FSTL: A Framework to Design and Explore Shingled Magnetic Recording Translation Layers

Mohammad Hossein Hajkazemi*, Mania Abdi†, Mansour Shafaei*, Peter Desnoyers†
Department of Electrical and Computer Engineering*, College of Computer and Information Science†
Northeastern University
hajkazemi@ece.neu.edu,abdi.ma@husky.neu.edu,shafaei@ece.neu.edu,pjd@ccs.neu.edu

***Abstract*—We introduce FSTL, a Framework for Shingled Translation Layers: a toolkit for implementing host-side block translation layers for Shingled Magnetic Recording (SMR) drives. It provides a Linux kernel implementation of key translation mechanisms (write allocation, LBA translation, map persistence, and consistent copying) while allowing translation policy (e.g. layout, cleaning algorithms, crash recovery) to be implemented in a user-space controller which communicates through an ioctl-based API to the kernel data plane. Due to its use of a journaled write format, FSTL-based translation layers are able to handle synchronous and durable writes to random LBAs at high speed.**

We describe the architecture and implementation of FSTL, and present two FSTL-based translation layers implemented in 400 lines of Python each. Despite the simplicity of the controllers, experiments show our first translation layer performing 1.5x to 10x better than a drive-managed translation layer on trace replay experiments, with performance roughly comparable to the drive-managed device for real file system-based benchmarks; the second translation layer, based on a full-volume extent map, is shown to offer significantly better performance than prior work. Furthermore, we implement and evaluate three cleaning algorithms to demonstrate how FSTL-based translation layers may be readily modified, while still offering the robustness needed for long-duration benchmarks and use.

I. Introduction

Shingled Magnetic Recording, or SMR, is a method of increasing disk track density beyond the paramagnetic limit [1] by overlapping tracks, resulting in an effective track width smaller than the write head width. This enables higher storage densities for a given head and platter technology than conventional recording, where tracks are written at the full width of the write head. Yet this increase in density comes at a price: random writes are no longer possible, as overwriting a sector will also overwrite the corresponding sector in the adjacent "downstream" track.

The result is a device with complex constraints on which sectors can be written at any time without corrupting previously written data. Although these constraints may be met by use of log-structured file systems, development of a high-performance general-purpose file system for multicore servers represents hundreds of person-years of effort; instead, most research in this field [2], [3], [4], [5] has focused on block translation layers. This approach allows effort to be concentrated on the write allocation, metadata persistence, and cleaning mechanisms specific to SMR, as well as being implemented per-spindle (whether in the host or device) and thus avoiding many of the issues [6] which complicate server file system design.

However, translation layer implementations are still challenging. Drive vendors do not provide researchers access to device firmware, so experimentation with on-drive firmware is not an option. Simulation, in turn, does not allow testing with real benchmarks and applications. Kernel-level drivers (e.g. Linux device mapper targets) are perhaps the best option; however although simpler than full-featured file systems, are still difficult to design and debug, making it difficult to replicate research or explore algorithm variations. To address these issues we present FSTL, an open-source[1] Linux framework for SMR translation layer development. FSTL is (a) flexible, allowing a wide variety of translation and cleaning algorithms to be implemented in a user-space controller, (b) fast, allowing benchmarking and comparison of FSTL-based translation layers against real devices, and (c) robust, to accomodate multi-terabyte tests which may run for hours or even days.

FSTL is based on a kernel-space data plane which handles reads and writes, coupled with a user-space control plane responsible for map manipulation and persistence, cleaning, and crash recovery. The data plane is implemented as a Linux device mapper target, providing an in-kernel extent map, write allocation and corresponding map update, and "self-journaling" of writes for robust recovery. A separate set of services is provided for control plane implementation, providing operations to get and set entries in the translation map as well as a data plane-aware copy mechanism for implementing cleaning algorithms and failure recovery. We demonstrate FSTL's power and flexibility by implementing two translation layers: a basic E-region (cache-based) translation layer modeled after the one described in Skylight [7], [3] and a fully extent-mapped translation layer, each requiring less than 400 lines of Python code. We evaluate each translation layer separately, comparing the former against a drive-managed device and the latter against a fully page-mapped translation layer (ZDM-Device-Mapper [8]); performance of the FSTL-based translation layers is found to be comparable or better (sometimes much better) in almost all cases.

The contributions of this work are:

- FSTL, a framework for SMR translation layer implementation,
- an E-region translation layer, TL1, implemented on top of FSTL and its evaluation, demonstrating performance comparable to that of a current drive-managed device.

[1] Available from sssl.ccs.neu.edu/FSTL

TL1 is based on a simple extent-mapped cache and direct-mapped data zone model observed in early drive-managed devices [7],
- a fully extent-mapped FSTL-based translation layer (TL2) and its evaluation, demonstrated higher performance than the comparable ZDM-Device-Mapper, and
- a comparison of cleaning algorithms and other translation layer parameters for TL1, demonstrating the flexibility of the FSTL framework.

The remainder of this paper provides additional background on SMR devices and interfaces, describes the architecture and implementation of FSTL, and presents the two FSTL-based translation layers and their evaluation before surveying related work and concluding.

II. Background

SMR behavior and interfaces: When viewed geometrically, the constraints imposed by SMR are fairly straightforward: when over-writing a sector, data may be lost in any sectors which are adjacent in the "downstream" (shingling) direction. Given the complexity of sector location in modern drives, due to factors such as inter-track skew and slip sparing [9] and variable-density formatting [10] these constraints become difficult to express in terms of LBAs. Additional complications arise due to the need to avoid adjacent track interference in the "upstream" direction due to repeated overwrites, which is handled in non-SMR drives by reading and re-writing the affected sectors.

These issues may be addressed by a block translation layer in the device firmware—much like a flash translation layer—which provides a traditional rewritable block interface, resulting in what is termed a *drive-managed* SMR device. Although most SMR drives on the open market are of this type, a perceived need to allow host control over the performance-critical translation and cleaning processes has led to SCSI and ATA extensions to expose "raw" SMR devices with their constraints. However rather than expose the full complexity of SMR constraints, and perhaps related proprietary design information, industry has converged on a strict write-once model much like the write/erase semantics of NAND flash. More specifically, the ANSI T.10 (SCSI) and T.13 (SATA) standards bodies have defined a zoned device model, where a zone may be written sequentially from beginning to end, and then "reset" back to the beginning, while reads are only allowed to sectors written since the last reset. Internally this corresponds to a set of contiguous tracks comprising each zone, separated by a "guard band" of one or more empty tracks, so that updates to the last track in one zone will not affect the first track in the next zone. Although in theory these zones may be of different sizes, the devices available to date implement a fixed zone size of 256 MiB; in addition, they typically support a small number of conventional re-writable zones at the beginning of the LBA space.

