SMORE: A Cold Data Object Store for SMR Drives

Peter Macko, Xiongzi Ge, John Haskins, Jr.*, James Kelley, David Slik, Keith A. Smith, and Maxim G. Smith
NetApp, Inc., *Qualcomm
peter.macko@netapp.com, james.kelley@netapp.com, keith.smith@netapp.com

Abstract—**Shingled magnetic recording (SMR) increases the capacity of magnetic hard drives, but it requires that each zone of a disk be written sequentially and erased in bulk. This makes SMR a good fit for workloads dominated by large data objects with limited churn. To explore this possibility, we have developed SMORE, an object storage system designed to reliably and efficiently store large, seldom changing data objects on an array of host-managed or host-aware SMR disks.**

SMORE uses a log-structured approach to accommodate the constraint that all writes to an SMR drive must be sequential within large shingled zones. It stripes data across zones on separate disks, using erasure coding to protect against drive failure. A separate garbage collection thread reclaims space by migrating live data out of the emptiest zones so that they can be trimmed and reused. An index stored on flash and backed up to the SMR drives maps object identifiers to on-disk locations. SMORE interleaves log records with object data within SMR zones to enable index recovery after a system crash (or failure of the flash device) without any additional logging mechanism.

SMORE achieves full disk bandwidth when ingesting data—with a variety of object sizes—and when reading large objects. Read performance declines for smaller object sizes where inter-object seek time dominates. With a worst-case pattern of random deletions, SMORE has a write amplification (not counting RAID parity) of less than 2.0 at 80% occupancy. By taking an index snapshot every two hours, SMORE recovers from crashes in less than a minute. More frequent snapshots allow faster recovery.

I. Introduction

Shingled magnetic recording (SMR) technology [1] provides the next major capacity increase for hard disk drives. Drive vendors have already shipped millions of SMR drives. Current SMR drives provide about 25% more capacity than conventional magnetic recording (CMR). The SMR advantage is expected to increase over time [2], making SMR a compelling technology for high-capacity storage.

In addition to increasing areal bit density, SMR drives introduce several challenges for storage software and applications. The most significant challenge is that SMR does not permit random writes. SMR drives are divided into large multi-megabyte zones that must be written sequentially. To overwrite any part of a zone, the entire zone must be logically erased and then rewritten from the beginning.

There are several ways of supporting SMR's sequential write requirement. One is to build a block translation layer, similar to the flash translation layer in an SSD, but this approach has limited access to higher level information for optimizing its use of the SMR. Another approach is to build a file system, but state-of-the art file systems are highly complex; while it is feasible to quickly prototype a proof of concept, commercial-quality file systems take years to develop and mature to the point where they are stable, reliable, and performant [3], [4].

We have opted for a third approach. Rather than developing a general-purpose storage system, our goal is to target a workload that is well suited to SMR drives—storing cold or cool objects, while frequently accessed objects are cached or tiered in high-performance storage.

For example, typical media files are read and written sequentially and range in size from a few MB to many GB or TB, which fits well with the excellent sequential throughput and low cost of SMR drives. Media storage is important in many wide-spread use cases, including entertainment, medical imaging, surveillance, etc. Such media use cases already account for a significant fraction of new data, a trend that is expected to continue in the future [5].

The resulting storage system, our SMR Object REpository (SMORE), targets this workload. While we anticipate ample demand for affordable solutions targeting the bulk storage of media data, SMORE is also applicable to other use cases that can benefit from low-cost storage for large objects, including backups, virtual machine image libraries, and others.

SMORE is designed to provide the full bandwidth of the underlying SMR drives for streaming read and write access. Although it will accept small objects, the performance for this type of storage has not been optimized. Finally, because we anticipate seldom changing data, the garbage collection overhead resulting from SMR write restrictions has only a modest impact on SMORE's overall performance.

SMORE fills SMR zones sequentially, erasure coding data across zones on separate drives for reliability. As the client deletes objects, SMORE uses garbage collection to migrate live data from partially empty zones. A working copy of object metadata is stored in an index on a cheap flash device. SMORE employs several techniques to optimize for the needs and limitations of SMR drives, such as interleaving a journal for crash recovery in the sequential stream of object writes.

The contributions of this work are:

- A recovery-oriented object store design, in which the disks remain on-seek during most writes.
- Decreasing the metadata overhead by managing disks at the granularity of zone sets, which are groups of SMR zones from different spindles.
- A system for efficient cold object storage on SMR drives.
- A rigorous evaluation of the resulting design using recent SMR drives, measuring write amplification and recovery costs as well as basic system performance.

* Work performed while at NetApp, Inc.

