Reconciling LSM-Trees with Modern Hard Drives using BlueFS

Abutalib Aghayev, Sage Weil (Red Hat Inc.), Greg Ganger, George Amvrosiadis

CMU-PDL-19-102
April 2019

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

LSM-Trees have become a popular building block in large-scale storage systems where hard drives are the dominant storage medium. Meanwhile, drive makers are shifting to Shingled Magnetic Recording (SMR), a recording technique that increases drive capacity by ≈25% but also works best with a new, backward-incompatible device interface. Large-scale cloud storage providers are updating their proprietary software stacks to utilize SMR drives, but widespread adoption requires more general-purpose support. This paper introduces BlueFS, an open-source user-space file system that allows widely-used LSM-Tree implementations to utilize SMR drives with zero overhead and no code changes. BlueFS's design aggressively specializes data placement and I/O sizes to exposed SMR drive parameters, while hiding those details. As a result, for example, unmodified RocksDB performs random inserts 64% faster atop BlueFS than atop XFS, when storing data on an SMR drive. In addition, LevelDB running on BlueFS is 2–20× faster than GearDB, a recent key-value store designed for SMR drives.

Acknowledgements: We thank the members and companies of the PDL Consortium (Alibaba Group, Amazon, Datrium, Dell EMC, Facebook, Google, Hewlett Packard Enterprise, Hitachi, IBM Research, Intel, Micron, Microsoft Research, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Veritas and Western Digital) for their interest, insights, feedback, and support.

Keywords: Shingled Magnetic Recording, SMR, HM-SMR, LSM-Tree, file systems

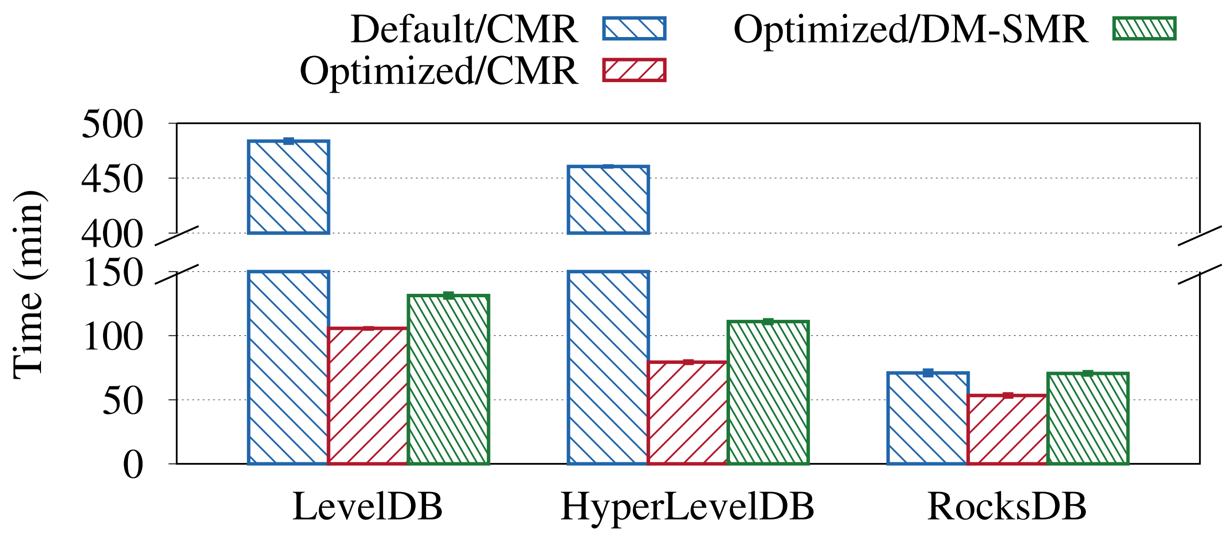

Figure 1: Time taken to insert 150 million key-value pairs in popular LSM-Tree-based key-value stores running on CMR and DM-SMR drives. The I/O is not sufficiently sequential, causing 20-40% slowdown on DM-SMR drives even when key-value stores' configurations are optimized for hard drive.

1 Introduction

The Log-Structured Merge-Tree (LSM-Tree) was introduced more than two decades ago as an indexing data structure that increases the effective use of bandwidth by reducing seeks in hard disk drives (HDDs) over the alternative, the B-Tree [9]. Today, it is the predominant method for implementing persistent key-value stores, such as LevelDB [17] and RocksDB [13], and is at the core of many scale-out storage systems [7,23,46] and databases [14,32]. While the basic design of LSM-Trees has stayed the same, the storage medium for which it was conceived is changing.

Faced with ever-growing capacity demands [39], many large-scale storage systems continue to rely predominantly on HDDs due to their low $/GB price [30,35]. Seeking greater density, drive manufacturers are now using Shingled Magnetic Recording (SMR) to achieve device capacity increases of more than 25% [43]. Unlike traditional "Conventional Magnetic Recording" (CMR), in which sectors (blocks) can be updated in any order, SMR requires that the drive be managed in units of sequentially-written *zones*, akin to SSD erase blocks. This constraint has led to the development of two types of SMR drives. Similar to SSDs, drive-managed SMR (DM-SMR) drives use an internal translation layer to expose a traditional block interface, but their performance suffers due to internal cleaning [2]. Host-managed SMR (HM-SMR) drives expose a backward-*in*compatible zone interface [19] that only supports sequential writes within a zone and zone erases.

Administrators seeking to deploy LSM-tree-based key-value stores or other applications on SMR-based storage face significant performance difficulties. While cloud providers [24,30] and storage server vendors [8,29] are adapting their private software stacks to support HM-SMR drives, existing support in available OSs is limited to two main options: (1) use a traditional block file system on a DM-SMR drive, which often results in much reduced throughput, or (2) use a log-structured file system (e.g., F2FS if using Linux) on HM-SMR drive, which we observe results in significant performance and storage capacity overheads due to redundant cleaning. This paper describes BlueFS, a user-space file system optimized for LevelDB-based LSM-trees running on HM-SMR drives, and shows how it provides unmodified LSM-trees with the full capacity and performance of SMR drives.

Difficulties with existing options. DM-SMR drives are known to handle purely sequential writes with minimal overhead [3], and LSM-Trees are often described as eliminating random I/O [28,34,44], suggesting that the two would be a good fit. In practice, however, the OS storage stack will split large I/O requests issued by LSM-Tree.

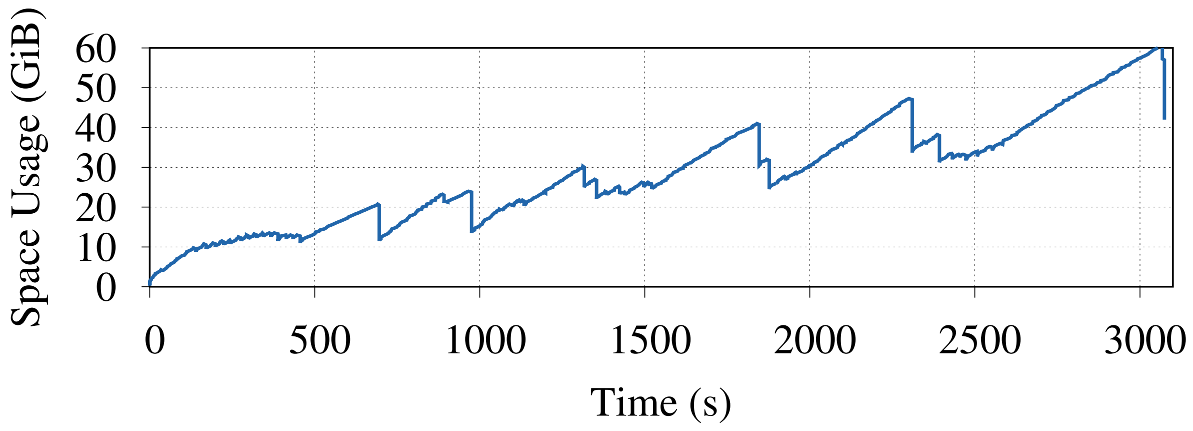

Figure 2: RocksDB space usage on HM-SMR ebbs and flows significantly due to compaction, triggering cleaning in F2FS that rearranges GiBs of data that will soon be deleted.

Combined with key lookup requests and metadata churn of the underlying system these will generate enough random I/O to degrade the performance of a DM-SMR drive. As shown in Figure 1, these factors can result in DM-SMR being 20-40% slower than CMR drives when inserting 150 million key-value pairs in three popular LSM-Tree implementations. Note that the backends' configuration had already been optimized for hard drives *prior* to the comparison, achieving up to 4× speedup compared to out-of-the-box performance. Our benchmark used 20-byte keys, 400-byte values, resulting in a 31 GiB database size, and the system RAM was limited to 6 GiB.

On the other hand, running these key-value stores on an HM-SMR drive using F2FS, can interfere with LSM-Tree operations and reduce the effective capacity of the device. LSM-Trees periodically *compact* data by merging files with stale data into new files and deleting the old files. This results in ebbs and flows in space usage, as shown in Figure 2. These temporary increases in space usage induced by LSM-Tree compactions trigger F2FS's segment cleaning process, causing it to rearrange GiBs of files that are deleted right after the compaction has completed. This inefficiency caused by *cleaning of compaction*, not unlike the *journaling of journal* anomaly [20], is the result of a violation of the end-to-end principle in layered system design [42] arguing for functionality to be implemented once, and in a layer that has the semantic knowledge to efficiently perform it. Furthermore, F2FS (as any instance of a log-structured file system [41]) needs to reserve extra space for efficient cleaning, which reduces the capacity benefit of SMR. This capacity overhead was 15-20% in our experiments.

BlueFS: bridging LSM-Trees and HM-SMR drives. We believe that the I/O characteristics of LSM-Tree-based key-value stores are an excellent fit for the zone interface offered by HM-SMR drives. Unfortunately, trying to fit current LSM-Trees with their default settings to an HM-SMR drive leads to reduced throughput and zone fragmentation that requires expensive cleaning [1, 31, 37]. However, as we demonstrate in this paper, simple configuration changes combined with a file system designed for LSM-Trees can enable them to adapt seamlessly to HM-SMR drives, utilizing their extra capacity and bandwidth with minimal overhead and with no changes to the LSM-Tree source code.

BlueFS is a user-space, extent-based, journaling file system optimized for LSM-Trees running on HM-SMR drives. Experiments with BlueFS show that it delivers on the goals. For example, unmodified RocksDB atop BlueFS on HM-SMR drive performs random inserts 64% faster than atop XFS on a DM-SMR drive, and 30% faster than atop XFS on a CMR drive. By specializing data placements and I/O sizes, BlueFS even allows unmodified LevelDB to run 2–20× faster than GearDB [1], a recent key-value store designed for SMR drives. BlueFS also allows Ceph, a widely-used open-source distributed storage system, to utilize HM-SMR drives as a metadata store, taking it one step closer to being an open-source storage solution that can leverage SMR benefits.

