dispatches them to the service layer. The service layer, consisting of multiple KV servers, has two functionalities. First, the service layer writes the objects to PANGU files. Second, the service layer maintains the objects' metadata (the mapping from objects' names to locations within the corresponding PANGU files) using an LSM-tree based KV store [25], and writes these metadata to additional PANGU files. The persistence layer is our distributed file system PANGU.

PANGU overview. PANGU is a HDFS-like distributed file system and each PANGU cluster comprises a set of masters (handling PANGU files' metadata, not objects' metadata) and up to thousands of chunkservers (storing data of PANGU files). Each chunkserver exclusively owns a physical storage server to operate, consisting of 60 HDDs for persistence and two high performance SSDs for caching [1]. We leverage Linux kernel storage stack (Ext4 file system and libaio with O_DIRECT) for the HDD storage engines and build a user-space storage file system for the SSD engines.

PANGU data abstractions. Figure 2 illustrates two levels of abstractions in PANGU, file and chunk. Each file is append-only and can be further split into multiple chunks. Each chunk has a Chunk ID (a 24-byte UUID) and is replicated via copies or erasure coding. PANGU can create, write(append), read, delete, and seal a chunk. Similar to the "extent seal" in Windows Azure Storage [13], PANGU seals a chunk when: i) the size of chunk—including data and corresponding checksum—reaches the limit; ii) the application closes the PANGU file when writing is finished; iii) in the face of failures (e.g., network timeout). Due to case ii) and iii), the chunks can be of variable sizes. Note that only the last chunk of a PANGU file can be appended (not sealed) and only sealed chunks can be

[1] PANGU also supports other services (e.g., block storage and big data) and can have various modifications. In this paper, our discussion on PANGU and corresponding software/hardware setups only apply to OSS scenario.

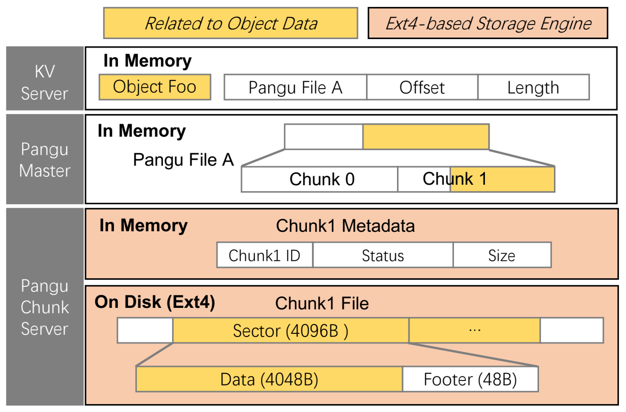

Figure 3: Dataflow with traditional storage engine (OSS (§2.1). *This figure illustrates the write path of an object in traditional CS-Ext4 stack (red shaded). The yellow shaded area means it is related to the object "Foo". The data is split as a series of 4048B segments where each is attached with a 48B footer for checksum.*

flushed from cache SSDs to HDDs for persistence. Storage engine does not provide any redundancy for failures. Instead, we rely on PANGU providing fault tolerance for each chunk across chunkservers by replication or erasure coding.

I/O Path. Figure 3 presents high-level write flows in OSS with the traditional Ext4-based storage engine. We use an example object called "Foo" to highlight the write flow. The KV server chooses the PANGU file *A* for storing "Foo". Then the KV server uses the PANGU SDK or contacts the PANGU master to locate the tail chunk (i.e.,chunk-1) and its respective chunkservers. Further, the chunkserver appends the object's data (with checksums) to the corresponding Ext4 file. To verify data integrity, we break the object's data into a series of 4048-Byte segments. We attach to each segment a 48 Byte footer which includes the checksum and segment locations in chunk. Note that each chunk in the chunkserver is an Ext4 file with the chunk ID as its filename.

Workloads. A KV server can open multiple PANGU files to store or retrieve the data and metadata of objects, and perform GC on deleted objects in PANGU files. From the perspective of chunkservers, we define an active PANGU file as a **stream**. The stream starts as the PANGU is opened by a KV server for read or write, and ends when the KV closes the PANGU file. Based on the operations (read or write) and types (metadata, data or GC), we can categorize the workloads issued by KV servers as five types of streams. Table 1 lists the characteristic of each type of streams. The "Concurrency" refers to the PANGU file concurrency, namely the number of PANGU opened files on a chunkserver to append data. The "Lifespan" refers to the expected lifespan of the data on disk (from being persisted to deleted), NOT the duration of the streams.

- *OSS Data Write Stream.* Persisting object data requires low latency to achieve quick response. Therefore, object data

Type	Latency (ms)	Concur-rency	iosize (Byte)	Lifespan (Day)
OSS Data W	<1	~1500	512K-1M	<7
OSS Data R	<20	-	512K-1M	-
OSS Meta W	<1	~2000	4K-128K	<60
OSS Meta R	<20	-	4K-128K	-
OSS GC	<20	~100	512K-1M	<90

Table 1: Characteristics of streams on a chunkserver(§2.1).

are first written to SSD caches and later moved to HDDs. Normally, hundreds of PANGU chunks are opened for writing on a chunkserver, thereby yielding high concurrency.
- *OSS Data Read Stream.* OSS directly reads object data from HDDs to achieve high throughput. Due to space limits, object data for read are not cached in the SSDs.
- *OSS Metadata Write Stream.* OSS metadata stream includes objects' metadata formatted as Write-Ahead Logs and Sorted String Table Files from KV store. The metadata are first flushed to SSD cache and then migrated to HDDs. The metadata accounts for around 2% of the total capacity used (around 24TB per chunkserver).
- *OSS Metadata Read Stream.* The KV server maintains a cache for object index. In most cases, the metadata read directly hits the KV index cache and returns. If cache misses, OSS routes to SSTFiles in PANGU.
- OSS *GC Stream.* The garbage collection in OSS service layer (referred to as OSS GC) is to reclaim garbage space in PANGU files. A PANGU file can hold multiple objects. When a certain amount of objects are deleted in a PANGU file, OSS would re-allocate the rest to another PANGU file, and delete the old file. The chunks written by OSS GC streams account for more than 80% of the total capacity used in one chunkserver. OSS GC streams run in background and directly routed to HDDs for persistence.

2.2 Host-Managed HM-SMR

HM-SMR drives overlap the tracks to achieve higher areal density but consequently sacrifices random write support. Specifically, HM-SMR drives organize the address space as multiple fixed-size zones including sequential zones (referred to as zones) and a few (around 1%) conventional zones (referred to as czones). For example, the Seagate SMR drive we use in this paper has a capacity of 20TB and 74508 zones (including 800 czones). The size of each zone is 256MB, and it takes around 20ms, 24ms and 22ms for opening, closing and erasing a zone, respectively. Note that, for certain SMR HDD models (e.g., West Digital DC HC650), there is a limit on the number of zones to be opened concurrently.

Device mapper translation. A straightforward solution is to insert a shim layer, called shingled translation layer (STL), such as dm-zoned [21], to provide dynamic mapping from logical block address to physical sectors and hence achieve random-to-sequential translation. Apparently, the major advantage of this approach is allowing the users (e.g., chunkserver process) to adopt the HM-SMR drives as cost-efficient drop-in replacement for CMR drives.