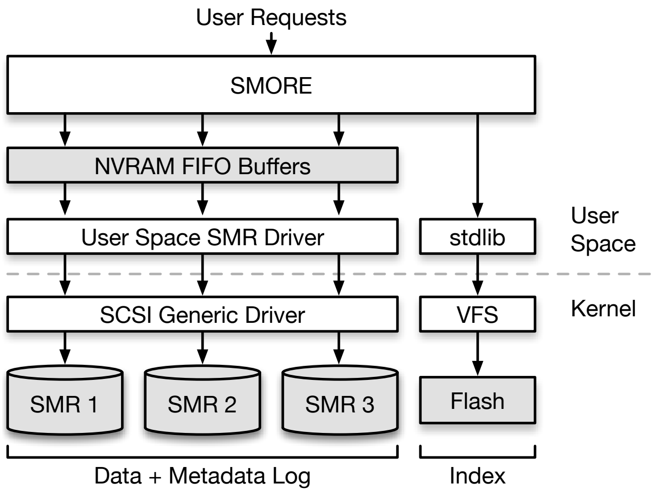

Fig. 1: **A high-level overview of the SMORE architecture.** SMORE splits incoming data into segments and erasure-codes each across zones from multiple SMR drives. Each drive is optionally front-ended with a small NVRAM-backed FIFO buffer that coalesces small writes. SMORE stores an index on a flash device.

II. Architecture

This section is an overview of the SMORE architecture. A detailed description of SMORE is available in a separate report [6].

At the high level, as illustrated in Figure 1, SMORE writes data and metadata in a log-structured format, erasure-coded across multiple SMR drives, and uses a flash device to store the index that maps object IDs to their physical locations. We optionally front-end each drive with a small buffer (a few MB in size) in battery-backed RAM for coalescing small writes, which improves performance and space utilization. Any kind of NVRAM will suffice for buffering, but NVRAM technologies with limited write endurance (e.g., PCM) will require extra capacity for wear-leveling.

SMORE uses a log-structured design because it is well suited to the append-only nature of SMR drives. Like a log-structured file system [7], SMORE divides storage into large, contiguous regions that it fills sequentially. In SMORE, these regions are called *zone sets*. When SMORE needs more free space, it garbage collects partially empty zone sets and relocates the live data. Unlike log-structured file systems, however, SMORE is an object store and runs on an array of SMR drives. This leads to a different design.

We also store the superblock in a log-structured manner in a small number of *superblock zones* that are set aside on each disk, usually just one per disk. When SMORE writes a superblock, it timestamps it and replicates it to three different disks by default. When a superblock zone is full and there exists a more recent superblock elsewhere, SMORE trims the zone and reuses it. During recovery, SMORE examines the superblock zones to find the most recent superblock, which it uses to bootstrap the rest of the recovery process.

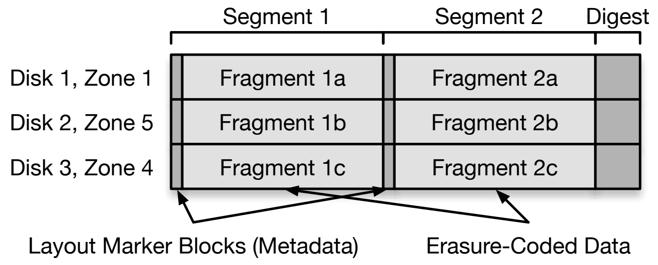

Fig. 2: **Anatomy of a zone set.** A zone set is an arbitrary set of zones from different SMR drives. SMORE chunks each object into equal-sized segments, erasure-codes each segment across the multiple zones, and writes them to the zone set together with headers called the *layout marker blocks* (LMBs). When a zone set becomes full, SMORE finishes it by writing a digest with the summary of the segments it holds.

A. The Building Blocks: Zone Sets, Segments, and Index

A zone set is a group of zones, each from a different drive, that form an append-only container in which SMORE writes data. The zones in a set are always filled in parallel with equal amounts of data striped across them, encoded by a data protection scheme specified at system initialization. SMORE spreads data evenly across the zones in a zone set so that their write pointers advance together. At any time, SMORE has one or more zone sets *open* to receive new data.

Figure 2 shows the anatomy of a zone set. SMORE chunks incoming objects into *segments* and writes each segment to one of the open zone sets. SMORE divides each segment into equal-sized *fragments* and computes additional parity fragments so that the total number of data and parity fragments matches the number of drives in a zone set. SMORE writes each fragment to one of the zones in the zone set, starting with a header, the *layout marker block* (LMB), which describes the segment and is used for error detection and recovery.

SMORE keeps track of all live segments (those that belong to live objects) in an index, which allows it to efficiently look up segment locations. Each segment is described with its zone set ID and the offset within that zone set. We address segments using zone sets instead of the individual physical zones, because this approach significantly decreases the size of the index, and it enables SMORE to recover from a failed disk by rebuilding the contents of the lost zones into any vacant zones on the remaining disks, similarly to parity declustered RAID [8].