Two classes of device providing this interface are defined: *host-managed* and *host-aware*. Host-managed drives provide additional commands for discovering zones and their current write pointers, and resetting these pointers; operations which do not obey the constraints (i.e. reads beyond the write pointer, writes not at the write pointer) will fail. Host-aware drives are a hybrid, implementing the host-managed commands but falling back to an internal translation layer for non-SMR-conforming operations.

The SMR model of sequential-write-once, erase-before-reuse regions is very similar to that of NAND flash, solutions to which have been extensively studied [11], [12], [13], [14]. However the differing characteristics of disk media lead to significant differences in solution strategies for the two media. In particular, these differences include the following:

- Seek time: While flash performs random operations nearly as fast as sequential ones, disks incur seek penalties equivalent to 1 MB to 2 MB of transfer time for a random I/O.
- Out-of-band data: SSDs are typically able to store some amount of metadata (in addition to ECC) in per-page out-of-band regions, while modern disks have a fixed sector size of 4 KB [15], with any per-sector metadata accessible only to low-level drive firmware.
- RAM-to-media ratio: Although flash is roughly ten times cheaper per gigabyte than DRAM, disk is cheaper by yet by another factor of ten, requiring more memory-efficient mapping strategies to be cost-effective.
- Cleaning unit: The unit of cleaning is substantially larger for SMR disk (256 MB zones vs. 2-16 MB erase units), representing 1-2s of transfer time to write and an equivalent time to read. Cleaning operations may require reading or writing a 256 MiB zone of data 3 or more times [7], potentially resulting in lengthy delays.
- Wear leveling: Flash requires wear leveling, while disk lifetime is not affected by the distribution of write operations [16].

As a result of these characteristics, translation layer strategies appropriate for SMR disk differ significantly from those used for flash. Due to the single disk channel and high seek time, the preferred write allocate strategy is strictly sequential, rather than e.g. first free channel [14]. The online map should be highly memory efficient, yet avoid access to disk when at all possible. Although out-of-place writes are not truly stable until both the data and the metadata are persisted, map updates cannot rely on out-of-band data or seeks to non-data locations after every write. In some implementations this problem is avoided by agressive use of write caching; FSTL instead journals map data in-place in a log-structured fashion (see Section III for more detail). Finally, the removal of the wear leveling constraint allows a wider range of cleaning algorithms, but care must be taken to avoid excessive performance impact due to lengthy cleaning operations.

III. FSTL Architecture and Implementation

FSTL is based on a split control plane/data plane architecture, with FSTL providing the data plane and a user-space *controller* implementing the control plane. The basic architecture is shown in Figure 1. The FSTL data plane is a Linux *device mapper*,

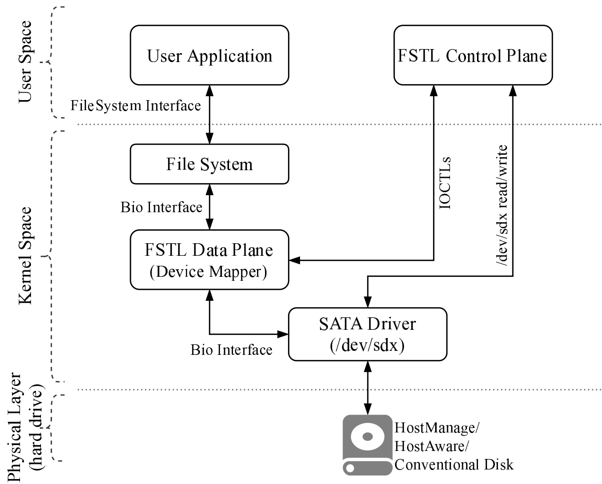


Fig. 1: FSTL framework within IO stack: FSTL is composed of a data plane that handles user application reads/writes, and an interface to a controller implementing a translation layer.

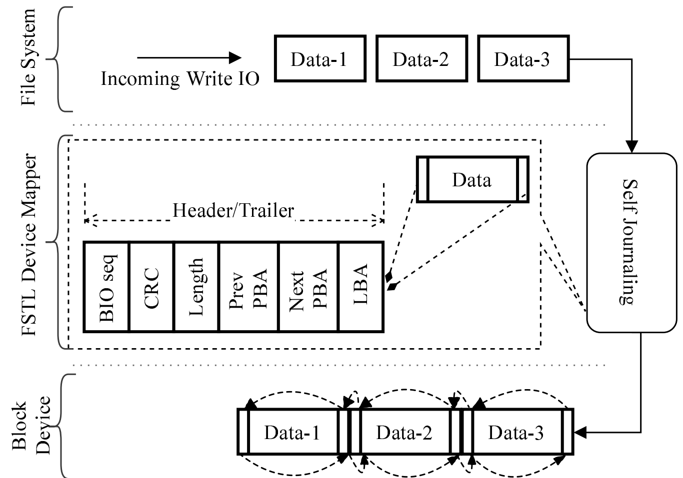


Fig. 2: Self-Journaling: the FSTL data plane prepends and appends a header and a trailer to the incoming write for recovery purposes in case of any crashes.

a form of stackable block driver used to implement facilities such as software RAID and volume management which remap or otherwise process I/O requests between the file system and underlying devices. The data plane receives application I/O requests (e.g. from the file system) and passes them to the underlying SATA or SCSI driver, which sends them to the SMR device. The user-space controller does not intervene in I/O, but uses a special-purpose interface to the kernel component in order to access the translation map, perform cleaning, etc. The controller also directly reads and writes metadata regions of the underlying disk, for persisting map data, reading it at startup, and performing crash recovery.

The FSTL framework provides support for the primary operations of a translation layer: (a) performing allocation and out-of-place writes, (b) maintaining an online translation map for read operations, (c) persisting the map to ensure write durability, and (d) atomic data movement for implementing cleaning operations. We describe FSTL and the solutions it provides to SMR-specific issues in terms of the mechanisms and interfaces it provides for achieving these four translation layer goals.