SMR-aware file systems. The log-structured design file systems (e.g., F2FS) make them an ideal match for the append-only zone design of HM-SMR disks. For example, F2FS started to support zoned block devices since kernel 4.10. Users can mount a F2FS on an HM-SMR drive and utilize the F2FS GC mechanism to support random writes. Similarly, Btrfs, a file system based on copy-on-write principle, currently provided an experimental support for zoned block device in kernel 5.12.

End-to-End Co-design. Instead of relying on dm-zoned or general file systems, applications that perform mostly sequential writes can be modified to adopt HM-SMR. The benefits of end-to-end integration has been proved by several recent works, such as GearDB [32], ZenFs [11], SMORE [19], etc. Applications could eliminate the block/fs-level overhead and achieve predictable performance by managing on-disk data placement and garbage collection at application level [24,30].

3 Evaluating Existing Solutions

We evaluate running F2FS atop HM-SMR drives with microbenchmark and macrobenchmark (i.e., simulated OSS workloads). We compare the performances of chunkservers with Ext4 on CMR drives (referred to as CS-Ext4) and F2FS on HM-SMR drives (referred to as CS-F2FS).

3.1 Evaluation Configurations

	CMR Server	SMR Server
OS	Linux 4.19.91	
CPU	2*Intel(R) Xeon(R) Platinum 8331C CPU@2.50GHz 48 Physical Cores 96 Threads	
SSD	2*INTEL SSDPF21Q800GB	
Mem	512G	
HDD	60*ST16000NM001G-2KK103 Rand. 4KB(IOPS): 113 Seq. 512KB(MB/s): 254.8(W) 254.5(R)	60*ST20000NM001J-2U6101 Rand. 4KB(IOPS): 121 Seq. 512KB(MB/s): 255.7(W) 255.6(R)

Table 2: Configurations of storage servers in evaluation. *A SMR server has the exact same setups with a CMR server, except the HDDs are 20TB SMR HDDs instead. The raw performance comparison with queue depth 1 random read and queue depth 32 sequential read/write is listed in the last row.*

Environment Setup. Table 2 lists the configurations of the storage servers in the evaluation. Note that F2FS does not support devices with a capacity larger than 16TB. Therefore, we format the disk with 6TB capacity. Moreover, in all cases we disable the disk write cache by hdparm [4] tool to prevent data loss upon crashes, a mandatory setting in OSS.

Workloads. For both micro- and macro-benchmarks, we use the Fio [3] (modified to use the PANGU SDK) as the workload

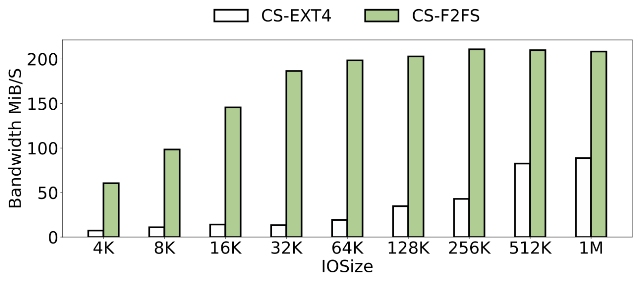

Figure 4: High Concurrency Write Throughput (§3.2). *The figure shows the write throughput of CS-Ext4 and CS-F2FS in microbenchmarks.*

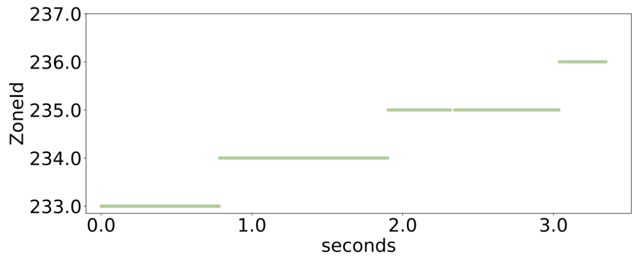

Figure 5: F2FS Access Pattern (§3.2). *The figure shows the accessed zoneIds by F2FS in a few seconds. F2FS writes all data into one zone in a period of time and switches to the next only when the zone is full.*

generator. For microbenchmark, we start a chunkserver with one disk, and focus on testing write throughput with different I/O sizes in the clean state (no F2FS GC). We start a Fio with 4 numjobs, 4 iodepth, and 128 nrfiles to simulate a high write pressure.

For macrobenchmark, we evaluate chunkserver with all disks loaded (60 HDDs and 2 cache SSDs) and run four Fio processes to simulate different types of write streams. Table 3 lists the detailed configurations. Note that we use two Fio processes to simulate two kinds OSS GC streams (i.e., OSS GC Wr 1 and 2). For OSS GC Wr 2, we use a smaller chunk size and rate (64MB and 20MB/s) to simulate the situations where the chunks are sealed before reaching the size limit (due to reaching the end of PANGU file or encountering I/O failures).

The macrobenchmark generates a stable 4GB/s throughput to simulate a typical high pressure workload. There are two phases in this test. In the first phase, we simply let the four streams to fill the HDDs and there is no file deletions. In the second phase, the utilization of capacity reaches around 80% (around 12 hours after the first phase started) and triggers the random deletions to maintain the utilization rate at around 80%. The average chunk deletion rate on a chunkserver ranges from 4 operations per second (ops/s) to 15 ops/s.

Stream Type	#Fio	Target	numjobs	iodepth	iosize	nrfiles	chunk size	rate
OSS GC Wr 1	1	HDDs	8	32	1MB	25	256MB	400MB/s
OSS GC Wr 2	1	HDDs	8	32	1MB	25	64MB	20MB/s
OSS Data Wr	1	SSDs	3	32	1MB	300	256MB	200MB/s
OSS Meta Wr	1	SSDs	1	8	4KB-128KB	500	4MB	80MB/s

Table 3: Macro benchmark setups (§3.1). OSS *GC Wr 1 refers to* OSS *GC streams with large chunks.* OSS *GC Wr 2 refers to* OSS *GC streams with small chunks.* OSS *Data Wr refers to* OSS *object data write streams.* OSS *Meta Wr refers to* OSS *metadata write streams.*

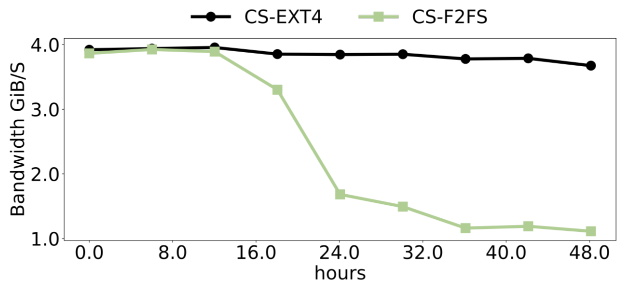

Figure 6: CS-Ext4 vs CS-F2FS in macrobenchmark (§3.2). *The test starts on empty disks and with steady 4GB/s throughput. At hour 12, the capacity utilization reaches 80% and random deletions occur.*

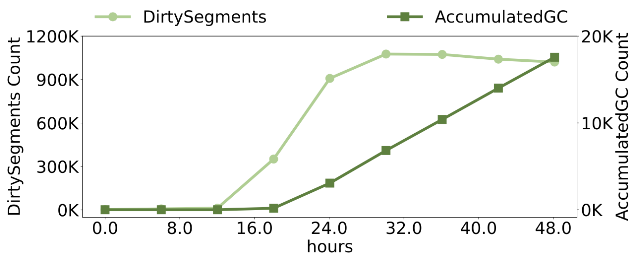

Figure 7: F2FS GC related metrics (§3.2.) *This figure illustrates the status of F2FS. The dirty segment count on the left axis reflects the generation of garbage space. The increasing accumulated GC count (right Y axis) indicates the continuing GC activities which are the immediate causes of the performance drop.*

3.2 Performance Comparison

Microbenchmark Performance. Figure 4 shows that CS-F2FS on HM-SMR drives achieves 1.3x - 12.9x higher throughput compared to CS-Ext4 on CMR drives. This is because F2FS writes from different streams to one zone at a time and thus always performs sequential writes. Figure 5 shows the accessing distribution of SMR ZoneIDs from the CS-F2FS. We can see F2FS fills up one SMR zone at a time (e.g., Zone 233 from second 0). This allocation strategy avoid overhead from jumping between zones.