The mapping of zone set IDs to the physical locations of the zones is stored in the superblock. The index is cached in RAM and backed up by files on the system's flash. The files are updated asynchronously.

The segment is the basic unit of allocation and layout and is typically a few tens of megabytes. An object is made accessible for reads only after its last segment is written. SMORE marks the last segment with a special bit in its layout marker block and the corresponding index entry. Old versions of objects and incomplete objects, such as those that were left unfinished by failed clients, are eventually garbage collected.

Segments provide several benefits. They reduce memory pressure by allowing SMORE to start writing an object to disk before the entire object is in memory. Likewise, they let SMORE handle objects that are too large to fit in a single zone set. Segments also ensure sequential on-disk layout by avoiding fine-grained interleaving when writing several objects concurrently. Finally, large segments minimize the amount of metadata (i.e., index entries) required for each object.

Each zone in an opened zone set can be optionally front-ended with a small NVRAM-backed *FIFO buffer,* which allows the system to efficiently pack small objects even in the presence of large physical blocks (which could possibly reach 32KB or larger in the future [9]) and optimize write performance. Fragments and FIFO buffers are sized so that the system typically reads and writes 2 to 4 disk tracks at a time, amortizing the cost of each seek across a large data transfer.

SMORE deletes an object by removing the object's entries from the index. It also writes a *tombstone* to an open zone set as a persistent record of the deletion. Tombstones are processed while recovering the index after a failure. SMORE's garbage collector reclaims the space occupied by deleted and overwritten objects in the background.

B. Recovery-Oriented Design

SMORE follows a *recovery-oriented design*. By designing for fast and simple recovery, we can use SMORE's recovery logic in place of more complex consistency mechanisms. There are a variety of failures that could damage the index. SMORE handles all of these scenarios with a single recovery mechanism—replaying updates based on the layout marker blocks intermingled with object data in zone sets. Using the same logic for multiple failure scenarios ensures better testing of critical recovery code. It also avoids the overhead and complexity of implementing different mechanisms to handle different faults.

SMORE periodically checkpoints the index, storing a copy in dedicated zone sets on the SMR drives. In the event of a failure, it reads the most recent checkpoint and updates it by scanning and processing all layout marker blocks written since the last checkpoint. As an optimization, SMORE writes a *digest* of all layout marker blocks as the last entry in each zone set. During recovery, SMORE can read this digest in one I/O operation instead of scanning the entire zone set.

We limit the number of zone sets that need to be examined during recovery by examining only those that could have been appended to since the last index checkpoint. Whether a zone set could have been written to is implied by its state:

- *Empty:* An empty zone set
- *Available:* An empty zone set that can be opened
- *Open:* A zone set that can receive writes
- *Closed:* A full zone set (does not accept any more writes)
- *Indexed:* A zone set that was *closed* at the time of an index checkpoint
- *Index:* A dedicated zone set for storing an index snapshot

We keep track of these states in the superblock, but update the superblock only when we transition zones from *empty* to *available* and from *closed* to *indexed,* which are both bulk operations. This makes the overhead of superblock updates negligible. We intentionally keep the number of *opened* zone sets low to reduce seeks and maximize the disk throughput for streaming writes. We thus do not transition zone sets directly from *empty* to *opened,* which would require the superblock to be updated frequently. Only the zone sets with states *available, open,* and *closed* need to be examined during recovery.

We do not update the superblock when trimming zone sets, we simply check the write pointers of all *open, closed,* and *indexed* zone sets during recovery and set their states to *empty* if they are trimmed. This is a very quick operation because the disks allow us to read the positions of all write pointers using a single command.

C. Garbage Collection (Cleaning)

When an object is deleted, its space is not immediately reclaimed, because those fragments became read-only once they were written into the sequential zones. A zone set containing deleted data is said to be *dirty*. Eventually space is reclaimed from dirty zone sets by moving any live data into a new zone set, then trimming the old zones. This cleaning may be done on demand when more space is needed in the system or as a background task concurrent with normal client operation. Superblock and index snapshot zones are trimmed during normal operation and do not need cleaning.

SMORE uses a greedy strategy by always cleaning the zone set with the most dead space. Once a zone set is selected for cleaning, all of the live data is relocated to another zone set and only the tombstones that are *newer* than the most recent index snapshot are relocated. As the data and tombstones are relocated, the index is updated accordingly. After all valid items have been copied and indexed anew, the zones of the old zone set are trimmed and made available for writing new content.