This paper makes three primary contributions:

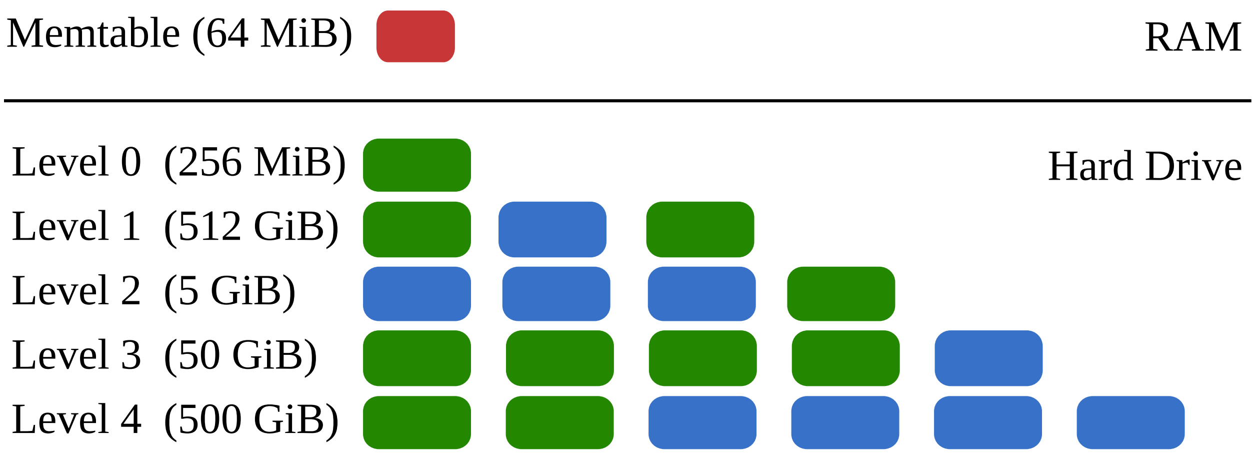

Figure 3: Data organization in RocksDB. Green squares represent Sorted String Tables (SSTs). Blue squares represent SSTs selected for two different concurrent compactions.

- A foray into the challenges of deploying LSM-Trees on *real* HM-SMR drives, and practical solutions for them (§ 3).
- The design and implementation of BlueFS, detailing how it matches the access patterns of LSM-Trees to the characteristics of HM-SMR drives (§ 4).
- An experimental evaluation of BlueFS with a breakdown of the contributions of each optimization, and a demonstration showing that LevelDB running on BlueFS outperforms other key-value implementations on HM-SMR drives (§ 5).

2 Background and Motivation

Since describing multiple LevelDB-based LSM-Trees that can run on BlueFS would be unwieldy, in the rest of the paper we use RocksDB-centric descriptions, given that it is the most optimized and actively developed key-value store. This section describes the high-level design of RocksDB, and provides a quick overview of SMR drives and the new zone interface. It then describes the cleaning problem of current LSM-Tree implementations that motivates our work.

2.1 RocksDB

Every key-value inserted to RocksDB is first individually written to a Write-Ahead Log (WAL) file using the `pwrite` system call, and then buffered in an in-memory data structure called *memtable*. By default, RocksDB performs **asynchronous inserts**: `pwrite` returns as soon as the data is buffered in the OS page cache and the actual transfer of data from the page cache to storage is done later by the kernel writeback threads. Hence, a machine crash may result in loss of data for an insert that was acknowledged. For applications that require the durability and consistency of transactional writes RocksDB also supports **synchronous inserts** that do not return until data is persisted on storage.

When the memtable reaches a preconfigured size, a new one is created. In batches, memtables are merge-sorted and written to storage as a Sorted String Table (SST) file. SSTs in RocksDB are organized into multiple levels, as shown in Figure 3. The aggregate size of each level L_i is a multiple of L_{i-1}, starting with a fixed size at L_1. At first data is written at level L_0. Once the number of L_0 SSTs reach a threshold, the compaction process selects all of L_0 SSTs, reads them into memory, merge-sorts them, and writes them out as new L_1 SSTs. For higher levels, compactions are triggered when the aggregate size of the level exceeds a threshold, in which case one SST from the lower level and multiple SSTs from a higher level are compacted, as shown in Figure 3. If memtable flushes or compactions cannot keep up with the rate of inserts, RocksDB stalls inserts to avoid filling storage and to prevent lookups from slowing down.

2.2 Shingled Magnetic Recording

SMR increases drive capacities by partially overlapping adjacent magnetic tracks, leaving narrower tracks for the drive read heads to still be able to access the data, similar to roof shingles. While this technique does not affect purely sequential write workloads, random (over)writes are challenging as they would corrupt the data of adjacent tracks. To mitigate this, SMR drives are partitioned in zones. Within each zone (e.g., 256 MiB), tracks are shingled and acceptable operations are limited to sequential writes or zone erases.

DM-SMR drives use a Shingled Translation Layer (STL) to present a block interface to the host instead of zones. Random writes are buffered in a *persistent cache* of reserved tracks, and are later written to their final locations by overwriting existing zones [2] during *cleaning*. Large sequential writes are directly written to their final locations [3].

HM-SMR drives expose zones through a novel interface [19]. The first few hundred zones are conventional tracks that can be written randomly, while the rest are shingled, i.e., strictly sequential. For each sequential zone the drive keeps a *write pointer* that starts at the beginning of the zone and is updated after each append. A write to a location other than the write pointer will fail, but the write pointer can be reset to the beginning of a zone.

2.3 Cleaning Host-Managed SMR Drives

Running RocksDB or other key-value stores on an HM-SMR drive leads to the *segment cleaning* problem of LFS [41], because the preferred SST sizes are much smaller than the zones of the drive. After multiple compactions zones will contain fragmented free space from SSTs that have been merged to a new SST. Reclaiming the space occupied by dead SSTs requires migrating live SSTs to another zone. Recent work proposes a new data format and compaction algorithm to avoid HM-SMR drive cleaning for an LSM-Tree with 4 MiB SSTs [1].

Cleaning can be eliminated, however, if we align the SST and zone sizes. This way, at the end of a compaction, "dead" SST-zones can be reclaimed by merely resetting the zone's write pointer. There are other compelling reasons for increasing SST size, such as enabling disks to do streaming reads with fewer seeks, reducing expensive sync operations, and reducing the number of open file handles. Applying this simple idea to production-grade key-value stores and real HM-SMR drives, however, involves several challenges. We describe those in the following section.

3 LSM-Tree Challenges on HM-SMR

Popular LSM-Tree implementations use a file system, and BlueFS is designed to make them work well on HM-SMR drives. Specifically, BlueFS manages the raw capacity directly, placing LSM-Tree file data and issuing I/O requests to efficiently use the zone interface of the HM-SMR drive. Implementing BlueFS involves resolving several challenges that we describe in this section.

3.1 Reordered Writes

Like all LevelDB-derived LSM-Tree implementations [1, 12, 21, 27, 37, 38], RocksDB uses buffered I/O when writing compacted SSTs. This improves performance significantly because compacted SSTs can be kept in the OS cache; this both speeds up future lookups, and reserves disk bandwidth for memtable flushes, which in turn increases transaction throughput.

Using buffered I/O, however, does not guarantee write ordering that is essential for HM-SMR drives. Page writeback can happen from different contexts at the same time, and the pages picked up by each context will not be necessarily zone-aligned. Furthermore, there are no write-alignment constraints with buffered writes, so an application may write parts of a page across different operations. In this case, however, the same last page cannot be overwritten to add the remaining data when the sequential write stream resumes. This forces the use of direct I/O with HM-SMR drives. *To mitigate performance issues, BlueFS implements its own cache so that reads are not always served from disk during compaction.*

Note that reordering may occur even with direct I/O if the I/O scheduler reorders writes. The Completely Fair Queueing scheduler is one such scheduler. Thus, the recommended schedulers for use with HM-SMR drives are

Deadline and Multi-Queue Deadline, which guarantee delivery of writes without reordering.

3.2 Synchronous Writes to the Log

The `libzbc` [47] library is the de-facto way of interacting with HM-SMR drives, and the library used by LevelDB-derived key-value stores designed for HM-SMR [1, 31]. As part of `libzbc`, the `zbc_pwrite` call is provided for positional writes to the device, which has similar semantics to the `pwrite` system call. Even though `pwrite` is a synchronous call, since it is used with buffered I/O in LevelDB-based key-value stores, it is effectively made asynchronous (§ 2.1). On the other hand, the `zbc_pwrite` call waits for the drive to acknowledge the write, since the HM-SMR drive can only be used with direct I/O. This works well when data is buffered in memory and written in large chunks, which is the case for memtable flushes and SSTs writes during compaction. Writes to the Write-Ahead Log (WAL), however, happen after every key-value insertion. As a result, *every insertion must be acknowledged by the disk, limiting the throughput of the key-value store to that of small synchronous writes to drive.*

To remedy this bottleneck BlueFS uses the `libaio` *in-kernel asynchronous I/O framework*. This approach works as long as the asynchronous I/O operations are issued in order and the right I/O scheduler is used. BlueFS still uses `libzbc` for resetting zones and flushing the drive.

3.3 Misaligned Writes to the Log

Random and misaligned writes violate the zone interface, and therefore cannot be used with HM-SMR drives (§ 3.1). In RocksDB, there are three sources of such writes. First, RocksDB produces a handful of small files that receive negligible amounts of non-sequential I/O. Placing those in a conventional zone solves the problem. Second, the last block of SST files suffers from overwrites, which we found were due to a bug in RocksDB, which we reported and has since been fixed by the RocksDB team [11]. Third, and more surprisingly, *the last block of the WAL tends to be overwritten if the previous append operation was not aligned, making it unsuitable for placing on a sequential zone.* Deployments of RocksDB typically shard key space and run multiple instances where WALs are close to the data, thereby avoiding long seeks. Placing the WAL in a conventional, however would result in expensive seeks—especially for synchronous inserts—because conventional zones are typically concentrated in one part of the drive. Furthermore, this would waste conventional track space on an append-only file.

BlueFS introduces a special format for the WAL that allows it to be written sequentially so that it can be placed on a sequential zone.

4 Design and Implementation of BlueFS

BlueFS is a user-space, extent-based, journaling file system specialized for LevelDB-based LSM-Tree implementations, such as RocksDB, running on a raw HM-SMR drive. BlueFS maintains an inode for each file, with a serial number, modification time, the list of extents allocated to the file, the actual and the allocated size of the file. The superblock resides at a fixed offset in the first conventional zone.

Journaling. The superblock contains an inode for the journal, and the journal has the only copy of all file system metadata. At mount, the journal is replayed and all metadata is loaded in memory. On every metadata operation, such as directory and file creation or extent allocation, the journal and in-memory metadata are updated. The journal does not have a fixed location; its extents may be interleaved with other file extents as Figure 4 shows. When the journal reaches a preconfigured size, it is compacted and written to a new location, and the superblock is updated to point to it. These design decisions work because large files and periodic SST compactions limit the volume of metadata at any point in time.