A. Data plane

Write allocation: FSTL uses a simple log-structured write policy: writes are performed at the current *write frontier*, which advances until the end of a zone; the next zone is selected in order from a free list provided by the user-space controller.

Translation map: Out-of-place write allocation requires keeping track of allocated physical block addresses (PBAs) to the incomming writes (LBAs). Since a full page map (e.g. of 4 KB sectors) would be too large to fit in memory, requiring over 2 GB for an 8 TB device, the FSTL kernel module implements an in-memory extent map, which is updated automatically and consistently on write. With mean write sizes of 10 sectors or more on modern workloads (see Section V) this reduces the map size by an order of magnitude, or even more if the controller implements a hybrid strategy [17] where most of the LBA space is mapped by a few large extents. FSTL allows the controller to leave extents of the LBA space *unmapped*; reads to these regions will return zeros, while writes will populate them with data.

On startup the map is empty, with I/O paused; it is the controller's responsibility to initialize the map (typically from metadata on disk) before allowing I/O to commence. In addition to updating the map, the controller is also able to read the current map; this is used both for checkpointing (i.e. persisting the map in an efficient form) and to locate zones for cleaning.

Map persistence: Persisting translation changes is a crucial problem for any translation layer using out-of-place write, as a future read is not guaranteed to see a write until both the data is written to a new location and the mapping of the LBA to that new location has been made persistent. This is especially a problem for SMR disks, which lack the per-page out-of-band data area or fast random write found in NAND flash. Practical host-based translation layers to date (Tancheff's ZDM-device-mapper [8] and Le Moal's dm-zoned [18], from Seagate and WD respectively) aggressively cache map updates in memory, writing them to disk whenever the file system requests that the device write cache be flushed. This is sufficient to maintain file system consistency on failure, but risks significant data loss as well as incurring substantial seek overhead for metadata-heavy workloads which flush write caches frequently.

In contrast, FSTL uses a "self-journaling" approach, as shown in Figure 2 which interleaves map updates with data writes, prepending a header (indicating the extent length) and appending a trailer (specifying the LBA at which it is mapped) to each write. Much like a file system journal, the header/trailer structure contains a sequence number and CRC in order to identify whether a particular write completed before a crash. Both headers and trailers contain the physical block address (PBA) of the next header/trailer; in the header this is equivalent to the length, while in the trailer it is only needed for the last

entry in a zone, as otherwise it points to the immediately following PBA.

To preserve 4 KB alignment, headers and trailers are 2 KB each, adding a negligible 20-40μs of transfer time to each write. The resulting space overhead for large I/Os is small: for maximum-sized writes (limited on our system to 500 KiB by the underlying Linux AHCI driver) the throughput degradation (and space overhead) due to 4 KiB of overhead per 500 KiB is about 0.8%, which we consider quite acceptable. Although space overhead for small writes is considerable, it is mitigated by two factors: (a) the header is not copied during cleaning, and (b) no one expects to be able to fill a multi-terabyte drive using small I/Os, anyway[2].

B. Control plane

As mentioned above, FSTL is not a complete translation layer—it handles read and write requests, but rather than implementing more complex functions such as cleaning and crash recovery, it provides interfaces to user-space control plane, allowing those functions to be implemented in the user-space. In this subsection we explain how this interface allows implementation of (a) consistent checkpointing and (b) garbage collection or *cleaning*.

Consistent checkpointing: In order to reliably checkpoint the translation map, the controller periodically and also at a clean shut down (1) queries the data plane for the current value of the write frontier, and then (2) retrieves the latest extent map. These form a checkpoint of the current state, and can be written to a reserved region of the disk or to another device. If no writes occur after the write frontier value is retrieved, the checkpointed map will be identical to the in-memory extent map. If additional writes occur after the last checkpoint (i.e. the system crashes before clean shutdown), they may be discovered on startup by following the chain of extent headers starting at the write frontier in order to recover mapping information for those additional writes. We note that in the worst case this process should take no longer than writing the data in the first place; thus recovery time is bounded by the interval between checkpoints.

Cleaning: Cleaning is performed by the controller, which retrieves the extent map in order to select a zone or zones for cleaning. The data plane provides a safe copy command for implementing cleaning: it freezes writes to the affected LBAs, reads the source data by LBA (to account for writes received since the map was retrieved by the controller), and updates the in-memory extent map after writing to the destination. Cleaning writes are not journaled; instead after cleaning a zone the controller should checkpoint the updated map before resetting (i.e. erasing) the zone and passing it back to the data plane. Note that failure before the checkpoint completes will leave the pre-cleaning state unchanged.

During periods of heavy cleaning it is necessary to stall write operations, as they cannot be performed faster than free space is reclaimed by the cleaner. This requires coordination between the controller and the data plane, as writes must be stalled in the kernel, while it is the user-space controller which is aware of how quickly cleaning progresses. If writes are blocked only when the kernel component finishes filling its last free zone, and unblocked again when a new zone is provided by the controller, then many I/Os will be blocked for an entire cleaning cycle, possibly taking many seconds.

To avoid this, the FSTL data plane allows the controller to specify a low-water mark within the last zone, and will stall writes after this point is reached. By gradually moving this limit while cleaning progresses, the controller can allow write operations to be interleaved with the cleaning process, achieving the same long-term throttling of I/O rate but with far lower peak latencies [19].

C. Implementation

The FSTL device mapper target is implemented in about 1100 lines of C code. It registers a custom character device, and is controlled from user space via ioctl system calls directed to this device. This interface is in theory language-agnostic; controllers to date have been written in Python, with a wrapper around the C structure-based interfaces.

The extent map is currently implemented as a red-black tree requiring 64 bytes per entry on a 64-bit architecture. This would for example allow 2 million extents (i.e. a mean extent size of 4 MB for an 8 TB drive) to be mapped in 128 MB of kernel memory. Future use of an optimized B+-tree-like structure will reduce memory usage by a factor of nearly 4; however to allow a finer-grained extent map (with reasonable memory usage) we are implementing a mechanism for faulting map misses to the controller allowing map data to be faulted from disk.

IV. FSTL-BASED TRANSLATION LAYERS

We use FSTL to implement two translation layers; a simple persistent cache-based translation layer, TL1, modeled on the first-generation Seagate algorithm described by Aghayev [7], and a fully log-structured translation layer, TL2, modeled after LFS [20].