If the node crashes during garbage collection, garbage collection can be begun anew after a reboot, starting from any zone set. Any incompletely cleaned zone set will eventually be selected again for cleaning. Content in the zone set that was cleaned previously will be found to be invalid at that time and discarded, while any still-valid content will be relocated.

III. Evaluation

SMORE is implemented as a library in approximately 19,500 lines of C++ (excluding tests and utilities). We used `libzbc` [10] to interface with SMR drives. The SMORE library presents an object-based read/write API (PUT, GET, and DELETE) and can be linked into a higher-level storage service, such as OpenStack Swift, to provide a complete solution.

We designed SMORE to support a cool storage tier for large media objects. Thus, our evaluation focuses on quantifying SMORE's performance under different aspects of this workload.

A. Test Platform

Our test platforms uses six HGST Ultrastar Archive Ha10 drives. These are 10TB host-managed SMR drives with 256MB zones. According to our measurements, the average read

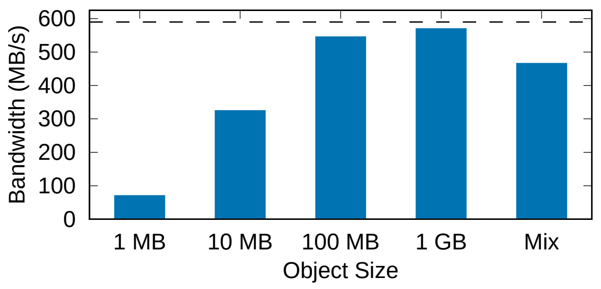

Fig. 3: **Read bandwidth as a function of object size.** The best possible read performance is 590MB/s (the dashed line). SMORE achieves near-optimal read performance for large objects.

performance is 118MB/s across all zones (with peak 150MB/s at the outer diameter) and the average write performance is 55MB/s (with 65MB/s at the outer diameter). The write bandwidth is lower than the read bandwidth because after the drive writes a track, it verifies the correctness of the previous track [11].

We restrict the capacity of the drives by using only every 60th zone. This limits the overall system capacity enough to make the duration of our benchmarks manageable while preserving the full seek profile of the drives. We configure our zone sets for 5+1 parity. The resulting total system capacity is 766GB. We verified that our results are representative by comparing them to the results of select test cases that we ran with the system's full 50TB capacity.

The drives are connected to a server with 32 Intel Xeon 2.90GHz cores and 128GB RAM. SMORE uses direct I/O, bypassing any buffering in kernel, so that our results are not skewed by the large amount of main memory in our system. We ran our workloads using six threads unless noted otherwise.

B. *Workload Generator*

We generate our workloads with two different distributions of object sizes: (1) workloads in which all objects have the same size, ranging from 1MB to 1GB, and (2) workloads with object sizes that follow a truncated log-normal distribution modeled after the file sizes in the cold storage system of the European Center for Medium-Range Weather Forecasts [12], which is representative of the types of workloads we expect SMORE to be used for. The peak of the distribution is around 128MB, with a majority of objects between 16MB and 512MB. We truncate the distribution to omit objects less than 1MB in size, because we assume that small objects will be stored further up in the storage hierarchy or coalesced into larger objects before being stored in SMORE.

C. *Ingest Performance*

To measure ingest performance, we look at workloads consisting of 100% PUT operations until the systems fills up, and we vary the object sizes from 1MB to 1GB and the number of threads from 3 to 24. SMORE ingests data at approximately 280MB/s regardless of object sizes and the number of threads, which is almost exactly 100% of the maximum write bandwidth allowed by our SMR drives. As long as there is any input data available, SMORE streams it sequentially onto the SMR drives, thus ensuring maximum possible performance. The performance does not depend on object size because per-object overheads are negligible. It does not depend on the number of threads because with only six drives our system quickly becomes disk limited.

D. *Object Retrieval Performance*

To evaluate read performance, we filled our test system to 80% of its capacity and read objects at random in their entirety. Reading random objects results in the worst-case behavior.

Figure 3 shows the aggregate read bandwidth across all client threads as a function of object size. The best possible read performance allowed by our disks is 5×118 = 590MB/s (the dashed line in the figure). SMORE achieves near-optimal read performance for large objects, but the read performance of small objects is dominated by seeks.

The read performance remains constant as the system ages. For example, we took the system with a mixture of object sizes and aged it by deleting and creating new objects until we wrote more bytes than 500% of the capacity of the system, which is a higher churn than we expect for cold data. We then measured the read performance again by reading random objects, and the resultant aggregate bandwidth was within the margin of error of the read performance measured on an unaged system.

SMORE achieves good read performance because it attempts to keep segments from a single object close together. By default, it schedules writes to zone sets so that a single writer can write 12 segments of data (240MB, or 48MB per drive) from a single object before switching to a different writer. The garbage collector then does a best effort to keep the fragments together. The lowered concurrency for writes is practically unnoticeable in large object workloads, while the read gains are significant.