Macrobenchmark performance. Figure 6 shows the throughput performances of CS-F2FS and CS-Ext4 along time. Initially, we can observe that both maintain stable throughput from hour 0 to 12. Then, after 12 hours, the CS-F2FS quickly drops and remains a low throughput for the rest of the time. This is because the random deletion starts and GC in F2FS kicks in to handle the increasing amount of obsolete data (see Figure 7). Note that F2FS puts chunks from different types of streams into one zone. Due to random deletions, severe F2FS GC can be frequently triggered and influence the OSS metadata/data streams, resulting in a performance drop.

We are aware that F2FS provides multi-head logging to separate streams on disk, but this technique cannot separate chunks from the same type of streams. In practice, PANGU file concurrency in each type of OSS stream can range from tens to hundreds, and F2FS would write all the chunks from the same type of stream into one zone. Therefore, random deletions on those chunks (a common scenario in standard-class OSS) still trigger severe F2FS GCs.

4 SMRSTORE Design Choices

No local file system. We build SMRSTORE to support chunk semantics (including `chunk_create`, `chunk_append`, `chunk_read`, `chunk_seal`, and `chunk_delete`) on SMR zoned namespace. There are three functionalities in SMRSTORE to support this feature. First, SMRSTORE directly manages the disk address space for persisting metadata (i.e., checkpoints and journalings) and data (i.e., the chunks). Second, SMRSTORE manages a mapping table between chunks and SMR zones to translate logical range in chunks (via chunkId, offset, and length) to the physical locations on disk (i.e., zoneId, offset, and length). Third, SMRSTORE orchestrates the lifecycle of zones and data placement strategies in the zones.

Everything is log. SMRSTORE stores both metadata and data as logs in SMR sequential zones. Specifically, SMRSTORE uses a basic structural unit, called *record*, to form different types of metadata and data. To avoid wasting space, record is of variable-length and enforces 4KB alignment with disk physical sector.

Guided data placement. Since SMR zones are append-only (except a few czones), and chunks from different PANGU files can be interleaved in SMR zones, deleting PANGU chunks can leave zones with obsolete data. This requires SMRSTORE to migrate valid data from old zones to new ones, termed as SMR GC in this paper. SMRSTORE reduces SMR GC by: i) only allowing chunks to be mixed in a zone if they are from the same type of streams (i.e., similar lifespans); ii) trying to allocate an exclusive zone for each large chunk if possible.

Note that SMR GC is different from the OSS GC. In OSS GC, after objects deleted by users, the corresponding PANGU

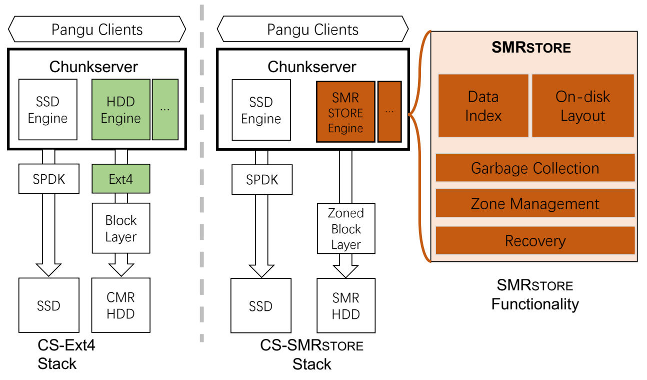

Figure 8: Overview of CS-EXT4 and CS-SMRSTORE (§5.1). SMRSTORE *is integrated in chunkserver, runs in the user space and communicates with HM-SMR drives directly by ZBD interface.*

files can be partially filled with obsolete objects. KV servers would create new PANGU files to store the valid objects collected from old PANGU files (i.e., generating OSS GC streams).

5 SMRSTORE Design & Implementation

5.1 Architecture Overview

Figure 8 shows a side-by-side comparison between running chunkserver with Ext4 on CMR disks (CS-Ext4), and with SMRSTORE on SMR disks (CS-SMRSTORE). The main difference is the addition of SMRSTORE to the chunkserver, sitting in the user space, and communicating with the SMR disks via Zoned Block Device (ZBD) subsystem. Next, we discuss the key functionalities of SMRSTORE:

- *On-disk data layout.* SMRSTORE divides an HM-SMR drive into three fixed-size areas, namely the superzone, the metazones, and the datazones. The SMRSTORE uses "record" as the basic unit for metazone and datazone.
- *Data index.* SMRSTORE employs three levels of in-memory data structures, including chunk metadata, index group, and record index, to map a chunk to a series of records on the disk.
- *Zone Management.* SMRSTORE uses a state machine to manage the lifecycle of zones, and keeps metadata (e.g., status) of each zone in the memory. Further, SMRSTORE adopts three workload-aware zone allocation strategies to achieve low SMR GC overhead.
- *Garbage Collection.* SMRSTORE periodically performs SMR GC to reclaim area with stale data at the granularity of zones. There are three steps in SMR GC procedure: victim zone selection, data migration, and metadata update.
- *Recovery.* Upon crashes, SMRSTORE restore through four steps: recovering meta zone table, loading the latest checkpoint, replaying journals, and completing the chunk metadata table by scanning opened data zones.

CS-SMRSTORE I/O Path. When replacing the storage engine with a SMRSTORE engine, the KV server and PANGU master follow the same procedures shown in Figure 3. As illustrated in Figure 9, SMRSTORE no longer relies on local system support and uses an in-memory chunk metadata table for mapping. SMRSTORE first locates the table entry and its index group linked list by using the chunk ID as index. SMRSTORE further identifies the targeted index group or creates a new one. Then, SMRSTORE appends data to the datazone (indicated by the targeted index group) as record(s), and updates the record index(es) in the corresponding index group.