A. E-region translation layer: TL1

In TL1 the disk is divided into a small *persistent cache* (or *exception region*, often abbreviated as "E-region") and *data zones*. LBAs are mapped to fixed locations ("home locations") in the data zones much the same way as they are mapped in conventional drives. Updates ("exceptions") are written to the persistent cache in a log-structured fashion. We note that although it is termed as "cache", it might be better described as a *log*, as it is used to persist updates in sequential order. When the cache fills, a cleaning process evicts extents from the cache by merging them back to their corresponding data zones.

As described in Algorithm 1, the cleaning process (a) reads data from the cache, (b) reads the data zone it is to be merged with, (c) writes a copy of the merged zone to a scratch location,

[2]At 100 IOPS, filling an 8 TB conventional drive with 4 KB writes would take nearly 8 months.

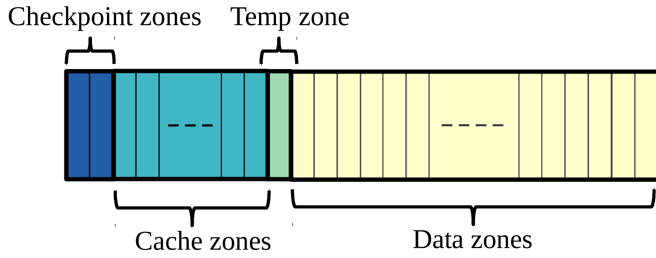


Fig. 3: TL1 On-disk data structure: two checkpoint zones, a small number of cache zones, a temporary zone and a large number of data zones.

Algorithm 1: TL1 controller

```
1  Function TL1Controller() is
2      activeThrshld = zoneSpace
3      idleThrshld = maxZoneSpace
4      while True do
5          emptySpace ← CalcEmptySpace()
6          if emptySpace < activeThrshld or (IsIdle() and
            emptySpace < idleThrshld) then
7              Clean();
8              Checkpoint();
9          end
10     end
11 end
12 Function clean(policy) is
13     if policy == FIFO then
14         cacheZnToClean ← FindOldestDirtyZ();
15     else if policy == MinData then
16         cacheZnToClean ← FindZwithMinData();
17     else if policy == MinAssoc then
18         cacheZnToClean ← FindZwithMinAsso();
19
20     for extent in cacheZnToClean do
21         homeZone ← FindZDataHome(extent)
22         ReadModifyWrite(homeZone)
23     end
24     AddToFreeZonelist(cacheZnToClean)
25 end
26 Function checkpoint() is
27     map ← GetMap();
28     wf ← GetWF();
29     freeZones ← GetFreeZones();
30     chkpt ← CreateChkpt(map, wf, freeZones);
31     WriteHeader();
32     WriteCheckpoint(chkpt);
33     WriteTrailer();
34 end
```

and then (d) overwrites the data zone. The additional write in (c) is needed due to the destructive nature of SMR writes—without this step, if power was lost during step (d) then some amount of data ahead of the write head could be lost.

The controller periodically polls data plane statistics to determine when to begin cleaning; background cleaning is triggered when disk goes to the idle mode and it continues till the cache is nearly empty unless disk becomes active again, while foreground cleaning (i.e. during active I/O) will not be done unless there is less than a zone's worth of space remaining in cache. The controller then reads the current map and selects a cache zone to clean; each extent cached in that zone is then merged back to its "home" location in a read-modify-write process. When cleaning an extent from some zone Z, all cached extents from that zone will be read from cache and merged.

The data copy process is performed using the FSTL copy operation, which identifies source extents by LBA and destinations by physical block address. If an address has been overwritten (and thus moved physical location) between when the cleaner reads the map and when it issues the copy command, this ensures that the correct data values are still copied; in either case, after the LBA has been moved, the physical block address which the controller *thought* it was moving it from will be empty.

If this merge were performed in memory and written back to the data zone, it would result in a window of vulnerability from when the data zone was "erased" (i.e. the write pointer reset) to when the zone was completely rewritten; a crash during this window could result in significant data loss. To prevent this, the merged zone is first saved to a temporary location before over-writing the data zone itself.

We implement and evaluate three different zone selection algorithms for this cleaning process (see algorithm 1 for more details):

- **fifo:** Cache zones are used and cleaned in strict round-robin order. This is the algorithm used by the device analyzed in Skylight [7]; it is simple but may not be as efficient as other strategies.
- **min_valid:** This is the classic Greedy algorithm from the FTL literature, where the cache zone with the fewest remaining valid sectors is chosen.
- **min_assoc:** Since cleaning time for this translation layer is dominated by the time to read and re-write data zones, this strategy chooses the cache zone with the fewest data zones represented among its cached extents. (similar strategies have been used for flash—e.g. see Cho's K-Associative Sector Translation [21].)

The TL1 controller is implemented in 400 lines of Python, plus a 230-line Python wrapper for translating the C-based structures used in the kernel interface. The translation layer is stable under long-term testing using the ext4 file system, although further testing would be needed to prove its suitability for production deployment.

B. Full extent-mapped translation layer: TL2

TL2 maintains a set of *data zones*, comprising almost all of the disk as shown in Figure 4. It is analogous to a fully page-mapped FTL, such as DFTL: LBAs have no pre-defined home locations, but instead stay where they are written until the disk fills up and the zone is cleaned to make room for new writes. Until an LBA is written it has no location, either; upon initialization all LBAs in a TL2 volume are un-mapped, leaving the entire disk as free space to be allocated for writes. As described in Algorithm 2, the cleaning process (a) reads data from one or a few data zones, (b) merges them, (c) writes back the merged data to the log head.

Algorithm 2: TL2 controller

```
1  Function TL2Controller() is
2  |  while True do
3  |  |  emptySpace ← CalcEmptySpace();
4  |  |  if emptySpace < threshold then
5  |  |  |  Clean();
6  |  |  |  Checkpoint();
7  |  |  end
8  |  end
9  end
10 Function clean() is
11 |  dataZonesToClean ← FindDirtyZones();
12 |  ReadLiveData(dataZonesToClean);
13 |  WriteBackDataToLogHead();
14 |  AddToFreeZonelist(dataZonesToClean);
15 end
```

Algorithm 3: Startup - FSTL controller deamon

```
1  Function startup() is
2  |  ReadSuperBlock();
3  |  ChaseDownLastCheckpoint();
4  |  wf, map, freeZones ← ReadChkpoint();
5  |  ReplayJournalRetrieveMap(wf);
6  |  FeedTargetWfFreeZones();
7  |  UnblockIOPassthrough(); //Allows IO to pass
   |    through the FSTL device mapper
8  end
```

Since large disk drives are commonly used for archival or backup purposes where files are added but never deleted, this means that over the lifespan of the drive the total volume of write traffic may be only fractionally larger than the disk itself, accounting for repeated overwrites of metadata. In this scenario cleaning will not begin until the drive is nearly full, and the total amount of cleaning required over the lifespan of the drive may be only a fraction of its capacity. (This is in contrast to SSDs, where the expectation is that they will support a lifetime write volume many times greater than their size, with cleaning adding a workload-dependent overhead to nearly all write traffic.)