To quantify this gain in read performance, we repeated our benchmarks with this feature disabled, so that segments from different objects are more interleaved. The read performance caps at 390MB/s, compared to 570MB/s from our original benchmarks.

E. *Write Amplification*

We measure the write amplification by repeating our benchmark on a system that already contains data. We delete objects at random as fast as new objects arrive, which provides the worst-case measurement of the garbage collection overhead, but we expect the deletes to be at least weakly correlated in practice. We run three sets of benchmarks, each measuring the write amplification while maintaining different proportions of live data in the system: 70%, 80%, and 90%.

Figure 4 summarizes our results. Note that the best write amplification we can achieve is 1.2, due to our 5+1 zone set parity configuration (represented by the dashed line). SMORE achieves good write amplification, especially for 70% utilization, and even for 80% utilization with large objects.

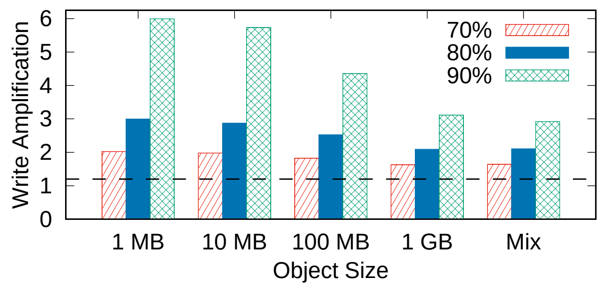

Fig. 4: **Write amplification as a function of object size and system utilization.** The theoretically best write amplification is 1.2 (the dashed line), given our 5+1 zone set configuration. SMORE achieves good write amplification, especially for 70% utilization, and 80% utilization for larger object sizes.

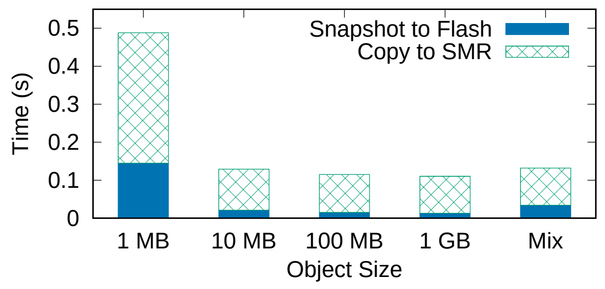

Fig. 5: **Time to create a snapshot vs. object sizes** for a system with 80% occupancy.

Write amplification is particularly high for large utilization levels and small object sizes, because objects are deleted at random. As objects get smaller, there is less variance in the amount of dead data per zone set. As a result, the greedy garbage collector has to copy more live data when cleaning zone sets. For example, deleting a single 1GB object typically results in a lot of dead data in a few zone sets. Deleting an equivalent amount of data in randomly selected 1MB objects results in a smattering of dead data in a large number of zone sets.

F. *Recovery Performance*

In SMORE, recovery consists of two phases: reading the most recent index snapshot, and then updating it from the zone digests and layout marker blocks in the recently updated zone sets. We can tune SMORE for faster recovery by taking more frequent index snapshots.

1) Creating Snapshots: Considering that SMORE is tuned for large objects, the overhead of snapshots is not significant. Figure 5 shows the time it takes to create an index snapshot for an 80% full system for various object sizes. The plot shows both the time it takes to create a snapshot just on the flash and the additional time it takes to copy it to the SMR drives. The time to create a snapshot depends primarily on the number of segments stored in the object store but not on the time since the last snapshot, since our snapshots are not incremental.

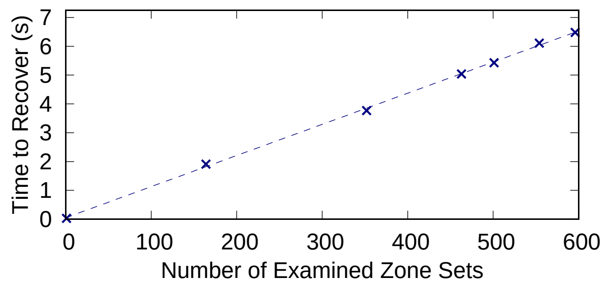

Fig. 6: **Recovery time vs. the number of examined zone sets** for a workload with mixed-size objects, when restoring from a snapshot on the flash device. (The crashing mechanism itself was approximate and nondeterministic as to when exactly it caused a crash. Therefore there is uneven spacing of data points.)

For all workloads except for the pure 1MB objects, it takes less than 0.15 seconds to create a snapshot on our test system. After extrapolating out the result to a 50TB system with mixed object sizes, we see that creating a snapshot would take approximately 1.5 to 1.6 seconds.