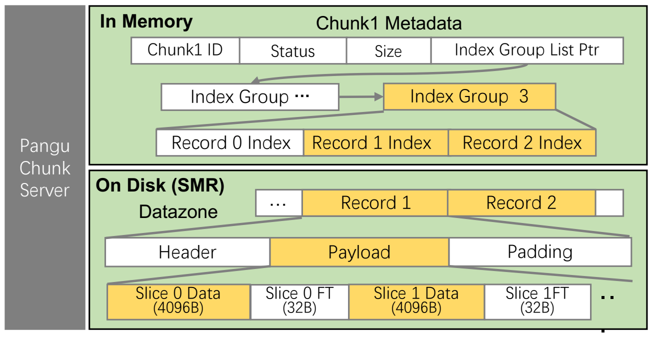

Figure 9: Dataflow in SMRSTORE engine. (§5.1). *Compared to Figure 3, the storage engine is* SMRSTORE *(green shaded) and the disk is an HM-SMR drive.* ***FT****: slice footer.*

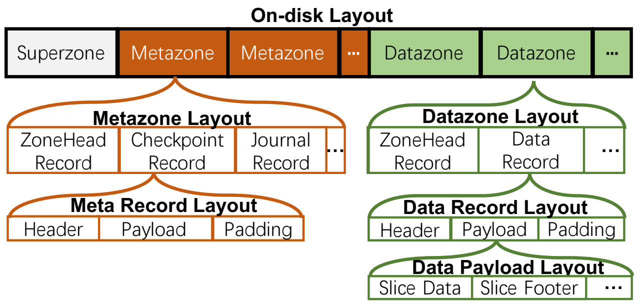

Figure 10: On-disk Data Layout of SMRSTORE (§5.2). SMRSTORE *divides a disk into three partitions. Both metadata and data are implemented based on the unified data structure called record. The record can be of variable in length and have different type. The payload of a data record is divided into several slices to support partial read.*

5.2 On-Disk Data Layout

Overview. Figure 10 shows the three partitions of an HM-SMR drive under the SMRSTORE, including one superzone, multiple metazones, and multiple datazones. All partitions are fixed-sized and statically allocated. In other words, we place the superzone on the first SMR zone, the metazones occupy the next 400 SMR zones, and the rest of SMR zones are assigned as datazones. We do not allow metazones and datazones to be interleaved along disk address space to facilitate

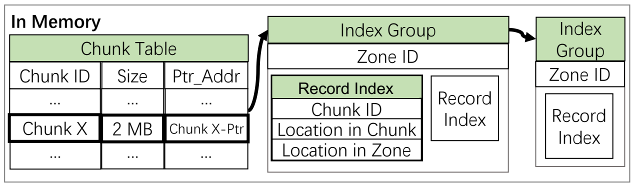

Figure 11: Data Index of SMRStore (§5.3). *In the chunk metadata table, each chunk has a pointer to an index group list. Each index group can have multile record indexes in a zone. The index groups and records are all sorted by the offset to the chunk.*

the metazones scanning during recovery.

Superzone. The superzone stores the information for initialization, including the format version, the format timestamp, and other system configurations.

Metazone. Inside the metazone, there are three types of metadata: the zonehead, checkpoint , and journal. Note that the metazones only store metadata of SMRSTORE not the metadata of OSS (i.e., data from OSS metadata stream). The metadata are composed by different types of records. The zonehead record stores the zone-related information, such as the zone type and the timestamp of zone allocation (used for recovery). The checkpoint is a full snapshot of in-memory data structures while the journals contain key operations of chunk and zone which we further introduce in §5.6.

Inside each record, there are also three fields: the header, the payload, and the padding. The header specifies the type of records (i.e., zonehead, checkpoint, or journal record), the length of the record and the CRC checksum of the payload. The payload contains the serialized metadata. An optional padding is appended at the end of the record as the SMR drive is 4KB-aligned.

Datazone. The datazones occupy the rest of the disk. In each zone, there are two types of records, the zonehead record and data record. The zonehead record is similar to the metazone zonehead record except the zone type.

Data record & slice. The payload of data record hosts user's data (i.e., a proportion of the chunk). The padding at the tail of a data record is used to bring it a multiple of 4096 bytes (i.e., 4KB-aligned). However, the payload field of data record is different from other types of records (see bottom right of Figure 10). To avoid read amplification, the payload is further divided into 4096-Byte slices, with a 32-Byte slice footer appended to each slice. The slice footer contains the chunk ID (24 bytes), the logical offset to chunk (4 bytes) and the checksum of slice data (4 bytes). Without payload slicing, reading a 4KB from a 512KB record would require SMRSTORE to fetch the whole record for verifying the payload with the record's checksum. Now, with slices, reading a 4KB only needs to read at most two slices, and SMRSTORE can use the footer in the slice for checksum verification.

5.3 Data Index

SMRSTORE uses an in-memory data structure, called record index, to manage the metadata of each record. The record index includes Chunk ID, the logical location of user's data in the chunk (i.e., chunk offset and size of user's data) and record's physical location in the datazone (i.e., offset in the datazone and size of the record).

A chunk usually can have multiple records that are distributed among several datazones. Note that SMRSTORE appends the data of a chunk to only one datazone at a time until that datazone is full. This guarantees two properties: i) the records in each datazone together must cover a consecutive range of the chunk; ii) the covered chunk ranges in each datazone are not overlapped with each other.

Therefore, we group the record indexes of a chunk in each datazone as an index group. Based on i), inside each index group, we can sort record indexes based on their chunk offsets. The index group also includes the corresponding datazone ID. Moreover, due to property ii), we can further sort the index groups of a chunk, based on the chunk offset of first record index in each group, as a list.

Then, SMRSTORE organizes the metadata of chunks as a table (see the left of Figure 11). Each entry of the table, indexed by the Chunk ID (a 24 byte UUID), contains the chunk size (the total length of the chunk on this disk), the chunk status (sealed or not, not illustrated in the figure), and the corresponding sorted list of index groups.

When receiving a read request (specified by the chunk ID, the chunk offset, and the data length), SMRSTORE can locate the chunk metadata with the ChunkID, find the target index group in the sorted list with chunk offset, and locate corresponding record index(es) with the chunk offset and data length.

For a write request, SMRSTORE always locates the last index group of the target chunk. If there are enough space left on the corresponding datazone, SMRSTORE appends the data to the datazone as a new record and adds the new record index to the index group. If not, SMRSTORE allocates a new datazone, appends the data, and adds the record index to the new index group.

5.4 Zone Management

Zone state machine. SMRSTORE employs a state machine to manage the status of datazones as shown in Figure 12. SMRSTORE maintains a pool of opened zones (55 zones by default) for fast allocation. SMRSTORE only resets the GARBAGE zones to FREE zones when the amount of FREE zones are not enough. Metazone follows the similar state machine except there is no pool of opened metazones.