As the size of the extent-map can grow over time, large memory requirement would be an issue for TL2; even with planned improvements in map memory efficiency (16 bytes per extent instead of 64) the kernel memory usage for a full 10 TB volume would be excessive. This does not prevent its evaluation, however; as shown in Table I even the longest traces available to us [22] have less than half a million entries.

C. Checkpointing and Recovery

TL1 and TL2 share the same logic for checkpointing the extent map and recovering it on startup. Algorithm 3 shows the operations done at startup. A region at the beginning of the disk is reserved for map checkpoints; each checkpoint includes the write frontier, a full extent map, and a sequence number for locating the most recent checkpoint. If only sequential-write zones are available on a drive, then this checkpoint region must be two zones long in order to avoid the potential for catastrophic loss when resetting the write pointer to reuse a zone, as zone data is not accessible after write pointer reset. If random-write zones are available, however, then the checkpoint region may be a single zone or less.

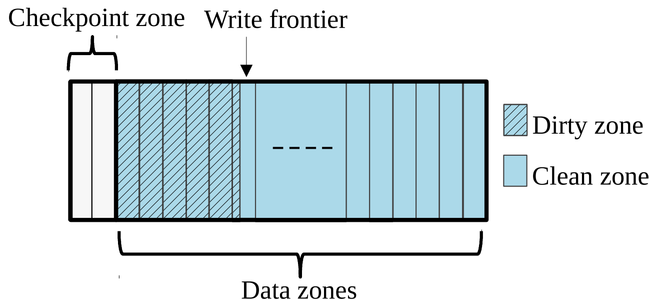


Fig. 4: TL2 on-disk data structure: two checkpoint zones and a large number of data zones.

TL1 and TL2 use the same header and trailer for checkpoints as used by the data plane, allowing them to readily locate the most recently persisted checkpoint. Map checkpoints are protected by a CRC, much like write headers and trailers. As a result checkpoint writes are atomic; if a crash occurs while writing a checkpoint then the incomplete record will be ignored, and recovery will begin from the previous checkpoint.

V. EVALUATION

To demonstrate that FSTL can perform fast enough as a framework to implement translation layers for SMR drives, we evaluate the performance of TL1 and TL2 translation layers and compare them with the equvalent indusrial ones. To this end we use the following set of experiments.

Trace-based replay, using the `fio` I/O testing tool [23], replaying several of the well-known MSR Cambridge traces [22] as well as traces provided by an industrial partner. In all cases direct I/O mode was used.

Filebench [24], a file system-level benchmark which includes a large number of configurations emulating different system behaviors. The following Filebench workload configurations were tested:

`randomwrite`: creates a single large file prior to measurement, then makes 8K random writes to the file from a single thread.

`singlestreamwrite`: creates a single large file prior to measurement, and then performs 1MB sequential overwrites to this file from a single thread.

TABLE I: Map size required in TL2 implementation for a number of MSR Cambridge and our industrial partner traces.

wrkld	extent count	wrkld	extent count
src1_2	18143	**SW1-R6Dv1**	124253
src2_2	349483	**Hadoop1-R6Lv1**	92974
wdev_1	1174	**Backup2-R6Dv1**	62317
proj_0	79815	**Hadoop1-R6Dv1**	72790
usr_0	29817	**DB1-R6Lv1**	6645

TABLE II: Filebench configuration used to becnhmark TL1 and TL2.

	file count	file size	IO size	thread count	mean append size	run time
`randomwrite`	1	15G	8K	1	16	600sec
`singlestreamwrite`	1	15G	1m	1	NA	600sec
`fileserver`	10000	128k	1m	50	16k	600sec
`varmail`	1000	16k	1m	50	16	600sec

`fileserver`: a mix of create, delete, append, read, and write operations on multiple threads, designed to emulate a network file server workload.

`modified fileserver`: the `fileserver` workload issuing `fsync` after every `append` operation.

`varmail`: a multi-threaded mix of create, write, fsync, open, and read operations designed to emulate a mail server storing messages in individual files.

The particular filebench parameters used are shown in Table II. For all tests the `ext4` file system was used, with default mkfs and mount options.

Compilebench [25], a tool which mimics the stages involved in copying, patching, and compiling the Linux kernel. (This benchmark is not surprisingly a favorite of many Linux kernel developers.) Each source tree is approximately 1 GB (after "compilation"); the specific parameters used were 50 iterations, with 30 source trees in each iteration. Again the `ext4` file system was used, with default mkfs and mount options.

The test system was equipped with an Intel i3-4340 CPU and H87 chipset, using the motherboard AHCI controller, and running Linux kernel 4.10.0 unless specified otherwise. Tests were performed with a Seagate ST8000AS022 8 TB 5900 RPM host-aware drive, firmware revision ZN01. This drive was used in host-aware mode for translation layer tests, as well as being used in drive-managed mode when indicated below. Except when indicated otherwise, drive write cache and lookahead were enabled, in keeping with standard Linux practice.

A. *TL1*

TL1 is modeled on the first-generation Seagate translation layer described in Skylight [7]; however, we compare it with the second-generation Seagate translation layer used by the ST8000AS022 in drive-managed mode (referred to as "DM" below) as it is more up-to-date. Since the first 64 zones (16 GB) of these drives are conventional (non-shingled) zones requiring no translation layer, these LBAs were skipped in all experiments. The drive-managed persistent cache was measured (using techniques from [19]) and found to be 27 GB; unless specified otherwise, the TL1 persistent cache size was set to this value to allow fair comparison. Traces were selected which were long enough to exhaust the persistent cache for both translation layers and thus incur cleaning overheads; "min_assoc" was chosen as the cleaning policy (see subsection V-C for more details). Traces were truncated as needed so that typically the device or translation layer was not actively cleaning (at greatly reduced throughput) for more than half of each experiment run.