Copying to the SMR involves simply a single seek and sequential write of the snapshot, because snapshots are stored in dedicated zone sets, and in the vast majority of cases, the index fits inside a single zone set. In the case of the mixed workload, the index is less than 1MB in size. This would fit into a single zone set even when extrapolated to a 50TB system. On the other hand, because copying to SMR is asynchronous, the actual time to perform the snapshot might be longer, depending on the foreground workload.

2) Recovery: Figure 6 shows the time it takes to recover SMORE for a workload with a mix of object sizes as a number of examined zone sets during recovery. The time from the most recent snapshot spans from zero to 4.5 hours. In this benchmark, we took a snapshot after filling up the system to 80% of its capacity and then continued with a 100% PUT & DELETE workload, varying the amount of time we allowed the system to run until we crashed it.

This models the most common case, in which SMORE recovers starting from an index snapshot stored on the flash device, and only if that fails (which is very rare) it uses the index snapshot backup stored on the SMR drives. When we reran our recovery benchmark with an empty flash device, it took 0.28 seconds to copy the index snapshot from the SMR drives to the flash.

With zero zone sets to replay, we see the best-case time, where the most recent index snapshot is fully up to date. At the other extreme, when we recover every zone set (the last two data points in the plot), we see the worst-case performance. Seven seconds is thus the longest possible duration of recovery in our test system. When we need to recover every zone set, we do not need to start from an index snapshot.

As shown in Figure 6, the trend is linear with the number of examined zone sets ($R^2 > 0.999$). After extrapolating this to a full 50TB system, we see that it would take only 7 minutes to recover the index in the worst case.

The overhead of creating snapshots and the time it takes to recover can be balanced to meet a specific recovery time objective. For example, if the system needs to recover from a crash within 5 seconds, we need to take a snapshot approximately every 2 hours. If it takes less than 0.15 seconds to take a snapshot, then the overhead of snapshotting is only $2.1 \times 10^{-3}\%$. Even when extrapolated to a 50TB system with 1.5 second-long snapshots, the overhead of snapshots would be only 0.021%.

IV. Related Work

SMORE builds on a long history of write-optimized storage systems, dating back to the Sprite Log-Structured File System (LFS) [7]. Like LFS, SMORE writes all data sequentially to large disk regions, but it writes the data incrementally, uses layout marker blocks to enable recovery, and maintains a working copy of its metadata in flash.

Sawmill [13] extended LFS to work on a RAID array, and several LFS-inspired file systems, such as WAFL [14], ZFS [4], and btrfs [3], have integrated RAID functionality within the file system, leveraging the write coalescing behavior of LFS to avoid small update penalties in RAID, but often relaxing the sequential write requirements of LFS to achieve lower (or no) cleaning costs. SMORE also integrates RAID functionality, but maintains strict adherence to LFS-style writes due to the requirements of SMR.

A. SMR File Systems

Similar to SMORE, prior SMR file systems use log-structured storage, and they generally place the primary copy of their metadata on a random write device, such as an unshingled section of the SMR drive [15], [16], or on an SSD if it is available [17]. SMORE limits the amount of flash storage it uses, storing only a single copy of the metadata index on flash and using log records embedded in zone sets to provide recovery. HiSMRfs [17] also uses flash and spans multiple SMR drives, but it targets general-purpose workloads, which leads to a different design that requires mirrored SSDs for storing metadata and hot files.

Huawei's Key-Value Store (KVS) [18] uses recovery-oriented design like SMORE, but it is a single-disk system with erasure coding of objects across multiple drives handled higher in the storage stack. SMORE benefits from being designed as a multi-disk system from the ground up, which decreases the index size and simplifies data management and recovery.

SAFS [19] is a single-disk system optimized for append-only workloads that stores all incoming data into the same zone but separates the data later, which optimizes for sequential reads at the cost of rewriting all data. In contrast, SMORE uses segments to write multiple megabytes of data to an object without interleaving.

Kadekodi et al. [20] demonstrated that an SMR disk that permits the client to write anywhere enables building a file system with better performance and less frequent garbage collection. SMRDB [21] is a key-value store for database-like workloads based on a Log-Structured Merge (LSM) Tree.

In addition to developing custom file systems and object stores for SMR drives, there is an ongoing effort to adapt existing file systems such as `ext4` [22], `nilfs` [23], and `xfs` [24] to SMR drives.

B. Shingle Translation Layers (STLs)

Another method to incorporate SMR drives into a storage system is by using a shingle translation layer, similar to drive-managed SMR drives. The most common approaches write incoming data to one or more persistent caches and merge the data later with the original band [25], [26]. SMR disks that allow random writes to shingled zones enable STLs that take advantage of circular buffers [25], [27], [28] or managing data at the level of small, wedge-shaped regions [29]. Using drives with only a few tracks per zone enables efficient static address mapping schemes [30].