The WAL file is always growing and its metadata needs to be updated after every insert. This results in two synchronous writes for every insert: one to the WAL file and another to the underlying file system's journal. We modified RocksDB to pre-allocate a pool of WAL files and reuse them. With this change, the size of the WAL files does not change, and a synchronous insert results in only one synchronous write to the WAL file. This optimization increases

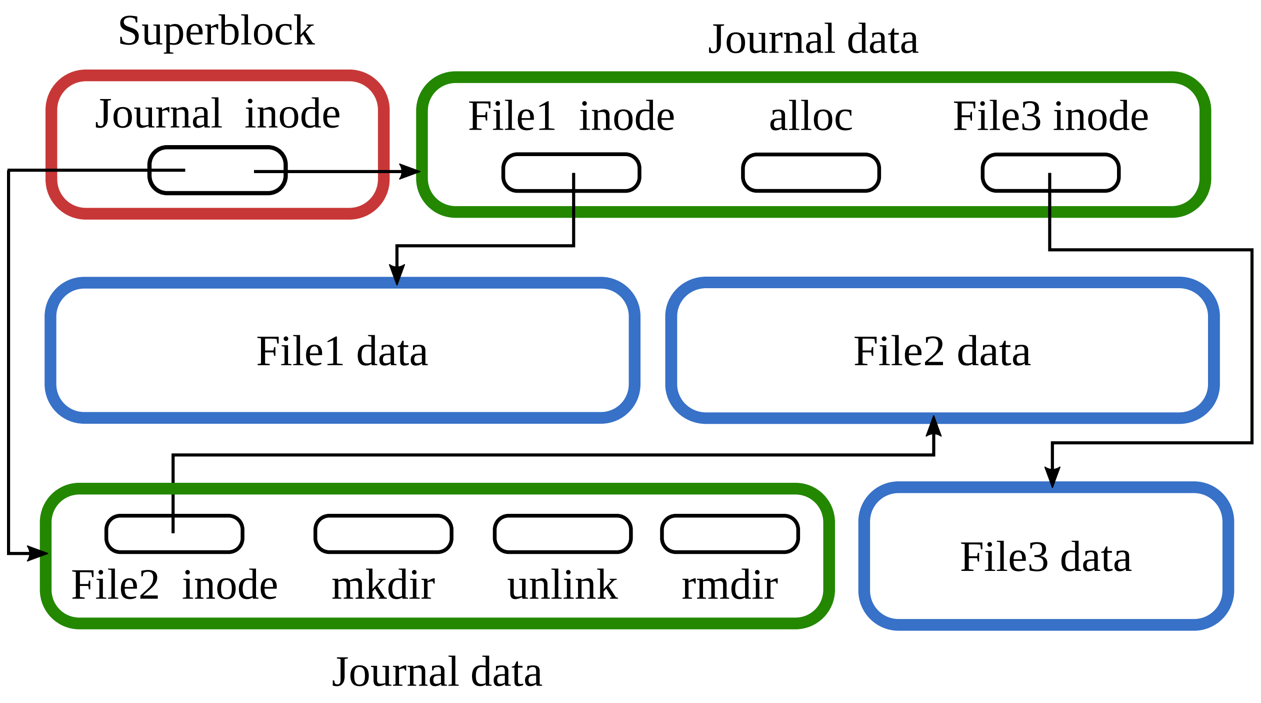

Figure 4: BlueFS data layout.

the throughput of synchronous inserts three-fold, and has been incorporated in the mainline RocksDB code[1].

Journal placement. To leverage the full bandwidth of the HM-SMR drive and to avoid synchronous writes to WAL (§ 3.2), BlueFS uses the `libaio` asynchronous I/O framework to issue requests, and uses a dedicated thread to poll for completed requests. In order to place the WAL in a sequential zone (§ 3.3), BlueFS wraps every write to the WAL in a record that keeps the length of the write inline and always pads out the record to a 4 KiB boundary. With this change, the read code for the WAL is no longer a direct mapping from an extent start to offset, and record lengths have to be read to determine the actual content. This, however, is not a problem because the WAL is only read sequentially and only during crash recovery.

Data placement. The extent allocator in BlueFS follows different policies for randomly updated and sequential files. Randomly updated files are given 256 KiB extents in a conventional zone. Sequential files, such as SSTs, WALs, and the BlueFS journal, are assigned complete sequential zones. Identifying these files is trivial, so BlueFS is configured to do that for all popular LSM-Tree implementations. Allocating complete zones to SSTs eliminates the cleaning problem (§ 2.3) because deleting an SST file during compaction consists of resetting the write pointer of the zone and returning it to the list of free zones.

Caching. To stop RocksDB from hitting the storage during compaction for every SST read, BlueFS implements a file cache with FIFO policy. Every SST file that is ever written is also placed to a cache of preconfigured size.

5 Evaluation

We run all experiments on a system with AMD Opteron 6272 2.1 GHz CPU with 16 cores and 128 GiB of RAM, running Linux kernel 4.18 on the Ubuntu 18.04 distribution. For Ceph experiments we use three such nodes connected with 40 GbE using Mellanox ConnectX-3 cards.

We incrementally show how each of our optimizations improves performance using a benchmark that inserts 150 million pairs of 20B keys, 400B values using the `db_bench` tool that comes with RocksDB. During the benchmark

[1]This is a performance optimization, but its adoption by RocksDB allows us to use it and still claim that BlueFS requires no source code changes from the LSM-Tree implementation.

Type	Vendor	Model	Cap.	Max. BW	Avg. BW
CMR	Hitachi	HUA72303	3 TB	160 MiB/s	120 MiB/s
DM-SMR	Seagate	ST8000AS0022	8 TB	197 MiB/s	140 MiB/s
HM-SMR	HGST	HSH721414AL	14 TB	237 MiB/s	170 MiB/s

Table 1: HDDs used in evaluation. We determine the average bandwidth by writing the whole drive sequentially in 1 MiB chunks and then averaging the bandwidth reported every second.

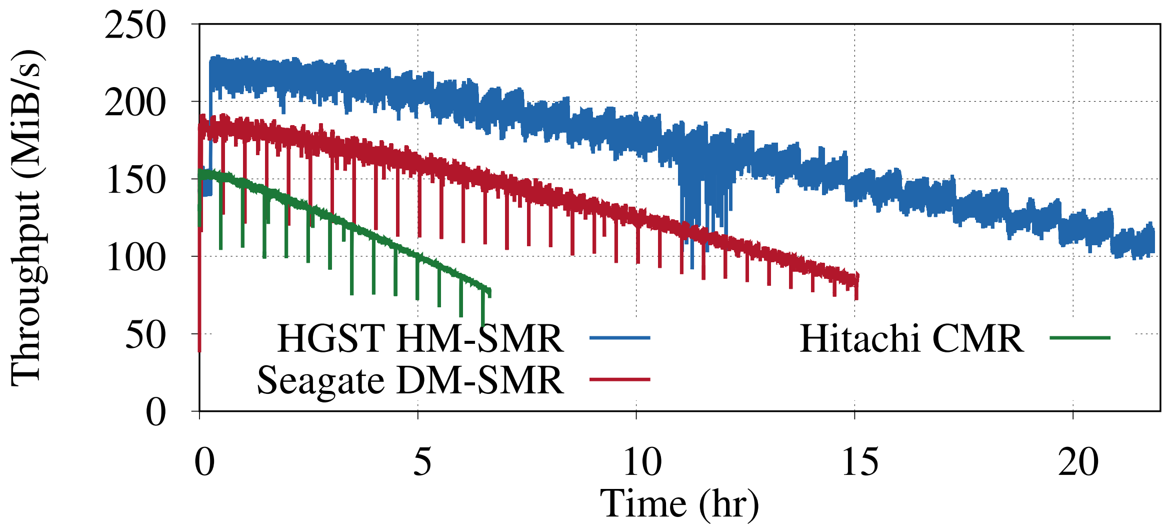

Figure 5: The sequential throughput of the drives used in our evaluation. fio was used to write to every byte of each drive, and the HM-SMR drive shows 32% higher throughput than the CMR drive.

run, RocksDB writes 200 GiB of data through memtable flushing, compaction, and WAL inserts, and reads 100 GiB of data due to compaction. The size of the database is 59 GiB uncompressed and 31 GiB compressed. To emulate a realistic environment where the amount of data in the OS page cache is a small fraction of the data stored on a high-capacity drive, we limit the operating system memory to 6 GiB, which leaves slightly more than 2 GiB of RAM for the page cache after the memory used by the OS and benchmark application, resulting in 1:15 ratio of cached to on-disk data. We repeat every experiment at least three times and report the average and the standard deviation.

We establish two baselines. The first is RocksDB running on an XFS-formatted CMR drive. This is the baseline we want to achieve with RocksDB on an HM-SMR drive, a similar mechanical device with a more restricted interface but higher capacity. The second baseline is RocksDB running on an XFS-formatted DM-SMR drive. This is a baseline we want to beat given that it is the only viable option for running RocksDB on a high-capacity SMR drive, but has suboptimal performance due to internal cleaning.

5.1 Establishing the CMR Baseline

Table 1 shows the hard disk drives used in our evaluation. The sequential throughput of those drives differs. In Figure 5 we use fio [5] to sequentially fill the drives and find that the HGST HM-SMR drive has 32% higher throughput than the Hitachi CMR drive. RocksDB writes are not purely sequential, though. In addition to small-sized XFS metadata and journal writes, there are memtable flushes, WAL inserts, compaction reads and writes that are happening concurrently at different offsets of the drive resulting in seeks and expensive cache `FLUSH` operations to the drives. For our CMR baseline, we run RocksDB on Hitachi CMR drive with XFS, and compare it against RocksDB running on HGST HM-SMR drive with BlueFS. This graph is aimed to demonstrate that BlueFS on

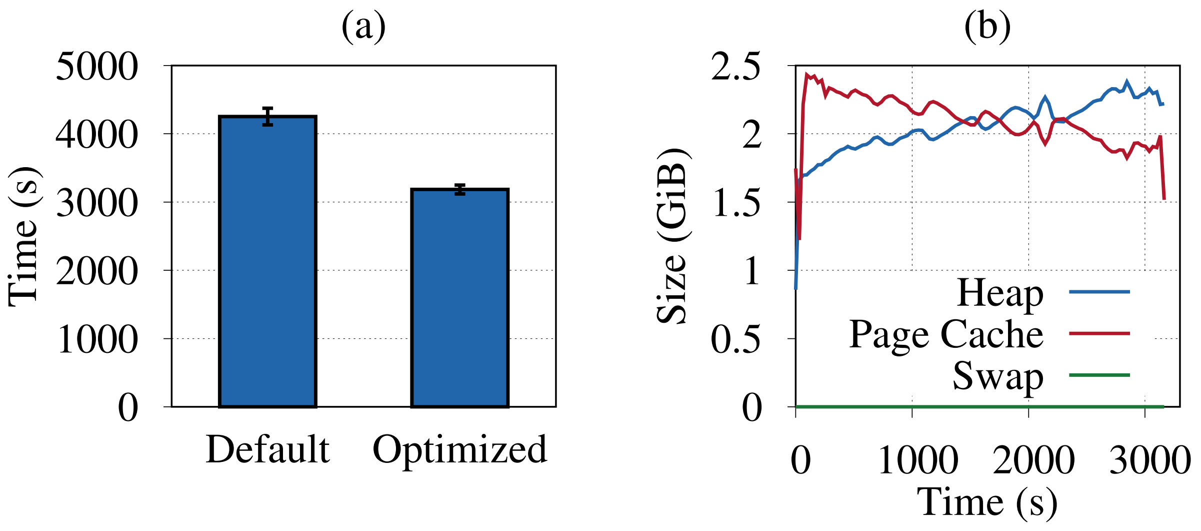

Figure 6: (a) Benchmark runtimes of RocksDB with default and optimized settings on a CMR drive with XFS. (b) Memory usage by RocksDB and `db_bench` heap allocations, OS page cache, and swap usage during the optimized RocksDB run.