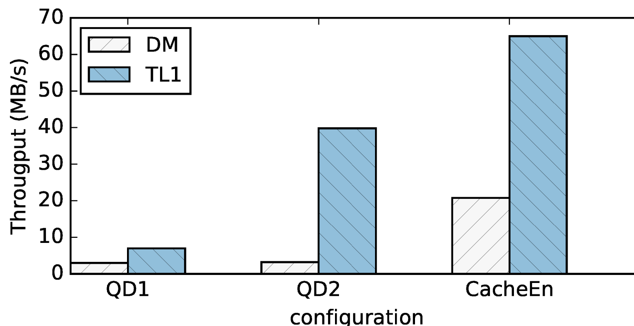


Fig. 5: Performance comparison of TL1 vs. drive-managed translation layer (DM) for `src2_2` trace, queue depth 1 and 2 (write cache disabled) and queue depth 1, write cache enabled.

As a result, only a small fraction of the longer MSR[3] traces were used.

Trace replay: In Figure 5 we see TL1 and DM results for `src2_2` with varying settings for write cache and queue depth: fully synchronous—i.e. queue depth 1 (QD1)—with write cache disabled, queue depth 2 (QD2) again with write cache disabled, and write cache enabled (with queue depth 1). Here "write cache" refers to on-disk RAM buffering; when it is enabled, writes are acknowledged before they are written to disk, and may be lost in a crash unless followed by a FLUSH or FUA command; file systems such as ext4 send such commands after e.g. journal writes. In the TL1 case the write cache was disabled or enabled on the underlying host-aware drive, and FSTL passes FLUSH commands through from the file system to the drive.

"Queue depth 2, write cache disabled" represents best-case conditions for TL1, as the high persistent write performance provided by the self-journaling mechanism results in nearly an order of magnitude of improvement over drive-managed. In the other two cases performance improvements are still substantial on this write-heavy workload, at 2x for queue depth 1 with write cache disabled, and roughly 3x with write cache enabled.

In Figure 6, we see TL1 throughput compared to drive-managed mode for a series of additional traces, in each case with write cache and look-ahead both disabled and a queue depth of 1; in each case TL1 is seen to perform roughly twice as fast as the internal drive translation layer. We explore the reasons for this performance disparity in more detail in Figure 7, where we see per-operation latency for `src2_2` trace. The

[3]Since the systems on which the MSR Cambridge traces were collected had been live for considerable lengths of time before the beginning of trace collection, any interval of a trace (e.g. a prefix after truncation) is representative of normal activity.

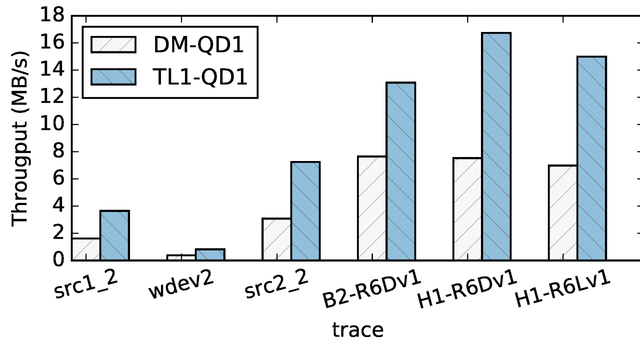
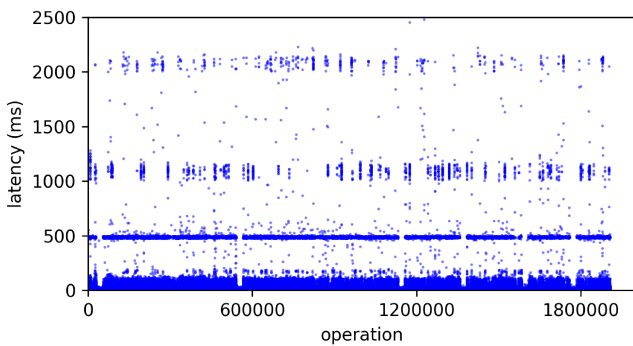
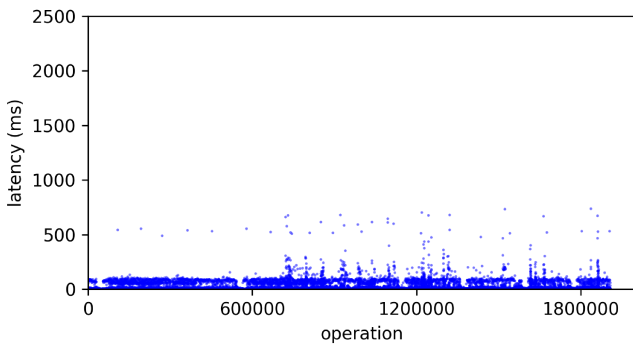


Fig. 6: Performance comparison of drive-managed translation layer (DM) vs. TL1 for various traces with queue depth 1, write cache and lookahead disabled.



(a) DM



(b) TL1

Fig. 7: IO latency of (a) DM and (b) TL1 for `src2_2`: while TL1 services all IOs in less than a second except a single one, in DM there are several IOs that require more than a second (up to 5 second) to complete.

drive-managed translation layer is thought to manage its cache space less efficiently as it begins cleaning earlier in time, and generates far more high-latency events: while TL1 completes all but one I/O in less than one second, with only a small number over 200ms, in drive-managed mode we see many I/Os taking one or two seconds, and a very large number of 500ms ones.

Filebench: Filebench performance for TL1 and drive-managed mode may be seen in Figure 8. TL1 outperforms DM significantly for `randomwrite` and `varmail`, is roughly comparable for `fileserver`, and is drastically slower for `streamwrite`.