Zone table. SMRSTORE maintains a zone table in the memory. Each entry of the zone table includes the zone ID, the zone status (OPENED, CLOSED, etc.), a list of live index groups, and a write pointer. We further introduce the usage

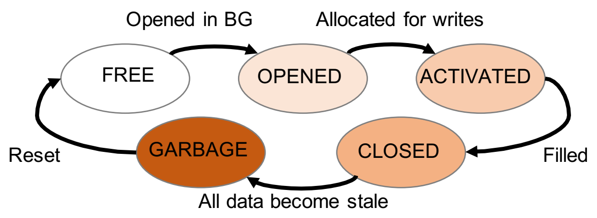

Figure 12: Zone State Transition of SMRStore (§5.4). SMRSTORE *maintains a pool of opened zones for fast allocation. When a zone is assigned to a new chunk, it transitions to ACTIVATED status. If a zone is closed, it will not be reopened for write before reset.*

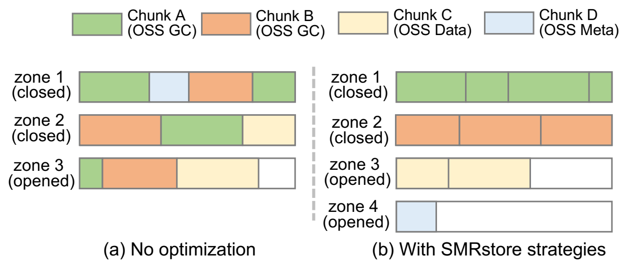

Figure 13: The effectiveness of SMRSTORE zone allocation strategies. (§5.4). *Chunk A-D come from four different* OSS *streams and shaded with corresponding colors, respectively. Subfigure(a) represents a possible scenario under the random allocation (no optimization) and (b) illustrates a possible layout with* SMRSTORE *strategies enabled. Each data block may be composed of one or more records.*

of per-zone live index groups list when discussing SMR GC (§5.5) and recovery (§5.6)

Zone allocation. Earlier in F2FS (see §3), we showed that allocating chunks from different types of OSS streams to the same datazone can result in high overhead led by frequent F2FS GC. One can allocate a single datazone for each chunk to reduce such GC. However, this can in return waste considerable space. For example, chunks from OSS metadata stream are usually just several megabytes large, much smaller than the size of a datazone (256MB).

Hence, a more practical solution is to only pursue the "one chunk per zone" for large chunks and let the small chunks with similar lifespans to be mixed together. A challenge here is that the size of a chunk is only determined after it is sealed. In other words, when allocating datazones for incoming OSS streams, SMRSTORE does not know the sizes of the chunks. Therefore, we design the following zone allocation strategies.

- ① *Separating streams by types.* Note that different OSS types of streams can have disparate characteristics (see Table 1). Therefore, we modify the OSS KV store to embed the types of the OSS streams (i.e., OSS Metadata, OSS Data or OSS GC) along with the data. SMRSTORE only allows chunks from the same type of streams to share a datazone.
- ② *Adapting chunk size limit for datazone.* Recall that a chunk is sealed when it reaches the size limit, the end of PANGU file or I/O failures. Hence, we configure the size limit of a chunk (including its checksum) to match the size of one datazone (256MB). A chunk may still be sealed well under 256MB (e.g., due to I/O errors). In that case, the left space would be shared with other chunks from the same type of streams if necessary. Note that we still use the default size limit (64MB) for chunks from OSS metadata stream as the corresponding PANGU files are usually small (several to tens of MBs each).
- ③ *Zone pool & round-robin allocation.* SMRSTORE pre-opens and preserves zones for different types of OSS streams. Specifically, we prepare 40, 10 and 5 opened zones for OSS GC, Data and Metadata stream, respectively. The rationale is that OSS GC stream is the main contributor of the I/O traffic. The OSS Metadata and Data streams have high PANGU file concurrency but can be throttled by the cache SSDs. Moreover, SMRSTORE allocates zones for new chunks in a round-robin fashion to reduce the chances of chunks to be mixed together.

In Figure 13, we use an example to showcase the effectiveness of our strategies. Consider there are four OSS streams—two OSS GC streams (green and red), one OSS Data stream (yellow) and one OSS Metadata stream (blue). If we do not enable any strategies, SMRSTORE would allocate datazones one by one for the incoming chunks. As a result, we can expect datazones to be interleaved as shown in Figure 13 (a), similar to the F2FS scenario in §3. In this case, for example if chunk *A* is deleted, all three datazones would have chunk *A*'s stale data and require further SMR GC to reclaim the space.

Now, in Figure 13(b), due to Strategy ①, chunks from different types of streams are no longer mixed together. Moreover, since we reconfigure the size limit of chunks (Strategy ②) and use round-robin allocation (Strategy ③), we can see that chunk *A* and *B* can both own a zone exclusively and fill the entire zone. The three strategies achieve our goal by allocating large-sized chunks with exclusive zones. Now, if chunk *A* is deleted, SMRSTORE can directly reset zone 1 (i.e., no SMR GC needed).

5.5 Garbage Collection

SMRSTORE performs garbage collection in three steps:

Victim zone selection. The SMRSTORE first choose a victim zone among the CLOSED ones to perform SMR GC. We use greedy algorithm to select a zone with most garbages.

Data migration. For the selected victim zone, by scanning live index group list from the zone table, SMRSTORE can identify valid data in this zone and migrate them to an available zone which is activated only for garbage collection. Moreover, SMRSTORE enables a throttle module that dynamically limits the throughput of SMR GC to alleviate interference to the foreground I/O.

Metadata update. During migration, SMRSTORE creates index groups with new record indexes for migrated data. After SMR GC finished, SMRSTORE replaces the old index groups in the linked list with the new ones. Finally, SMRSTORE updates the zone table by marking the victim zone as GARBAGE.

5.6 Recovery

SMRSTORE relies on journals and checkpoints to restore the in-memory data structures. In this section, we first introduce the detailed design of journal and checkpoint. Then, we discuss the four steps of recovery.

Checkpoint design. The checkpoint of SMRSTORE is a full snapshot of the in-memory data structures including the chunk metadata table (§5.3) and zone table (§5.4). SMRSTORE periodically creates a checkpoint and persists it into the metazones as a series of records. The zone table is usually small and can be stored in one record. The chunk metadata table is much larger (including all the index groups and record indexes, see Figure 11) and requires multiple records to store. Therefore, we also use two records to mark the start and end of a checkpoint, called checkpoint start/end record.

Journal design. In SMRSTORE, only the create, seal, delete operations of chunk, and the resetting of the zone need to be recorded by journals. Note that SMRSTORE does not journal write operation (i.e., chunk append) as this can severely impact the latency. Instead, we can restore the latest data locations by scanning the previously opened zones. SMRSTORE journals the zone reset operation to handle the case where the same zone may be opened, closed and reused multiple times between two checkpoints. Note that the checkpoint of SMRSTORE is non-blocking, hence the journal records and checkpoint records can be interleaved in the metazones.

Recovery process. The four steps of recovery are as follows:

- *Identifying the latest valid checkpoint.* The first step is to scan zonehead record of each metazone. Recall that, when opened, each metazone is assigned with a timestamp and stored in the zonehead record. Now, by sorting the timestamps, we can scan the metazones from the latest to the earliest to locate the most recent checkpoint end record and further obtain the corresponding checkpoint start record.
- *Loading latest checkpoint.* By scanning records between the checkpoint start and end record, SMRSTORE can recover zone table and chunk metadata table (including index groups and record indexes) from the most recent checkpoint.
- *Replaying journals.* Next, after the checkpoint start record, SMRSTORE replays each journal record till the checkpoint end record to update the zone table, and chunk metadata table.
- *Scanning datazones.* Recall that the journals do not log the write (i.e., `chunk_append()`) operations in order to reduce impacts on the write latency. Therefore, the last step of recovery is to check the datazones that have not been covered by the checkpoint and journals for yet-to-be-recovered writes. SMRSTORE checks the validity (i.e., allocated for writes before crash) of datazones by reading their zonehead records. For each valid datazone, SMRSTORE verifies the data record one by one with the per-record checksums. Finally, SMRSTORE updates the in-memory chunk metadata table (including index groups and record indexes).