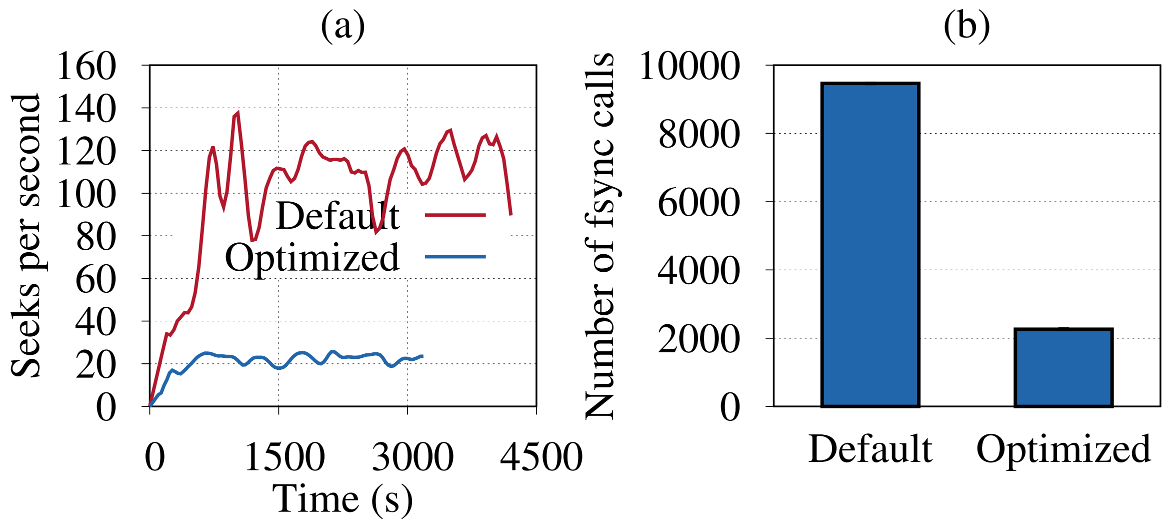

Figure 7: (a) Number of seeks per second due to reads during the benchmark run with default and optimized RocksDB settings on CMR with XFS, extracted from `blktrace` data using `seekwatcher`. (b) Number of `fsync/fdatasync` system calls issued during the benchmark run.

the HM-SMR has a bandwidth advantage over XFS on the CMR drive, which should be taken into account when interpreting our benchmark results.

For the baseline we run RocksDB on XFS (the file system recommended by RocksDB team [15]) and tune RocksDB settings for optimal performance on a hard drive [16]. Figure 6 (a) shows that with optimized settings, the benchmark completes 34% faster. Figure 6 (b) shows the heap memory allocated by RocksDB and `db_bench`, the OS page cache, and the swap usage. The two key observations are that the page cache almost always uses more than 2 GiB of RAM, and no swapping occurs. Next, we look at the settings we change to get this effect.

We focus on two tunables that impact performance most: `compaction_readahead_size` and `write_buffer_size`. We increase the former from the default of 0 to 2 MiB. This setting determines the size with which `pread` system call issues read requests for reading SSTs during compaction. With the default setting, RocksDB issues 4 KiB

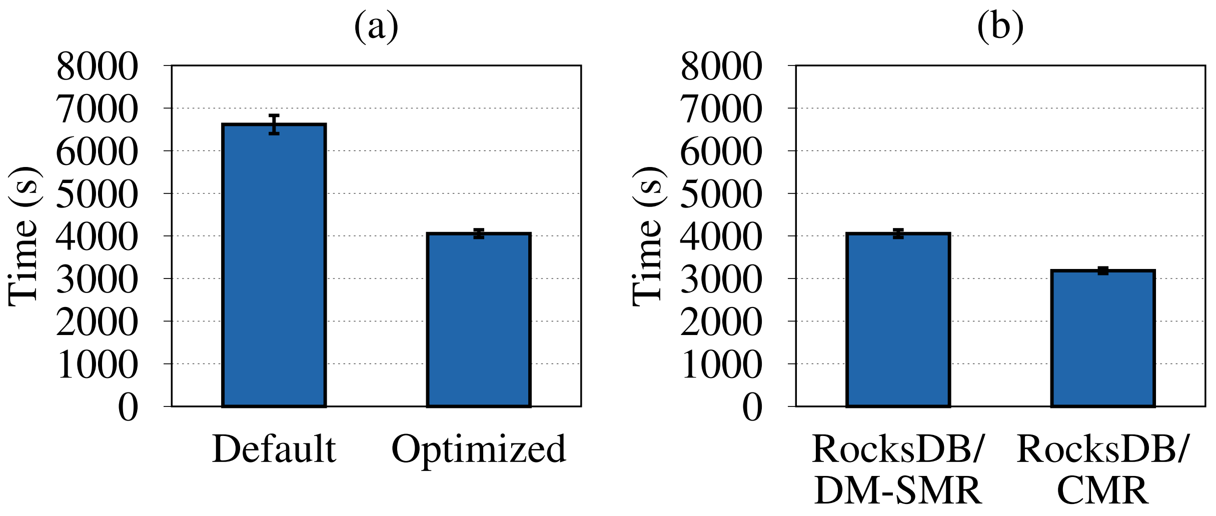

Figure 8: (a) Benchmark runtimes of RocksDB with default and optimized settings on a DM-SMR drive with XFS. (b) The two baselines, RocksDB/DM-SMR and RocksDB/CMR, that run with optimized RocksDB settings on DM-SMR and CMR drives, respectively, with XFS.

requests and prefetches 256 KiB at a time, however, given that the compaction threads read large SSTs sequentially into memory, 256 KiB request size is suboptimal and incurs unnecessary seeks due to interruptions from other ongoing disk operations. Figure 7 (a) shows that by increasing the request size to 2 MiB, the number of seeks drops from the average of 110 seeks per second to almost 20 seeks per second.

We also increase the `write_buffer_size` from the default of 64 to 256 MiB. This setting determines the memtable size that will be buffered in memory before being flushed to disk. As expected, increasing the memtable size by 4×, reduces the number of expensive `fsync/fdatasync` system calls issued by a similar amount, as Figure 7 (b) shows.

These two configuration changes are the reason behind the 34% speedup shown in Figure 6 (a). We also experimented with the maximum number of concurrent background jobs and settle on 4 threads; other values result in worse performance. Finally, we increased SST sizes from 64 to 256 MiB, but this change did not have a noticeable effect on performance; it did, however, allow us to exactly map an SST to an HM-SMR zone. We choose RocksDB with optimized settings running on XFS as our CMR baseline and refer to it as RocksDB/CMR from here on.

5.2 Establishing the DM-SMR Baseline

The optimized settings that we used for the CMR drive are beneficial to a DM-SMR drive for the same reasons (§ 5.1), the analysis of which we omit for brevity. Figure 8 (a) compares the runtime of the benchmark on a DM-SMR drive with default and optimized RocksDB settings. We see that optimizations have even bigger impact on the DM-SMR drive where the optimized RocksDB completes the benchmark 63% faster.

We choose RocksDB with optimized settings as our DM-SMR baseline and refer to it as RocksDB/DM-SMR from here on. Figure 8 (b) shows both of our baselines—RocksDB/CMR and RocksDB/DM-SMR, with the former being about 27% faster. This difference increases with larger workloads. When we insert one billion key-value pairs, where the compressed database size is 200 GiB and the total I/O size is 5 TB, RocksDB/CMR completes 50% faster than RocksDB/DM-SMR.

Both RocksDB/CMR and RocksDB/DM-SMR run on top of the XFS file system. From here on, we refer to RocksDB running on HM-SMR, which runs on top of BlueFS, as RocksDB/HM-SMR. Our aim is to get RocksDB/HM-SMR to beat RocksDB/DM-SMR and to perform at least as well as RocksDB/CMR.

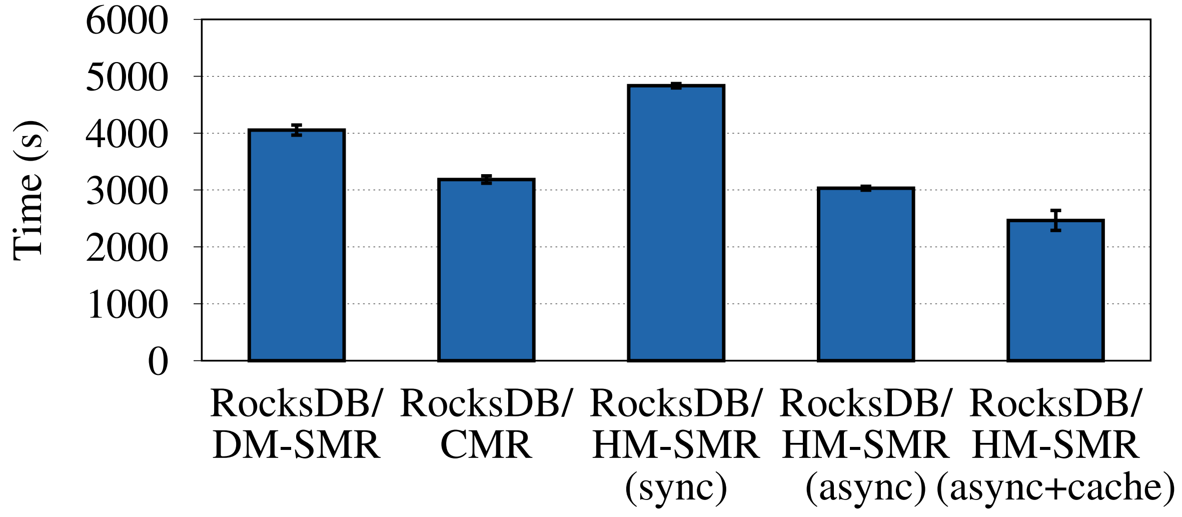

Figure 9: Benchmark runtimes of the baselines and RocksDB/HM-SMR iterations. The benchmark performs 150 million **asynchronous** inserts of key-value pairs of size 20 bytes and 400 bytes, respectively.