C. Other Related Work

Aghayev and Desnoyers [26] and Wu et al. [31] provide benchmark-based analysis of the behavior of commercial drive-managed and host-aware SMR drives, respectively.

Categorizing data based on hotness can significantly decrease write amplification on SMR drives [32], [33], [28], and has also been found helpful in F2FS [34].

Finally, there is a rich history of archival storage systems built using conventional hard drives. Some of this work describes complete systems [35], [36], [37]. Other researchers have focused on specific problems, such as data reduction [38], [39], power management [40], [41], or long-term data preservation [42], [43]. This research predates the introduction of SMR technology and does not address the unique requirements of SMR disks.

V. Conclusion

In this work, we presented SMORE, an object storage system designed to reliably and efficiently store large, seldom-changing data objects on an array of host-managed or host-aware SMR disks. Using a log-structured approach to write data to the large, append-only shingled zones, we were able to achieve full disk bandwidth when ingesting data—for a variety of object sizes—with only a moderate amount of system optimization. Moreover, SMORE achieves low write amplification during worst-case churn when the system is filled to 80% of its capacity. Finally, the recovery-oriented design of SMORE, specifically the interleaving of log records with object data, allows a simple and efficient recovery process in the event of a failure without any additional logging mechanism.

Acknowledgments

We would like to thank the anonymous FAST and MSST reviewers for their helpful comments on previous drafts of this paper. We also thank Al Andux, Tim Emami, Jeffrey Heller, and all of NetApp's Advanced Technology Group (ATG) for their advice, support, and suggestions throughout this project.

References

[1] R. Wood, M. Williams, A. Kavcic, and J. Miles, "The feasibility of magnetic recording at 10 terabits per square inch on conventional media," *IEEE Transactions on Magnetics*, vol. 45, no. 2, pp. 917–923, Feb. 2009.

[2] G. A. Gibson and G. Ganger, "Principles of operation for shingled disk devices," CMU Parallel Data Laboratory, Tech. Rep. CMU-PDL-11-107, Apr. 2011.

[3] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: the Linux B-tree filesystem," *ACM Transactions on Storage*, vol. 9, no. 3, pp. 9:1–9:32, Aug. 2013.

[4] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Addison Wesley, 2015, ch. 10.

[5] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," IDC iView Report, http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf, Dec. 2012.

[6] P. Macko, X. Ge, J. Haskins Jr., J. Kelley, D. Slik, K. A. Smith, and M. G. Smith, "SMORE: a cold data object store for SMR-drives (extended version)," *Computing Research Repository (CoRR), ArXiv.org*, 2017.

[7] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[8] M. Holland and G. A. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Systems*, ser. ASPLOS '92, Oct. 1992, pp. 23–35.

[9] J. Edge, "Handling 32kb-block drives," https://lwn.net/Articles/636978/, Mar. 2015.

[10] HGST, "ZBC device manipulation library," https://github.com/hgst/libzbc, Jan. 2016.

[11] ——, "Ultrastar Archive Ha10: Data sheet," https://www.hgst.com/sites/default/files/resources/Ha10-Ultrastar-HDD-DS.pdf, Aug. 2015.

[12] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth, "Analysis of the ECMWF storage landscape," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST '15, Feb. 2015, pp. 15–27.

[13] K. W. Shirriff and J. K. Ousterhout, "Sawmill: a high-bandwidth logging file system," in *Proceedings of the USENIX Summer 1994 Technical Conference*, Jun. 1994, pp. 125–136.

[14] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," in *Proceedings of the USENIX Winter 1994 Technical Conference*, Jan. 1994, pp. 235–246.

[15] D. LeMoal, Z. Bandic, and C. Guyot, "Shingled File System: Host-side management of shingled magnetic recording disks," in *2012 IEEE International Conference on Consumer Electronics*, ser. ICCE '12. IEEE Computer Society, Jan. 2012, pp. 425–426.

[16] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic, "ZEA, a data management approach for SMR." in *8th USENIX Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '16. USENIX Association, Jun. 2016.

[17] C. Jin, W. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim, "HiSMRfs: A high performance file system for shingled storage array." in *2014 IEEE 30th Symposium on Mass Storage Systems and Technologies*, ser. MSST '14. IEEE Computer Society, Jun. 2014, pp. 1–6.

[18] Q. Luo and L. Zhang, "Implement object storage with SMR based key-value store." in *2015 Storage Developer Conference*, ser. SDC '15. Storage Networking Industry Association, Sep. 2015.