6 Evaluation

Software/Hardware setup. We evaluate the end-to-end performance of three types of candidates, including the chunkserver with CMR drives (i.e., CS-Ext4), the chunkserver with F2FS on SMR drives (i.e., CS-F2FS), and SMRSTORE as the storage engine for chunkserver on SMR drives (i.e., CS-SMRSTORE). Additionally, we setup two alternative versions of CS-SMRSTORE. The CS-SMRSTORE-20T shows the performance with full-disk 20TB capacity and the CS-SMRSTORE-OneZone imitates the data placement strategy of F2FS (i.e., mixing data from different streams into one zone). Our node configurations are listed in Table 2.

Workloads setup. We use Fio (modified to use the PANGU SDK) to generate workloads. Our experiments evaluate the following aspects of SMRSTORE.

- *High concurrency micro benchmark.* We extend the microbenchmark in §3 to further evaluate the candidates under highly concurrent random read workloads.
- OSS *simulation macro benchmark.* We also repeat the multi-stream OSS simulation in §3 to evaluate the candidates with multiple write streams, random file deletion and high disk utilization rate.
- *Garbage collection performance.* We evaluate the SMR GC overhead in SMRSTORE and further examine the effectiveness of data placement strategies by comparing corresponding SMR GC overheads under different strategy setups.
- *Recovery.* To evaluate the recovery performance, we restart chunkserver on 20TB SMR drives with 60% capacity utilization, then analyze time consumption in recovery.
- *Resource consumption.* We compare the resources, such as CPU and memory usage, between CS-Ext4 and CS-SMRSTORE (i.e., the two generations of storage stack for standard-class OSS), under a similar setup.
- *Field deployment.* Both CS-Ext4 and CS-SMRSTORE are currently deployed in standard-class OSS. We summarize, demonstrate, and compare key performance statistics of a CS-Ext4 cluster and a CS-SMRSTORE cluster in the field.

6.1 High Concurrency Microbenchmark

In this microbenchmark, we evaluate the candidates on one disk (SMR or CMR) with two types of workloads: High Concurrency Write (HC-W) and High Concurrency Rand

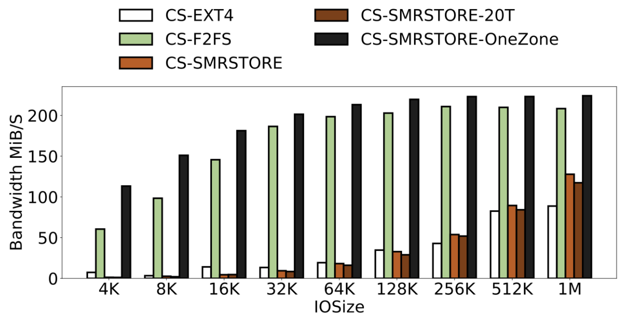

Figure 14: High Concurrency Write Throughput (§6.1). *This figure presents the comparison of write throughput between different storage engines. CS-F2FS (green) and CS-*SMRSTORE*-OneZone (black) achieve rather high throughputs as they place all incoming chunks onto the same zones, which can incur high F2FS/SMR GC overhead later.*

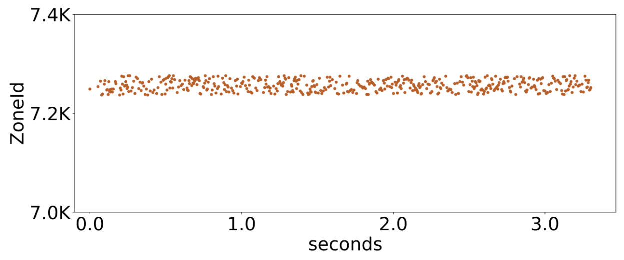

Figure 15: SMRSTORE Access Pattern (§6.1). *The figure presents the distribution of accessed zones under* SMRSTORE *during a few seconds. Each red dot represents the corresponding zone is accessed (zone ID on the Y axis). This shows the effectiveness of the round-robin allocation, rendering a clear contrast to the zone accessing in CS-F2FS (Figure 5).*

Read (HC-RR). Note that in this experiment, the disk is in the clean state and thus would not trigger F2FS or SMR GC.

Figure 14 shows the HC-W throughput of each candidate under different I/O sizes (from 4KB to 1MB). We can see that CS-SMRSTORE-OneZone and CS-F2FS always have much higher throughput. As discussed in §3.2, flushing data from different streams to enforce the “one zone at a time” policy can significantly benefit the throughput during the clean state (no deletion and F2FS/SMR GC).

For the rest three, their performance gradually increase with I/O size. We notice that, for small I/O size (i.e., <32KB), SMRSTORE shows low throughput. This is caused by the round-robin zone allocation strategy which tends to allocate a new zone for each new chunk to avoid mixed placement, thereby generating random writes for the disk (see Figure 15). As I/O size increases, the throughput of SMRSTORE gradually catches up at 128KB, finally reaches 110MB/s and exceeds CS-Ext4 by 30% at 1MB I/O size. This is acceptable as most writes in standard-class OSS are larger than 128KB (see Table 1).

Figure 16 shows the performance comparison of candidates

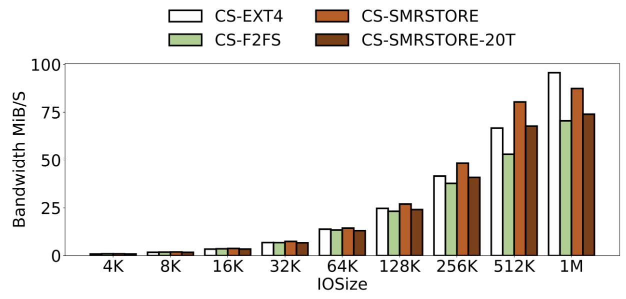

Figure 16: High Concurrency RandRead Throughput (§6.1).

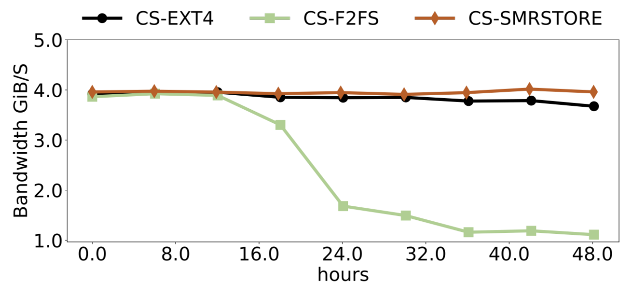

Figure 17: Throughput Comparison of Multi-Stream Benchmark (§6.2).

with HC-RR. We can see CS-SMRSTORE manages to deliver comparable performance to the CS-Ext4. Moreover, in both HC-W and HC-RR experiments, we can observe that the full-disk version (i.e., CS-SMRSTORE-20T) does not suffer severe performance drops.

6.2 Multi-Stream Benchmark

Next, same as the multi-stream experiment in §3.2, we evaluate the candidates under a more realistic setup with multiple data streams, random deletion, and subsequent F2FS/SMR GC. We reuse the set of parameters as Table 3. In Figure 17, all candidates begin with a stable throughput of around 4GB/s. After reaching 80% capacity, random deletion starts, and then the GC kicks in. Recall our discussion in §3.2, CS-F2FS experiences a considerable performance drop due to frequent F2FS GC led by mixed data allocation. CS-Ext4 is hardly affected by the random deletion as Ext4 does not incur GC. Finally, CS-SMRSTORE continues to offer high throughput under random deletion. The main reason is that CS-SMRSTORE adopts several strategies to reduce the frequency and overhead of SMR GC.