5.3 Getting RocksDB to Run on HM-SMR

Our first iteration of BlueFS uses synchronous I/O calls in `libzbc`, which cause the writes to the WAL to become a bottleneck (§ 3.2). Figure 9 shows that RocksDB on this iteration of BlueFS, which we denoted by RocksDB/HM-SMR/sync, completes the benchmark in 4,800 seconds, slower than both RocksDB/CMR and RocksDB/DM-SMR.

Figure 10 gives a detailed look at the first 50 seconds of the run. We see that in the first 7 seconds, during which only writes to the WAL happen, the write throughput is fixed at 40 MiB/s. In the next 3 seconds a memtable is flushed, which increases the write throughput to 80 MiB/s and reduces the insert throughput to 60 Kops/s. This is expected because insert throughput is determined by the speed of writes to WAL, and during memtable flush, we have two threads sharing the badwidth: one is flushing memtable data and the other writing to WAL. Once the memtable flush completes at the 10th second, the insert throughput jumps back and the write throughput is again at 40 MiB/s. Another memtable flush happens and the pattern repeats, and right at the end of the second memtable flush a compaction starts at the 17th second. This increases read and write throughput because the compaction is done by a single thread that reads old SSTs and writes new ones, and reduces insert throughput to the same level as during the memtable flush because we still have two threads sharing the bandwidth. Before the compaction completes, another memtable flush starts at the 23rd second. Now we have three threads sharing the disk bandwidth. The write bandwidth is the highest because all of them are writing, read bandwidth is slightly lower, and most importantly, the insert throughput is at the lowest.

During the whole run, RocksDB never stalls inserts (§ 2.1) because the slow writes to the WAL produce small enough work that memtable flushes and compactions can keep up with.

5.4 Running Fast with Asynchronous I/O

In our second iteration we use `libaio` to perform asynchronous I/O and keep `libzbc` for zone operations. Figure 9 shows that RocksDB on the second iteration of BlueFS, which we denote by RocksDB/HM-SMR/async, completes in 3,000 seconds, 60% faster than the synchronous version. At this point, RocksDB on BlueFS is already 30% faster than RocksDB/DM-SMR and as fast as RocksDB/CMR, despite using direct I/O and reading all the data from the drive during compaction. This is not surprising given that the HM-SMR drive has higher bandwidth than the CMR drive (Table 1).

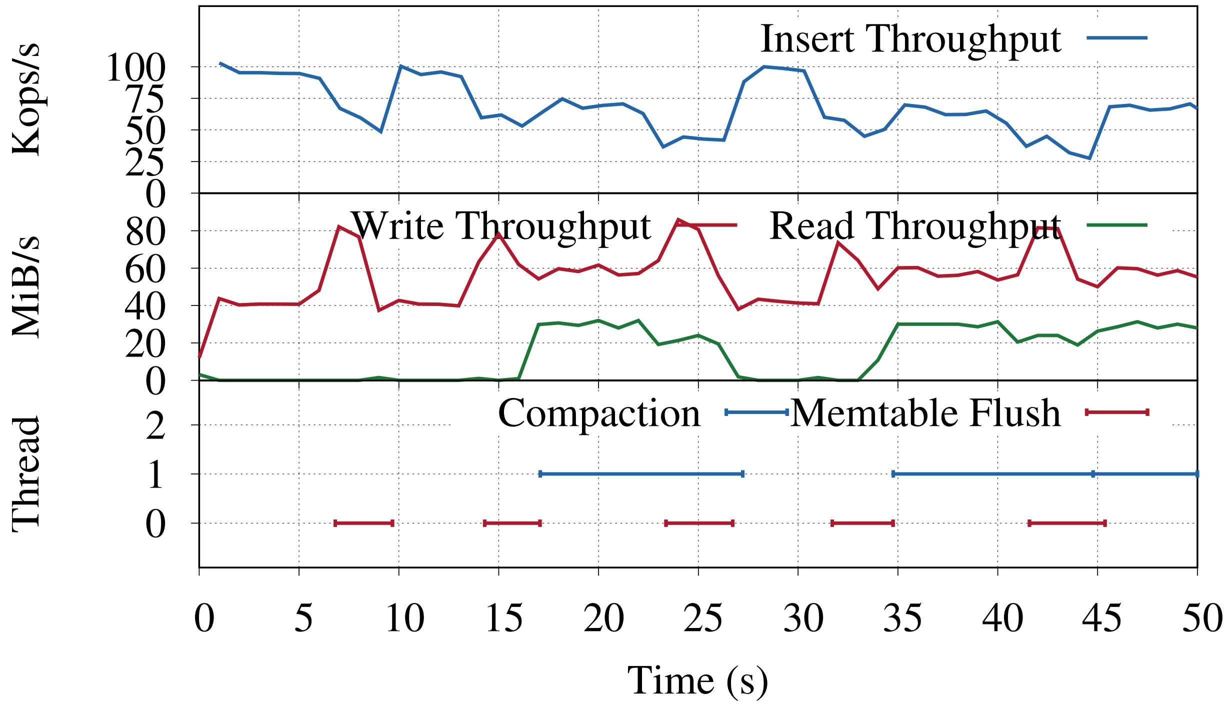

Figure 10: Insertion throughput, read/write bandwidth, and compaction/memtable flush operations going on during the first 50 seconds of RocksDB/HM-SMR/**sync** running the benchmark.

Figure 11 gives a detailed look at the first 50 seconds of the run. We see that writes to the WAL during the first 3 seconds are almost twice as fast, and memtable flush starts 3 seconds earlier, compared to Figure 10. Fast inserts result in continuous memtables flushes, and similarly, compaction starts 7 seconds earlier and does not stop. Unable to keep up with the work, RocksDB stalls inserts almost every 5 seconds, which results in a spiky throughput graph. Despite the stalls, the average throughput is higher and RocksDB completes the benchmark 60% faster on BlueFS using `libaio`.

Figure 12 shows the similar graph for the whole run of RocksDB/HM-SMR/async. The most outstanding pattern in Figure 12 (a) are the long periods of very low throughput, such as the one between 450-660s. Figure 12 (c) shows that during this period only compaction and no memtable flushing is happening, suggesting that RocksDB has stalled inserts until compaction backlog is cleared. During this period, effectively only the compaction thread is running and as the period 450-660s shows in Figure 12 (b), it divides the disk bandwidth between reading SSTs from the drive and writing them out. To complete the reads, and thereby compactions faster, in our next iteration we add SST cache to BlueFS.

5.5 Running Faster with a Cache

In our third iteration we add a simple write-through FIFO cache for SST files. Figure 9 shows that RocksDB on the final iteration of BlueFS, which we denote by RocksDB/HM-SMR/async+cache, completes in 2,500 seconds, 30% faster than the version with no cache. In this run, we set the SST cache size to 2 GiB—the same amount of RAM that the OS page cache used with RocksDB/CMR. Figure 13 shows the detailed graph for the whole run. Comparing it to Figure 12 we see that, for example, up to 450s, the read throughput is lower, and between 450-660s the read throughput stays below 30 MiB/s compared to rising 40 MiB/s, suggesting that some of the reads are being served from the cache. Similarly, the write throughput stays above 60 MiB/s compared to staying at 40 MiB/s, and

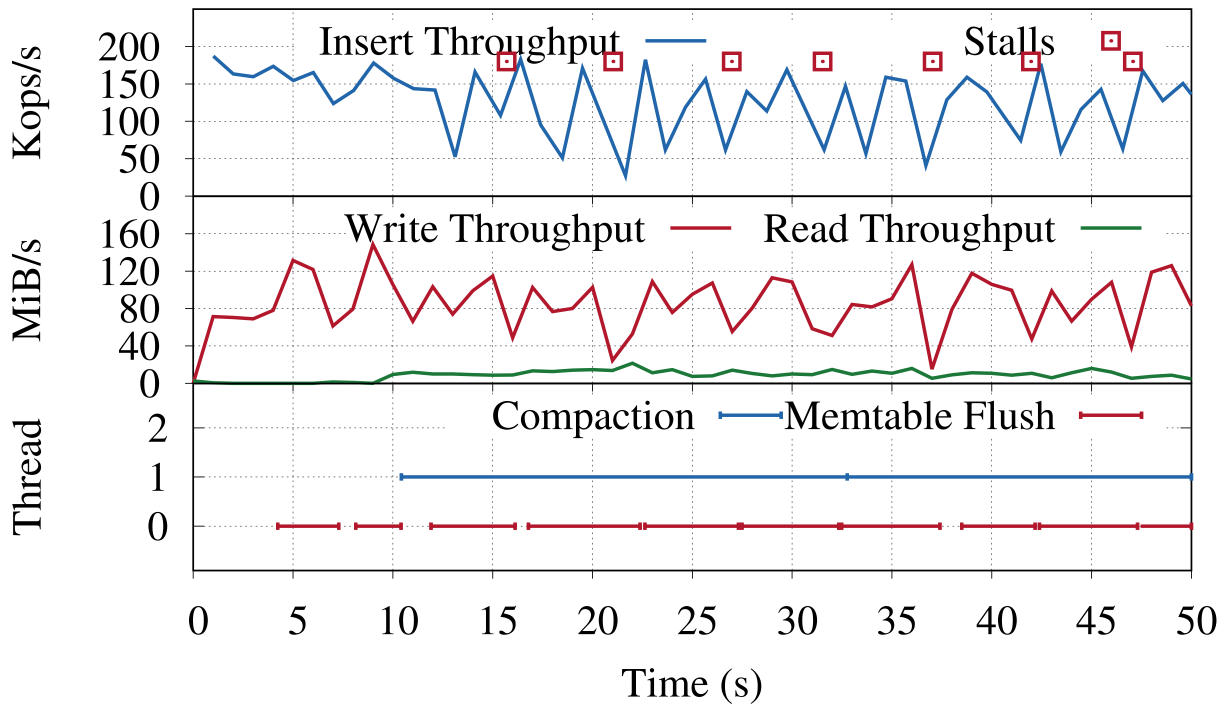

Figure 11: Insertion throughput, read/write bandwidth, and compaction/memtable flush operations going on during the first 50 seconds of RocksDB/HM-SMR/**async** running the benchmark.

the insertion throughput is not flat at the bottom, indicating that writes to the WAL are still happening and the inserts are not stalled as badly as in the no-cache case. Overall, Figure 13 has shorter periods of low insertion throughput compared to Figure 12 and therefore, higher average insertion throughput.