[19] C.-Y. Ku and S. P. Morgan, "An SMR-aware append-only file system." in *2015 Storage Developer Conference*, ser. SDC '15. Storage Networking Industry Association, Sep. 2015.

[20] S. Kadekodi, S. Pimpale, and G. A. Gibson, "Caveat-scriptor: Write anywhere shingled disks." in *7th USENIX Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '15. USENIX Association, Jul. 2015.

[21] 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*, ser. SYSTOR '15. Association for Computing Machinery, May 2015, pp. 18:1–18:11.

[22] A. Palmer, "FS design around SMR: Seagate's journey with EXT4." in *2015 Storage Developer Conference*, ser. SDC '15. Storage Networking Industry Association, Sep. 2015.

[23] B. A. Dhas, E. Zadok, J. Borden, and J. Malina, "Evaluation of nilfs2 for shingled magnetic recording (SMR) disks," Stony Brook University, Tech. Rep. FSL-14-03, 2014.

[24] D. Chinner, "SMR layout optimisation for XFS," http://xfs.org/images/f/f6/Xfs-smr-structure-0.2.pdf, Mar. 2015.

[25] Y. Cassuto, M. A. A. Sanvido, C. Guyot, D. R. Hall, and Z. Bandic, "Indirection systems for shingled-recording disk drives." in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, ser. MSST '10. IEEE Computer Society, May 2010, pp. 1–14.

[26] A. Aghayev and P. Desnoyers, "Skylight – a window on shingled disk operation." in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST '15, Feb. 2015, pp. 135–149.

[27] D. Hall, J. H. Marcos, and J. D. Coker, "Data handling algorithms for autonomous shingled magnetic recording HDDs," *IEEE Transactions on Magnetics*, vol. 48, no. 5, pp. 1777–1781, May 2012.

[28] C.-I. Lin, D. Park, W. He, and D. H. C. Du, "H-SWD: Incorporating hot data identification into shingled write disks." in *MASCOTS*. IEEE Computer Society, Aug. 2012, pp. 321–330.

[29] J. Wan, N. Zhao, Y. Zhu, J. Wang, Y. Mao, P. Chen, and C. Xie, "High performance and high capacity hybrid shingled-recording disk system." in *2012 IEEE International Conference on Cluster Computing*, ser. CLUSTER '12. IEEE Computer Society, Sep. 2012, pp. 173–181.

[30] W. He and D. H. C. Du, "Novel address mappings for shingled write disks." in *6th USENIX Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '14. USENIX Association, Jun. 2014.

[31] F. Wu, M.-C. Yang, Z. Fan, B. Zhang, X. Ge, and D. H. Du, "Evaluating host aware SMR drives," in *8th USENIX Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '16. USENIX Association, Jun. 2016.

[32] A. Amer, D. D. E. Long, E. L. Miller, J.-F. Pris, and S. J. T. Schwarz, "Design issues for a shingled write disk system." in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, ser. MSST '10. IEEE Computer Society, May 2010, pp. 1–12.

[33] S. Jones, A. Amer, E. L. Miller, D. D. E. Long, R. Pitchumani, and C. R. Strong, "Classifying data to reduce long term data movement in shingled write disks." in *2015 IEEE 31st Symposium on Mass Storage Systems and Technologies*, ser. MSST '15. IEEE Computer Society, May 2015, pp. 1–9.

[34] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST '15, Feb. 2015, pp. 273–286.

[35] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler, "Deconstructing commodity storage clusters," in *31st International Symposium on Computer Architecture*, ser. ISCA '05. Association for Computing Machinery, Jun. 2005, pp. 60–71.

[36] L. L. You, K. T. Pollack, and D. D. E. Long, "Deep Store: An archival storage system architecture," in *21st IEEE International Conference on Data Engineering*, ser. ICDE '05. IEEE Computer Society, Apr. 2005.

[37] 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 the 11th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '14. USENIX Association, Oct. 2014, pp. 383–398.

[38] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02, Jan. 2002, pp. 89–101.

[39] L. L. You, K. T. Pollack, and D. D. E. Long, "PRESIDIO: A framework for efficient archival data storage," *ACM Transactions on Storage*, vol. 7, no. 2, pp. 6:1–6:60, Jul. 2011.

[40] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, "Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST '08, Feb. 2008, pp. 1–16.

[41] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron, "Pelican: A building block for exascale cold data storage," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '14. USENIX Association, Oct. 2014, pp. 351–365.

[42] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker, "The LOCKSS peer-to-peer digital preservation system," *ACM Transactions on Computer Systems*, vol. 23, no. 1, pp. 2–50, Feb. 2005.

[43] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, "POTSHARDS—a secure, recoverable, long-term archival storage system," *ACM Transactions on Storage*, vol. 5, no. 2, pp. 5:1–5:35, Jun. 2009.