Now, we take a closer look to understand the reason behind CS-SMRSTORE performance. In Figure 18, we plot the CDF of zone utilization under SMRSTORE. We can see that most zones are 100% used (i.e., the 100 on the X axis) and only a few zones are occupied with small chunks, thereby indicating less frequent SMR GC.

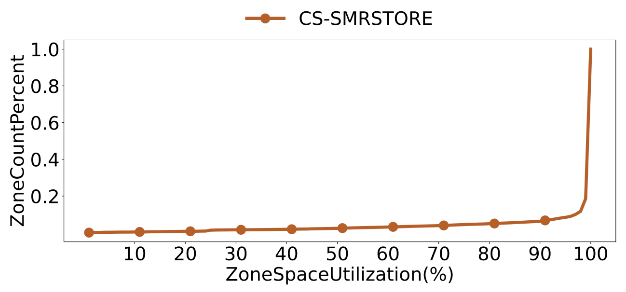

Figure 18: Zone utilizations (CDF) of CS-SMRSTORE (§6.2).

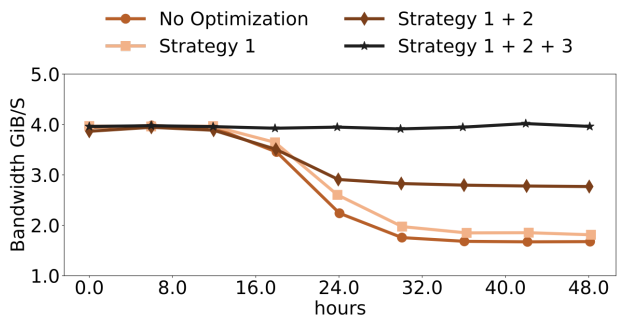

Figure 19: Throughput with different data placement strategies (§6.3). *No optimization: no separating, 64MB chunk size, random allocation on 55 opened zones. Strategy 1: separating streams by types. Strategy 2: adapting chunk size limit for datazone. Strategy 3: zone pool & round-robin allocation.*

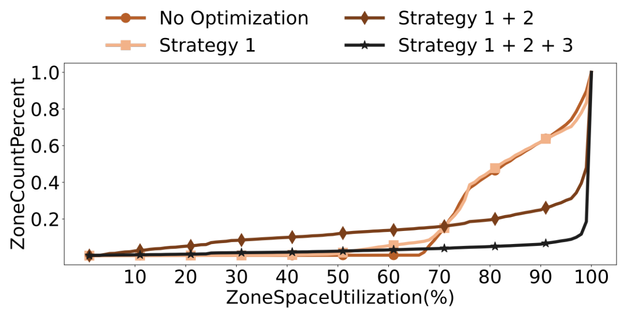

Figure 20: Zone Space Utilizations (CDF) Comparison (§6.3). *The results show that* SMRSTORE *can maintain a high space efficiency by enabling three end-to-end data placement strategies.*

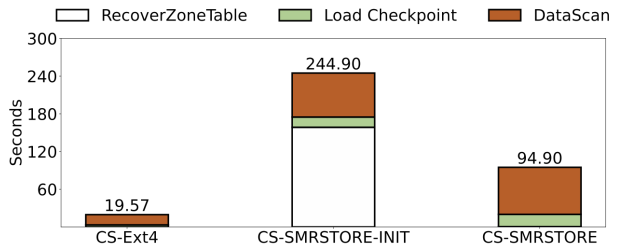

Figure 21: Recovery Performance (§6.4). *The figure shows the breakdown of recovery time. CS-*SMRSTORE*-INIT refers to the initial version of* SMRSTORE *without fixed metazone partition. Recovering zone table refers to the step "Identifying the latest valid checkpoint (§5.6)". Replaying journals is negligible and not shown.*

6.3 Effectiveness of Placement Strategy

The concentrated distribution of high utilization zones is a joint effort of different data placement strategies. Figure 19 shows the various combinations of individual strategies and corresponding effectiveness on write throughput. Here, we run the same multi-stream experiment with four different combinations.

From Figure 19, when only 'separating streams by types' is enabled, the SMR GC overhead is quite obvious and the performance is close to that of no optimization on allocation. Moreover, 'adapting chunk size limit' or 'zone pool & round-robin allocation' each contributes around half of the speedup and such phenomenons are also reflected by the zone utilizations CDFs in Figure 20.

6.4 Recovery Performance

In this experiment, we measure the time consumption in the recovery of a 20TB SMR drive with 60% capacity occupied. Figure 21 shows that CS-Ext4 with a 16TB CMR drive costs less than 20 seconds. CS-Ext4 only has two steps in recovery, loading checkpoint (which takes 3.27 seconds) and data scanning (which takes 16.3 seconds). CS-SMRSTORE completes the recovery with 94.4 seconds which takes around 19 seconds to load the checkpoint, less than 1 second to replay a few journals, and the remaining 75 seconds are for scanning the previously opened zones.

Note we also include a previous implementation, the CS-SMRSTORE-INIT which takes more than 4 minutes to recover. The major reason is that in this version, the on-disk layout is dynamic, meaning the metazones and datazones can be interleaved. As a result, SMRSTORE needs to scan all zone headers (both metazone and datazone) for recovery. Therefore, we switch to static zone allocation.

6.5 Resource Consumption

Memory. In a single server (60 HDDs and 2 SSD caches), the CS-SMRSTORE occupies 49.3GB of memory, around two times more than CS-Ext4. Memory growth is mainly contributed by the in-memory data structures of SMRSTORE. Specifically, the metadata of each chunk occupies around 200 bytes, and each record index in memory needs 8 bytes. The record indexes can be further compressed and we decide not to discuss in this paper due to space limit.

CPU. The CS-SMRSTORE uses around 19 cores which are 26.7% more than CS-EXT4. We use 8 cores for 8 partitions of the two cache SSDs (polling with spdk). We use another 4 cores for user-space network threads. SMRSTORE uses another 7 cores for processing requests, memory copy, checksum calculation, and background GC tasks of 60 SMR drives. With increasing areal density and comparable performance,

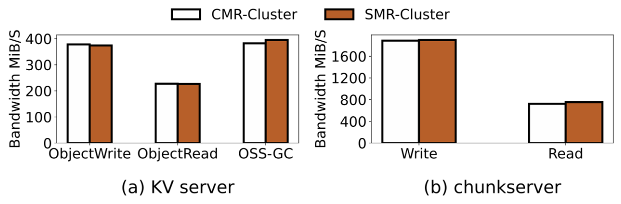

Figure 22: Performance comparison in OSS benchmark (§6.6). *Figure(a) compares key metrics of KV servers, including throughput of object write, object read and* OSS *GC. Figure(b) compares the corresponding read and write throughput of chunkservers.*

the extra overhead on CPU and memory usage is acceptable.

Space efficiency. Apart from persisting data, SMRSTORE further requires extra space for record headers, record paddings, and slice footers. For large IOs (512KB-1MB)—a common scenario in SMRSTORE (i.e., OSS GC/data stream, see Table 1)—SMRSTORE requires another 1-2% space of the IO size. The percentage increases for smaller writes but they are rather uncommon for HDDs due to IO merging in cache SSDs.