A disadvantage of our cache implementation is that its unit of eviction is a file. Instead, the OS page cache will evict only part of a file when it runs out of space and still serve the remaining blocks. By evicting the whole file, our implementation leaves 12.5% of the 2 GiB cache empty, given the 256 MiB SSTs. We leave the full treatment of a cache optimized for LSM-Tree workloads as future work.

5.6 Space Efficiency

Figure 12 (d) shows that RocksDB/HM-SMR makes optimal use of disk space. Matching the figure with Figure 12 (c) shows that allocated zones grow during long compaction processes and they are released at the end. This is because RocksDB first merges multiple SSTs into one large file and deletes it after creating new files as a result of the merge. We see that the number of zones drop to 168, which corresponds to about 42 GiB—larger than the 31 GiB input data. This is the result of the benchmarking application, `db_bench`, shutting down as soon as it finishes the benchmark, without waiting for all compactions to complete, leaving some large temporary SSTs around. We modified `db_bench` to wait until all compactions have completed, and observed that the number of bands dropped to 135, which is about 33 GiB, only slightly larger than the database size because some of the space is occupied by the WAL files, and some of the SSTs produced as a result of compaction may end up being smaller than 256 MiB.

5.7 Faster Synchronous Inserts

In addition to the default asynchronous inserts, RocksDB also allows users to perform synchronous inserts (§ 2.1). Unsurprisingly, synchronous inserts are slower by orders of magnitude, but still, they provide consistency guarantees that cause applications to often mix them with asynchronous inserts. So far, we have been optimizing BlueFS for

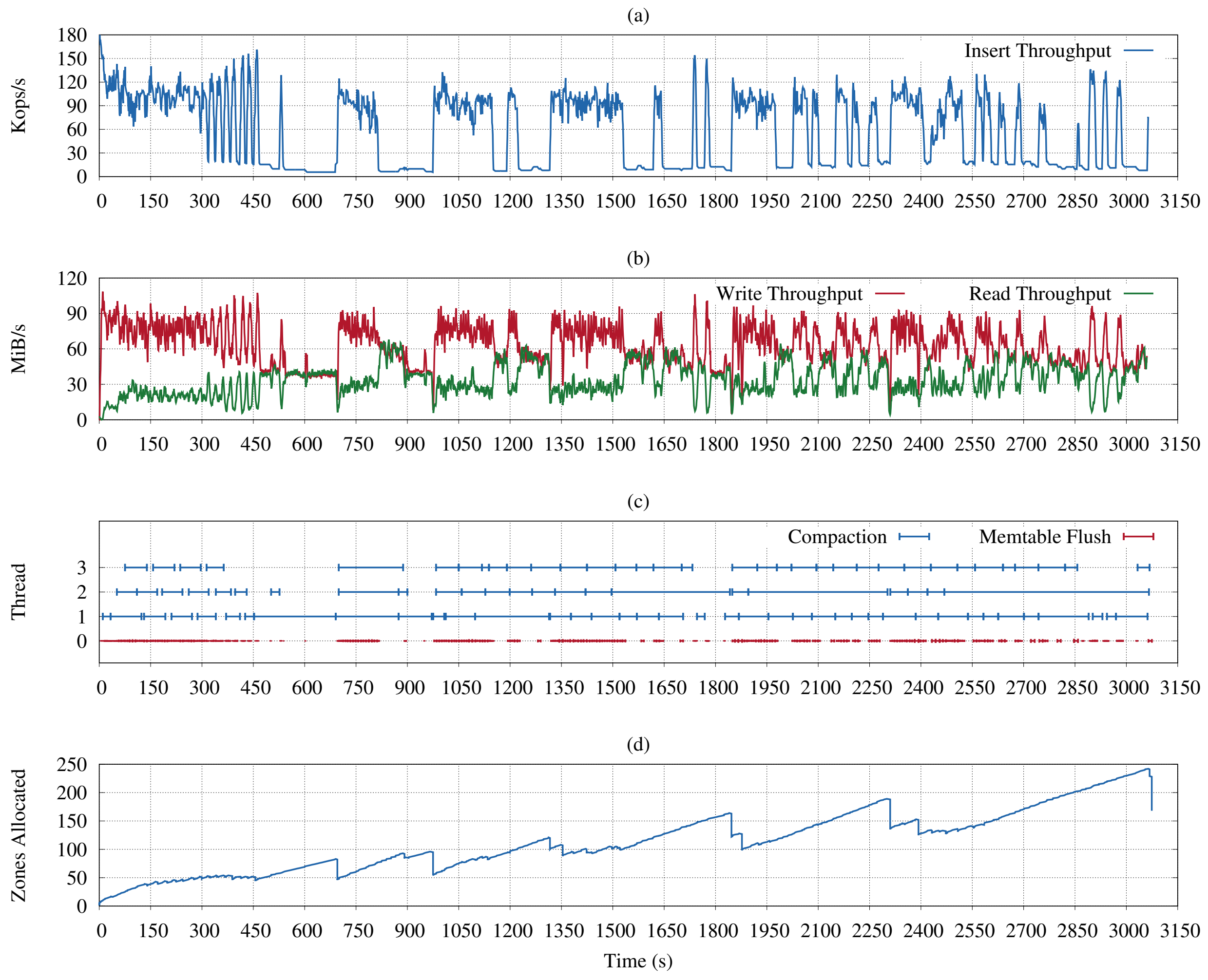

Figure 12: (a) Transaction throughput graph of the benchmark on RocksDB/HM-SMR implemented using asynchronous I/O. (b) Read and write throughput of HM-SMR drive during the benchmark run. (c) The ongoing compaction and memtable flushing processes during the benchmark run. (d) Number of zones allocated during the benchmark run.

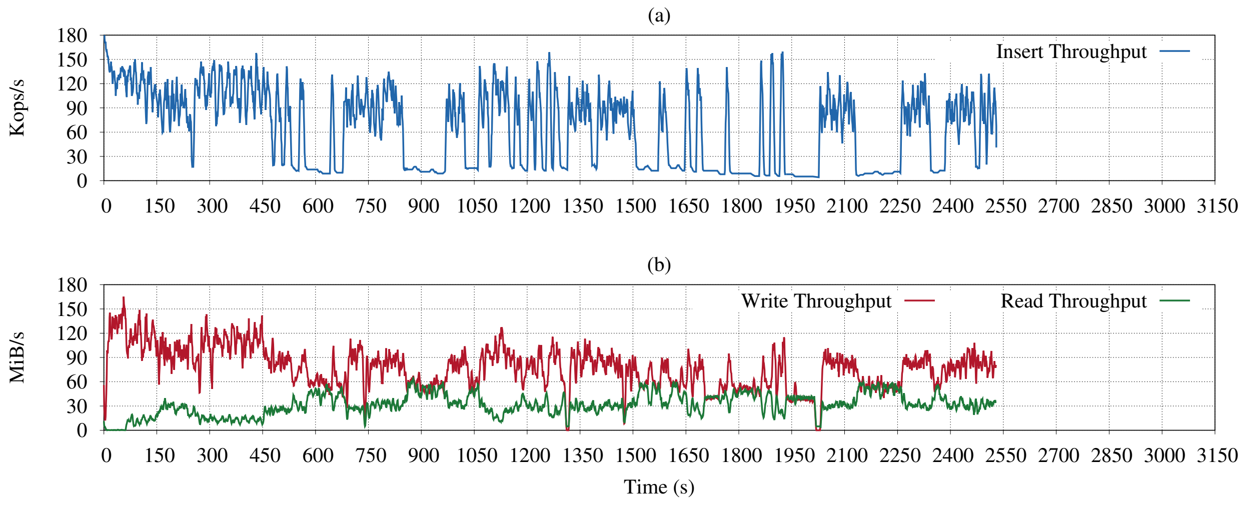

Figure 13: (a) Transaction throughput graph of the benchmark on RocksDB/HM-SMR implemented using asynchronous I/O and caching. (b) Read and write throughput of HM-SMR drive during the benchmark run.

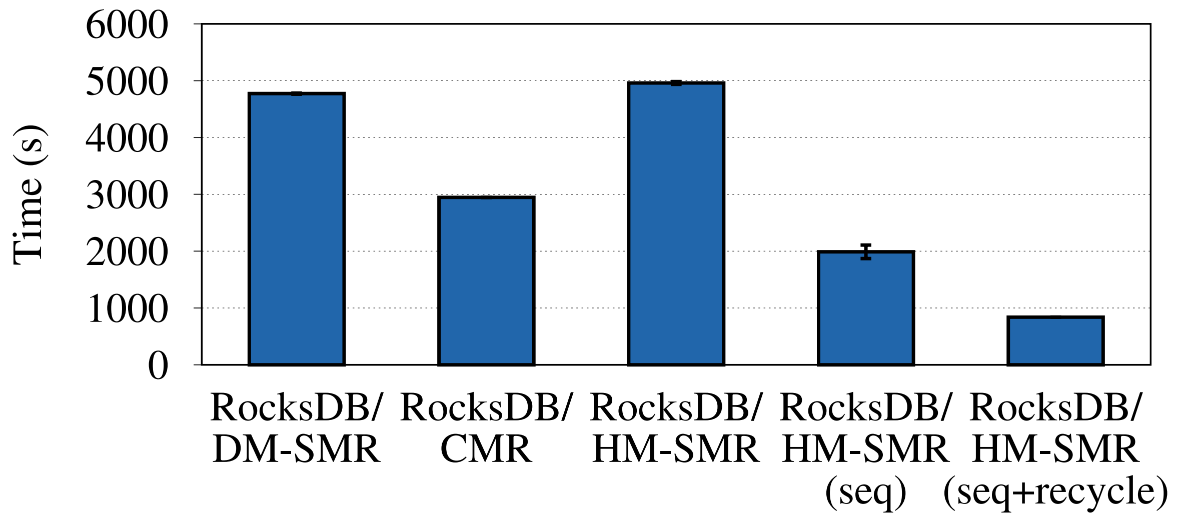

Figure 14: Benchmark runtimes of the baselines and RocksDB/HM-SMR iterations. The benchmark performs 150,000 **synchronous** inserts of key-value pairs of size 20 bytes and 400 bytes, respectively.