We explored this performance disparity by graphing throughput over time (as measured via `strace`) in Figure 9, smoothed with a time constant of 5 seconds (i.e. roughly the maximum latency observed). In Figure 9a and Figure 9b, we see that TL1 and DM show different behaviors when running `randomwrites`; while DM throughput falls rapidly and remains low (1-2 MB/s) till the end of experiment, TL1 throughput starts at 70 MB/s and remains steady for a while; then it drops after nearly 180 seconds and starts fluctuating between 0 and nearly 40 MB/s. Similar to the trace replay experiment, the DM low performance is again due to its poor cache space management and early cleaning. As for TL1, our observation depicts that the performance drop at time 180 second is because the persistent cache is filled up, and consequently the cleaning process is triggered. Unlike the `randomwrite` case, DM shows a significantly higher performance for `streamwrite` although it fluctuates between 100 MB/second and 175 MB/second (see Figure 9d). We believe that the reason for this is that for sequential writes, DM bypasses the media cache and directly write IOs to their home locations. For the TL1 case, as shown in Figure 9c, the performance remains steady for a long time (300 seconds), then drops and starts fluctuating between 0 and 40 MB/second. As can be seen, compared to the `randomwrite` case, `streamwrite` shows a slightly higher performance because the cleaning process starts further in time; that is because `randomwrite` causes more meta-data (header and trailer) write to the cache due to small size IO compared to `streamwrite`.

Among studied Filebench micro-benchmarks and workloads, `varmail` depicts the lowest performance. To have a better insight we looked into operations issued by `varmail` and realized that it calls `fsync` after every single `append` operation. This causes poor utilization of the buffer cache, more stress on the drive and consequently lower performance. Moreover, as the mean append size in `varmail` is 16 KB, a

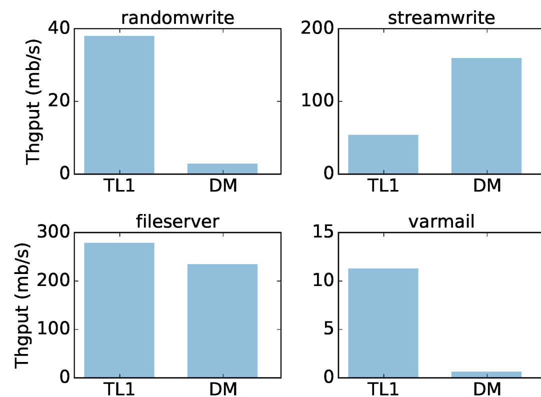
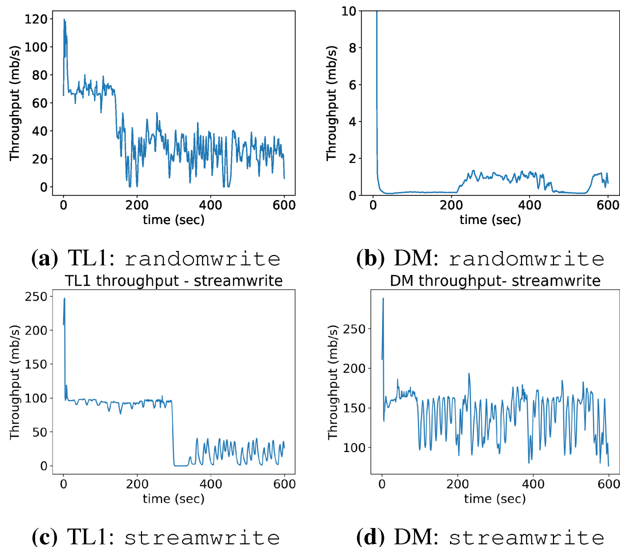


Fig. 8: Filebench performance: TL1 vs. DM.



(a) TL1: `randomwrite` **(b)** DM: `randomwrite`

(c) TL1: `streamwrite` **(d)** DM: `streamwrite`

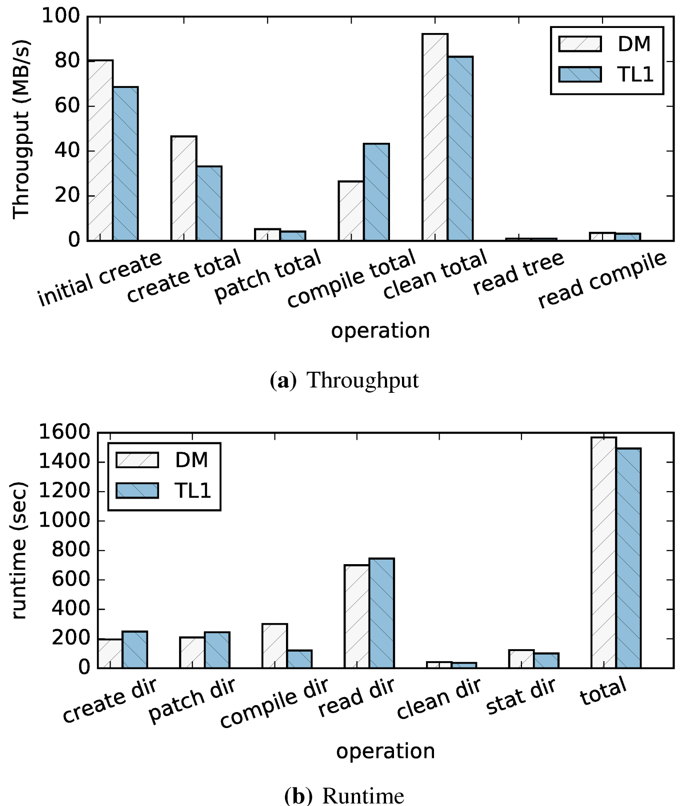
Fig. 9: TL1 and DM performance over time: (a) and (b) `randomwrite`, and (c) and (d) `singlestreamwrite`.

significant number of random writes are issued which in turn results in early background cleaning of DM and therefore a lower performance compared to that of TL1.

`Fileserver` shows higher performance relative to `randomwrite`, `streamwrite` and `varmail` for both TL1 and DM. Moreover, for `Fileserver` case the TL1 and DM performance are almost identical. We believe that the main reason behind this is the impact of extensive use of buffer cache; it is so significant that both TL1 and DM policies impact on performance becomes marginal. To explore this, we added `fsync` to the `fileserver` workload and re-ran the experiments. Results show performance drop in both cases; while TL1 throughput drops by half (around 125 MB/second) the DM throughput falls by factor of 20. To investigate more and validate our hypothesis, we monitored the disk traffic while running unmodified `fileserver`, modified `fileserver` as well as `varmail` workload. We observed a smaller portion of IO traffic serviced by the disk in unmodified `fileserver` case compared to modified `fileserver` and `varmail` cases.

Compilebench: Compilebench results for TL1 and drive-managed are seen in Figure 10, broken down by compilebench phase; results are shown for both throughput and overall completion time. TL1 performance is seen to be roughly identical to drive-managed, with a slightly lower overall completion time (Figure 10(b) right-most bar), although individual phase performance varies significantly between the two.