6.6 Field Deployment

In the OSS full stack benchmark, all of the key metrics in the SMR cluster are on par with the CMR cluster. The two clusters are both deployed with 13 KV store servers, 13 chunkservers, and 780 HDDs in total. Figure 22 shows that, at OSS service layer, each KV server in the SMR cluster achieves 374.2MB/s object write throughput, 227.7MB/s object read throughput, and 394.8MB/s OSS GC throughput. Each chunkserver in the SMR cluster provides 1898.6MB/s write throughput and 752.8MB/s read throughput. Similarly, in the CMR cluster, each chunkserver provides 1888.3MB/s write throughput and 723MB/s read throughput. This suggests, from an end-to-end perspective, we are able to replace CMR drives in standard-class OSS with SMR drives with no performance penalty thanks to SMRSTORE.

7 Limitation & Future Work

CZone. SMRSTORE follows a strictly log-structured design and thus does not require random writes support from czones. The use of the czones is under discussion. We could use czones as szones by maintaining a writer pointer in memory and a sequence number for each czone. The sequence number is used to identify valid records when the czone is reused.

Ad hoc to Alibaba standard OSS. At the moment, SMRSTORE is dedicated to serve standard-class OSS in Alibaba Cloud. However, SMRSTORE can easily adopt other zoned block devices , such as ZNS SSD. In fact, adapting SMRSTORE to ZNS SSD devices is in progress and will serve other services (e.g., Alibaba EBS).

Garbage collection. The expected on-disk lifespans of OSS data, OSS metadata and OSS GC are different from one OSS cluster to another. Certain clusters can have regular patterns on object creations and deletions while others perform more randomly. Currently, we are exploring more efficient SMR GC algorithms to better serve a variety of OSS workloads based on the accumulated statistics.

8 Related Work

Enabling HM-SMR drives. There are mainly three fashions of solutions in enabling HM-SMR, including adding a shim layer between the host and the ZBD subsystem [20,21], adopting local file systems to provide support [16,17], and modifying applications to efficiently utilize SMR devices [19,26,32]. SMRSTORE differs from above from two aspects. First, SMRSTORE completely discards random write by building everything as logs and hence avoid the potential constraints led by using the limited conventional zones or the tax imposed by random-to-sequential translation. Second, SMRSTORE significantly minimizes SMR GC overhead by end-to-end data placement strategies with the guidance of workloads.

Storage engine designs. To avoid the indirect overheads of general-purpose file systems [17,27], storage engines of cloud storage systems [18] and distributed file systems [31]) tend to evolve towards to user space, special purposed [9], and end-to-end integration [11,32]. SMRSTORE follows and further explores this path by building in the user space and implementing the semantics of PANGU chunks, which is much simpler than general file semantics (e.g., directory operations, file hardlink). Further, the range of the end-to-end integration in SMRSTORE is much wider than host-device, which includes OSS service layer, PANGU distributed file system layer, the storage engine persistence layer, and a novel but backward-incompatible device (i.e., HM-SMR drive). The results of SMRSTORE showcase the benefits can inspire future storage system designs under similar circumstances.

9 Conclusion

This paper describes our efforts in understanding, designing, evaluating, and deploying HM-SMR disks for standard-class OSS in Alibaba. By directly bridging the semantics between PANGU and HM-SMR zoned namespace, enforcing an all-logs layout and adopting guided placement strategies, SMRSTORE achieves our goal by deploying HM-SMR drives in standard-class OSS and providing comparable performance against CMR disks yet with much better cost efficiency.

Acknowledgments

The authors thank our shepherd Prof. Peter Desnoyers and anonymous reviewers for their meticulous reviews and insightful suggestions. We also thank the OSS and the PANGU team for their tremendous support on the SMRSTORE project and this paper. We sincerely thank Yikang Xu who pioneered the SMRSTORE prototype development. This research was partly supported by Alibaba AIR program and NSFC(62102424).

References

[1] Archival-class OSS on Alibaba Cloud. https://www.alibabacloud.com/solutions/backup_archive.

[2] Data Lake on Alibaba Cloud. https://www.alibabacloud.com/solutions/data-lake.

[3] Fio. https://github.com/axboe/fio.

[4] hdparm. https://www.man7.org/linux/man-pages/man8/hdparm.8.html.

[5] Shingled Magnetic Recording. https://zonedstorage.io/docs/introduction/smr.

[6] INCITS T13 Technical Committee. Information technology - Zoned Device ATA Command Set (ZAC). Draft Standard T13/BSR INCITS 537, 2015.

[7] INCITS T10 Technical Committee. Information technology-Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, 2017.

[8] A. Aghayev and P. Desnoyers. Skylight—A window on shingled disk operation. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[9] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[11] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[12] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so. Disks for Data Centers. https://research.google/pubs/pub44830.pdf, 2016.

[13] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[14] T. R. Feldman and G. A. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *Usenix Magazine*, 2013.

[15] G. Gibson and G. Ganger. Principles of operation for shingled disk devices. *Canregie Mellon Parallel Data Laboratory, CMU-PDL-11-107*, 2011.

[16] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim. HiSMRfs: A high performance file system for shingled storage array. In *Proceedings of 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.

[17] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[18] Q. Luo. Implement object storage with smr based key-value store. In *Proceedings of Storage Developer Conference (SDC)*, 2015.

[19] P. Macko, X. Ge, J. Haskins, J. Kelley, D. Slik, K. A. Smith, and M. G. Smith. SMORE: A Cold Data Object Store for SMR Drives (Extended Version). https://arxiv.org/abs/1705.09701, 2017.

[20] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic. ZEA, a data management approach for SMR. In *Proceedings of 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[21] D. L. Moal. dm-zoned: Zoned Block Device device mapper. https://lwn.net/Articles/714387/, 2017.

[22] D. L. Moal. Linux SMR Support Status. https://events.static.linuxfound.org/sites/events/files/slides/lemoal-Linux-SMR-vault-2017.pdf, 2017.

[23] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm BLOB storage system. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[24] G. Oh, J. Yang, and S. Ahn. Efficient Key-Value Data Placement for ZNS SSD. *Applied Sciences*, 2021.

[25] Z. Pang, Q. Lu, S. Chen, R. Wang, Y. Xu, and J. Wu. ArkDB: A Key-Value Engine for Scalable Cloud Storage Services. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021.

[26] R. Pitchumani, J. Hughes, and E. L. Miller. SMRDB: Key-Value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)*, 2015.

[27] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 2013.

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 1992.

[29] A. Suresh, G. A. Gibson, and G. R. Ganger. Shingled Magnetic Recording for Big Data Applications. Technical Report CMU-PDL-11-107, 2012.

[30] Q. Wang, J. Li, P. P. C. Lee, T. Ouyang, C. Shi, and L. Huang. Separating data via block invalidation time inference for write amplification reduction in Log-Structured storage. In *Proceedings of 20th USENIX Conference on File and Storage Technologies (FAST)*, 2022.

[31] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, High-Performance distributed file system. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[32] T. Yao, J. Wan, P. Huang, Y. Zhang, Z. Liu, C. Xie, and X. He. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.