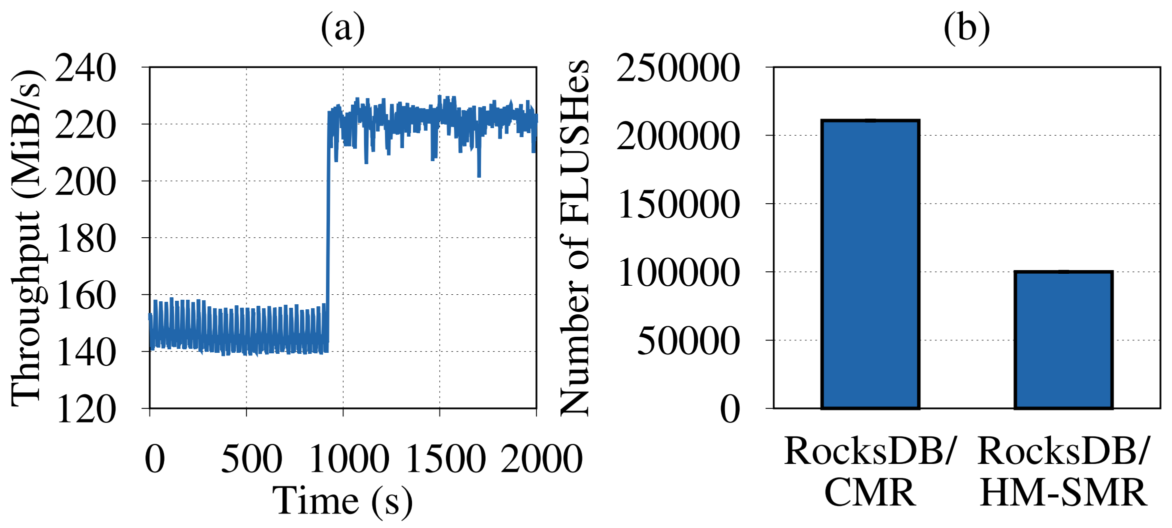

Figure 15: (a) The throughput of the HGST HM-SMR drive during the first 2,000 seconds of a sequential write starting at LBA 0. (b) The number of FLUSH operations issued to disk during the benchmark on RocksDB/CMR and RocksDB/HM-SMR.

asynchronous inserts. Thus, we now discuss two optimizations improving the throughput of synchronous inserts three-fold.

We use the same baselines as before, but modify the benchmark to do 100,000 inserts, given the slowness of synchronous inserts. The left two bars in Figure 14 shows the runtimes of our baselines under this workload, while the center bar shows the runtime of RocksDB/HM-SMR with optimizations we have made so far to BlueFS. Although it is almost as fast as RocksDB/DM-SMR, it is 40% slower than RocksDB/CMR. This slowness stems from the combination of two facts: (1) So far BlueFS allocates space for the WAL in conventional zones due to rewrites happening to the WAL, and (2) in HM-SMR drives the conventional zones logically start at lowest LBAs but they are physically mapped to the center of the drives.

We have already mentioned that it is a bad idea to place the WAL in a conventional zone because it will cause

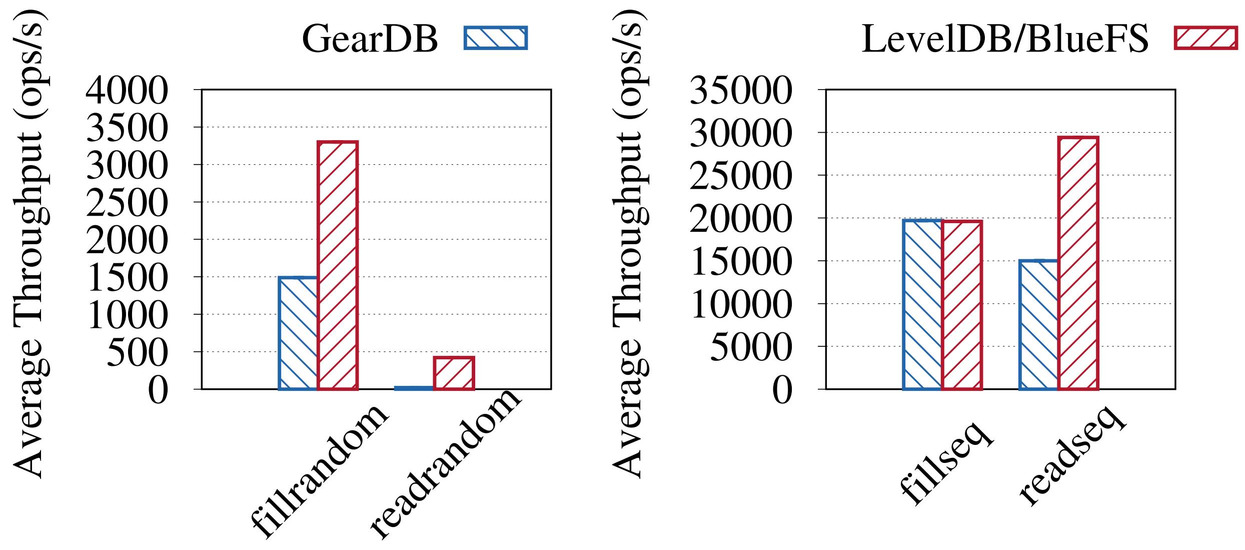

Figure 16: Average operation throughput of GearDB, and stock LevelDB on BlueFS running on HM-SMR drive. The benchmarks randomly (fillrandom) and sequentially (fillseq) insert 20 million key-value pairs of size 16 bytes-4 KiB to the database, and query 1,000,000 keys randomly (readrandom) and sequentially (readseq) on a database that was built with random inserts. GearDB numbers taken from the respective publication [1].

seeks, especially for synchronous inserts (§ 3.3). However, since BlueFS starts allocating sequential zones right next to the conventional zones, there shouldn't be any seeks. But because of the aforementioned fact (2), there is. Figure 15 (a) zooms into the first 2,000 seconds of Figure 5 (a) for the HGST HM-SMR drive, where we observe that the throughput at the starting LBAs—which are the conventional zones—is about 150 MiB/s. This indicates that the conventional zones, although logically located at the lowest LBAs, are physically mapped to zones at the center of the drive. This convention of mapping conventional zones to the center of the drive is followed by other vendors as well, based on the assumption that often-overwritten metadata would be written to conventional zones, therefore they should be equidistant to the rest of the drive. Hence, given that BlueFS journal is located at the sequential zone, every write operation results in a synchronous write to the WAL file located at the middle diameter (MD) followed by a synchronous journal update to BlueFS journal located at a sequential zone at the outer diameter (OD).

After we apply our special format for the WAL file (§ 4) and place it in the sequential zone, RocksDB/HM-SMR (denoted by RocksDB/HM-SMR/seq in the graph) is 48% faster than RocksDB/CMR, as Figure 15 shows. This is because unlike BlueFS that has its journal inline, XFS keeps its journal at the center of the partition and it still has to pay a seek for synchronous writes.

Finally, we apply the RocksDB optimization that preallocates WAL files (§ 4). As a result of this change, Figure 15 (b) shows that the number of expensive `FLUSH` operations to the drive is halved. With these two optimizations, RocksDB/HM-SMR (denoted by RocksDB/HM-SMR/seq+recycle in the graph) is three times faster than RocksDB/CMR, as Figure 14 shows.

5.8 Comparison with Other Systems

We compare stock LevelDB on BlueFS with GearDB, a recent LevelDB-based key-value store for HM-SMR drives designed to deliver high performance [1]. We repeat the benchmarks described in the GearDB paper on an identical setup with 32 GiB of RAM, Seagate ST13125NM007 HM-SMR drive, and 4.15.0-33-generic kernel.

Figure 16 shows that LevelDB/BlueFS is 2.2×, 21×, and 2× faster than GearDB in `fillrandom`, `readrandom`, and `readseq` benchmarks, respectively. While it is hard to know the reason behind the large performance gap without examining GearDB source in detail (not available yet), we know that it uses `libzbc`, small 4 MiB SSTs, and we guess

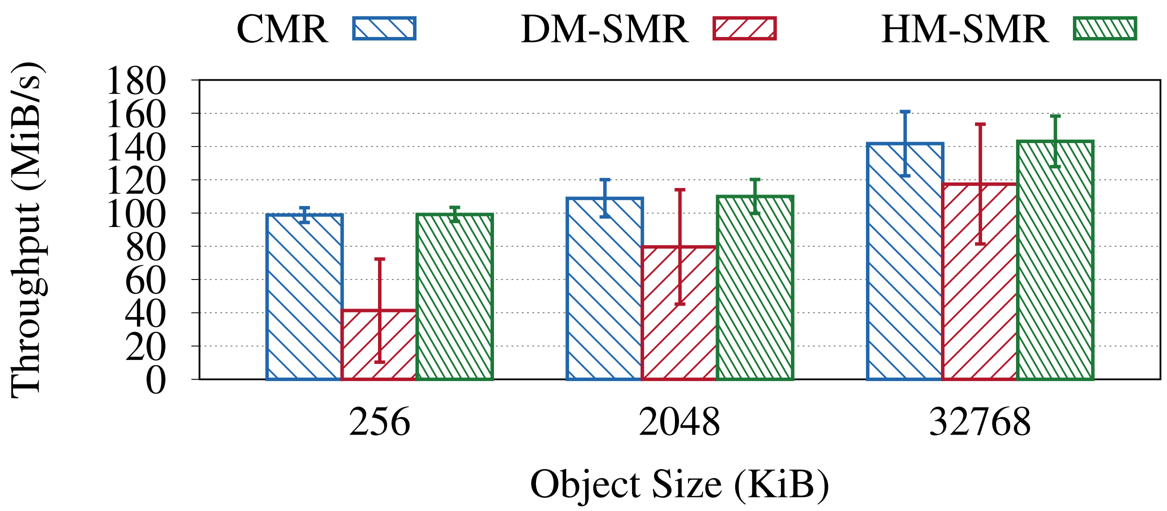

Figure 17: The write throughput of a small Ceph cluster with metadata being stored on a CMR, DM-SMR, and HM-SMR drives. The throughput in the DM-SMR case is 141%, 38%, and 22% lower for 256 KiB, 2 MiB, and 32 MiB object writes and has a larger variance than CMR and HM-SMR cases.

that it is not caching SSTs and uses stock LevelDB I/O sizes that are too small. LevelDB on BlueFS, on the other hand, is configured with large SST size (`max_file_size` set to 256 MiB), and through BlueFS indirectly uses large I/O sizes, prefetches SSTs for fast sequential reads and caches them after writes for fast reads during compaction.

The `fillseq` benchmark, where the systems have similar performance, streams sequential writes to the drive and results in no compaction: key-value pairs are buffered in memory and dumped into SSTs already ordered, which means they are simply moved to the lower levels without compaction. Additionally, since in LevelDB, unlike in RocksDB, writes to WAL are batched and written in large chunks, GearDB's reliance on `libzbc` does not result in a bottleneck (§ 3.2). We omit YCSB benchmark results where LevelDB on BlueFS also outperforms GearDB. Finally, since GearDB already outperforms SMRDB [37] in all benchmarks, transitively, LevelDB on BlueFS outperforms SMRDB.

5.9 Ceph Experiments

To demonstrate the utility of BlueFS, we next plug it into Ceph, a widely-used distributed open-source storage system [46]. The latest Ceph backend, BlueStore [45], runs on raw hard drives and has the ability to store data and metadata on separate drives. BlueStore uses RocksDB as a metadata store and as a write-ahead log.

We setup a 3-node Ceph cluster as an object store with the BlueStore backend, and configure it to store data and metadata on separate drives. We run three experiments where we always store data on a CMR drive and alternate storing metadata on a CMR drive, DM-SMR drive, and HM-SMR drive. For CMR and DM-SMR cases we use stock BlueStore code that runs RocksDB on a raw block device, and for the HM-SMR case we run RocksDB on BlueFS. In each experiment we write small (256 KiB), medium (2 MiB), and large (32 MiB) objects to the object store from a single client using 64 threads. Figure 17 shows that when the metadata is stored on DM-SMR drive, the throughput is 141%, 38%, and 22% lower for small, medium, and large objects respectively, from when it is stored on HM-SMR drive, and it has a large variance. When metadata is stored on CMR and HM-SMR drives, however, the throughput is similar and has lower variance.

The speedup of RocksDB on HM-SMR drive that we observed before does not directly translate to Figure 17 because most of the I/O is directed at the CMR drive that stores object data. While the metadata traffic is not large enough to demonstrate the advantage of the HM-SMR drive, we have enabled Ceph to successfully store metadata on HM-SMR drive with zero overhead, making it one step away from fully leveraging the high bandwidth and capacity advantage of SMR. We plan to tackle storing object data in HM-SMR drives in followup work.

6 Related Work

Research into LSM-Trees spans a range of topics [28], such as reducing write amplification [6,38,48], optimizations for special workloads [33, 40], automatic tuning [10, 26, 49], and adapting LSM-Trees to emerging hardware platforms [8,21,25,27,29,31,37,44]. Our work falls into the latter category, and within that we limit our discussion to the work that adapts LSM-Trees to SMR drives. We believe the other algorithmic techniques are complementary to our work and can be applied to LSM-Trees running on BlueFS.

GearDB [1], a LevelDB-based key-value store for HM-SMR drives, avoids cleaning by introducing a new data format that places 4 MiB SSTs of the same level (§ 2.1) on the same zone, and a new compaction algorithm that compacts all SSTs in the zone at the same time, avoiding fragmentation. BlueFS avoids cleaning by using a simpler technique of mapping SSTs sizes to zones, and uses a series of systems techniques to achieve a significantly better performance.

ZEA [31] introduces a zone-based extent allocator targetted at general-purpose file systems. ZEA sequentially interleaves data and metadata within a zone and demonstrates its usability by building ZEAFS file system and running a modified version of LevelDB on top. ZEA API leaves cleaning of fragmented zones to the consumer. BlueFS, on the other hand, handles cleaning automatically and enables an unmodified LevelDB to utilize SMR.

SMRDB [37] customizes LevelDB to run on an emulated SMR drive by introducing a custom data format and reducing the number of levels to two and evaluates the results an emulated SMR drive. BlueFS is similar to SMRDB in that SMRDB also maps SSTs to ”bands”. However, unlike SMRDB, BlueFS (1) does not require any changes to LevelDB to run on HM-SMR, (2) can enable other LevelDB-based LSM-Trees implementations to run on HM-SMR, and (3) solves many practical systems challenges that SMRDB did not face with an emulated SMR drive.

7 Conclusion

BlueFS enables LevelDB-based LSM-Tree software to use HM-SMR drives with minimal overhead and no code changes. By specializing data placement and I/O sizing to LSM-Tree file access patterns and HM-SMR drive characteristics, BlueFS avoids the space and performance inefficiency of previous approaches. For example, unmodified RocksDB performs random inserts 64% faster atop BlueFS than atop XFS, when storing data on an SMR drive. BlueFS even makes unmodified LevelDB 2–20× faster than GearDB, a recent key-value storage designed for HM-SMR drives. LSM-Trees have become the predominant key-value store mechanism, and also a core component of many scale-out storage systems and databases. By providing a solution for using HM-SMR drives with LSM-Trees, BlueFS can help such systems to address the ongoing transition of the hard drive industry to SMR.

We view this as a crucial step in the widespread adoption of performant HM-SMR drives, the deployment of which is currently limited to enterprises that can scale the barrier of their backwards-incompatible interface. We expect such widespread adoption to benefit everyone by reducing the cost of these devices.

References

[1] GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (Boston, MA, 2019), USENIX Association.

[2] Aghayev, A., and Desnoyers, P. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, USA, Feb. 2015), USENIX Association, pp. 135–149.

[3] AGHAYEV, A., TS'O, T., GIBSON, G., AND DESNOYERS, P. Evolving Ext4 for Shingled Disks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 105–120.

[4] ANANDTECH—GANESH T. S. Western Digital Stuns Storage Industry with MAMR Breakthrough for Next-Gen HDDs. https://www.anandtech.com/show/11925/western-digital-stuns-storage-industry-with-mamr-breakthrough-for-nextgen-hdds, 2017.

[5] AXBOE, J. Flexible I/O Tester. git://git.kernel.dk/fio.git, 2016.

[6] BALMAU, O., DIDONA, D., GUERRAOUI, R., ZWAENEPOEL, W., YUAN, H., ARORA, A., GUPTA, K., AND KONKA, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 363–375.

[7] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)* (2006), pp. 205–218.

[8] CHAO, L., AND ZHANG, T. Implement Object Storage with SMR based key-value store. https://www.snia.org/sites/default/files/SDC15_presentations/smr/QingchaoLuo_Implement_Object_Storage_SMR_Key-Value_Store.pdf, 2015.

[9] COMER, D. Ubiquitous B-Tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121–137.

[10] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 79–94.

[11] DONG, S. Direct I/O Close() shouldn't rewrite the last page. https://github.com/facebook/rocksdb/pull/4771, 2018.

[12] ESCRIVA, R., WONG, B., AND SIRER, E. G. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 25–36.

[13] FACEBOOK. RocksDB. https://rocksdb.org/, 2018.

[14] FACEBOOK, INC. A RocksDB storage engine with MySQL. http://myrocks.io/, 2018.

[15] FACEBOOK INC. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, 2018.

[16] FACEBOOK INC. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2018.

[17] GHEMAWAT, S., AND DEAN, J. LevelDB. https://github.com/google/leveldb.

[18] GIBSON, G., AND GANGER, G. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory, Apr. 2011.

[19] INCITS T10 TECHNICAL COMMITTEE. Information technology - Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, American National Standards Institute, Inc., Sept. 2014. Available from http://www.t10.org/drafts.htm.

[20] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 309–320.

[21] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 993–1005.

[22] KRYDER, M., GAGE, E., MCDANIEL, T., CHALLENER, W., ROTTMAYER, R., JU, G., HSIA, Y.-T., AND ERDEN, M. Heat Assisted Magnetic Recording. *Proceedings of the IEEE 96*, 11 (Nov. 2008), 1810–1835.

[23] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[24] LAWRENCE YING, T. T. Dynamic Hybrid-SMR: an OCP proposal to improve data center disk drives.

[25] LI, Y., HE, B., YANG, R. J., LUO, Q., AND YI, K. Tree Indexing on Solid State Drives. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 1195–1206.

[26] LIM, H., ANDERSEN, D. G., AND KAMINSKY, M. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 149–166.

[27] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 133–148.

[28] LUO, C., AND CAREY, M. J. LSM-based Storage Techniques: A Survey. *CoRR abs/1812.07527* (2018).

[29] MACKO, P., GE, X., JR., J. H., KELLEY, J., SLIK, D., SMITH, K. A., AND SMITH, M. G. SMORE: A Cold Data Object Store for SMR Drives (Extended Version), 2017.

[30] MAGIC POCKET & HARDWARE ENGINEERING TEAMS. Extending Magic Pocket Innovation with the first petabyte scale SMR drive deployment. https://blogs.dropbox.com/tech/2018/06/extending-magic-pocket-innovation-with-the-first-petabyte-scale-smr-drive-deployment/, 2018.

[31] MANZANARES, A., WATKINS, N., GUYOT, C., LEMOAL, D., MALTZAHN, C., AND BANDIC, Z. ZEA, A Data Management Approach for SMR. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (Denver, CO, USA, June 2016), USENIX Association.

[32] MONGODB, INC. MongoDB: Open Source Document Database. http://www.mongodb.com/, 2018.

[33] MUTH, P., O'NEIL, P., PICK, A., AND WEIKUM, G. The LHAM Log-structured History Data Access Method. *The VLDB Journal 8*, 3-4 (Feb. 2000), 199–221.

[34] OLAMENDY, J. C. An LSM-tree engine for MySQL. https://blog.toadworld.com/2017/11/15/an-lsm-tree-engine-for-mysql, 2017.

[35] OPENSTACK FOUNDATION. 2017 Annual Report. https://www.openstack.org/assets/reports/OpenStack-AnnualReport2017.pdf, 2017.

[36] PALMER, A. SMR in Linux Systems. In *Vault Linux Storage and File System Conference* (Boston, MA, USA, Apr. 2016).

[37] PITCHUMANI, R., HUGHES, J., AND MILLER, E. L. SMRDB: Key-value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference* (New York, NY, USA, 2015), SYSTOR '15, ACM, pp. 18:1–18:11.

[38] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 497–514.

[39] REINSEL, D., GANTZ, J., AND RYDNING, J. Data Age 2025: The Evolution of Data to Life-Critical. https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf, 2017.

[40] REN, K., ZHENG, Q., ARULRAJ, J., AND GIBSON, G. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. *Proc. VLDB Endow. 10*, 13 (Sept. 2017), 2037–2048.

[41] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1991), SOSP '91, ACM, pp. 1–15.

[42] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst. 2*, 4 (Nov. 1984), 277–288.

[43] SEAGATE TECHNOLOGY LLC. Breaking Capacity Barriers With Seagate Shingled Magnetic Recording. https://www.seagate.com/tech-insights/breaking-areal-density-barriers-with-seagate-smr-master-ti/, 2014.

[44] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 16:1–16:14.

[45] WEIL, S. Goodbye XFS: Building a New, Faster Storage Backend for Ceph. https://www.snia.org/sites/default/files/SDC/2017/presentations/General_Session/Weil_Sage%20_Red_Hat_Goodbye_XFS_Building_a_new_faster_storage_backend_for_Ceph.pdf, 2017.

[46] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[47] WESTERN DIGITAL INC. ZBC device manipulation library. https://github.com/hgst/libzbc, 2018.

[48] YAO, T., WAN, J., HUANG, P., HE, X., WU, F., AND XIE, C. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Trans. Storage 13*, 4 (Nov. 2017), 29:1–29:28.

[49] YOON, H., YANG, J., KRISTJANSSON, S. F., SIGURDARSON, S. E., VIGFUSSON, Y., AND GAVRILOVSKA, A. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, ACM, pp. 162–173.

[50] ZHU, J., ZHU, X., AND TANG, Y. Microwave Assisted Magnetic Recording. *IEEE Transactions on Magnetics 44*, 1 (Jan 2008), 125–131.