Contrary to results we observed in trace replay, TL1 only shows a slight improvement relative to drive-managed mode. To investigate more we monitored the individual latency of every IO in both cases and realized that in the drive-managed case a large fraction of I/Os are completed in much less than 1ms, which we suspect is due to merging of I/O requests in the scheduler queue before they reach the driver, a kernel feature



(a) Throughput

(b) Runtime

Fig. 10: Compilebench performance for TL1 and device-managed: (a) throughput, (b) runtime.

not available to bio-based device mapper targets such as FSTL. In addition we note that the internal drive translation layer is the vendor's second-generation algorithm, and has likely undergone considerable optimization for its expected use[4].

B. TL2

We first briefly compare TL2 with TL1. In our trace replay experiments TL2 performance is much better than TL1; e.g. for `src2_2` TL1 throughput is 67 MB/s (Figure 5) vs. 100 MB/s for TL2 (Figure 11). The reason for the higher TL2 performance is that it does not perform any cleaning during the experiment; since it is a fully extent-mapped translation layer which does not map LBAs until they are written, it will not begin cleaning until the entire drive is full.

To provide a more fair comparison, we compare TL2 with Shaun Tancheff's ZDM-Device-Mapper [8], the only comparable translation layer available to the authors. ZDM-Device-Mapper is similar to the DFTL [26] flash translation layer: the entire device is mapped at a 4KB granularity, with the map stored on disk and cached in memory as needed. Like DFTL it is implemented as a device mapper target. Unlike TL2/FSTL, or other device mapper targets which we are aware of, ZDM-Device-Mapper buffers the data from some write

[4]These results seem to offer a lesson concerning the naive usage of trace-driven storage system evaluations.

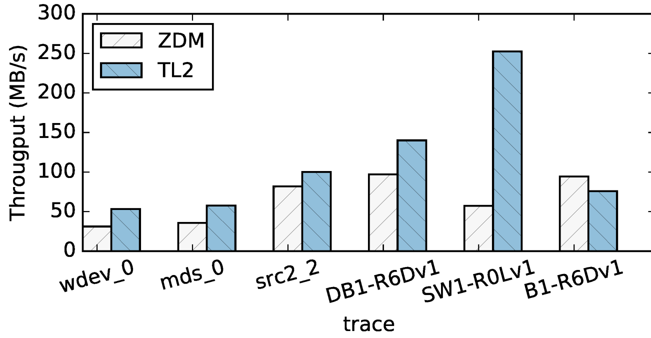


Fig. 11: Trace replay: TL2 and ZDM.

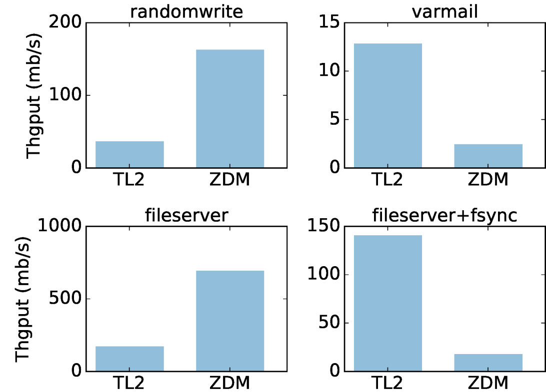


Fig. 13: Filebench performance: TL2 vs ZDM.

operations in kernel memory, signaling completion of the operation but not writing to the device until a flush command is received; this complicates several of our experiments.

Trace replay: To fit with the constraints of ZDM-Device-Mapper, I/O offsets and lengths were aligned to multiples of 4 KB. Traces were replayed against TL2 and ZDM-Device-Mapper with write cache and drive lookahead enabled on the underlying device; results may be seen in Figure 11. This represents a best-case scenario for each translation layer: operations are streamed to disk as fast as possible, with no flush operations inserted by `fio`; for one of the shorter traces `SW1_R0Lv1` the amount of buffered data on the device at the end of the test results in apparent write speeds greater than that of the disk media itself.

Under these circumstances TL2 performance is seen to be significantly faster than that for ZDM. (Since the variation in speedup is not random, but rather specific to each trace, we do not calculate a mean.) To explore the reason for the improvement, Figure 12 shows the cumulative distribution of latencies for TL2 and ZDM on a specific trace. Low-latency I/Os (i.e. up to about 50th percentile) are seen to be substantially faster for TL2, by about 10 μs. We hypothesize that the complexity of the ZDM-Device-Mapper code and its buffering model result in significant per-I/O software overhead for each operation. This hypothesis is supported by system measurements: system CPU utilization when running the ZDM test was over twice that seen when running the same test over TL2. In other words, TL2 appears to be significantly faster than ZDM on this particular test not due to large differences in what is being written to the underlying drive, but rather due to the speed of the more streamlined FSTL kernel implementation.

Filebench: Filebench experiments were performed with both device write cache and lookahead enabled; results may be seen in Figure 13. ZDM shows a significantly higher throughput compared to TL2 across all workloads except for `varmail` and `modified fileserver` (Similar to TL1/DM experiments explained in subsection V-A we added `fsync` to `fileserver` workload to neutralize the impact of buffer cache to be bale to compare the TL2 and ZDM-device-mapper fairly).

Compilebench: We were unable to use Compilebench to compare TL2 and ZDM-Device-Mapper, as in our configuration the benchmark would freeze due to stability issues with ZDM-Device-Mapper when used with ext4.

C. Cleaning algorithms

Finally, we compare three zone selection algorithms implemented in TL1 for cleaning, described previously: FIFO, which selects cache zones to be cleaned in a same order they

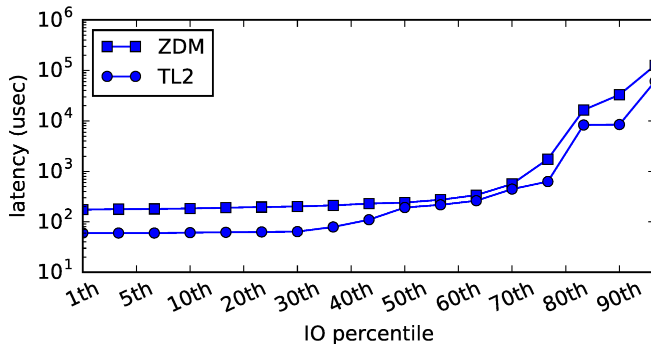


Fig. 12: Cumulative IO latency of TL2 and ZDM for wdv_0 trace.

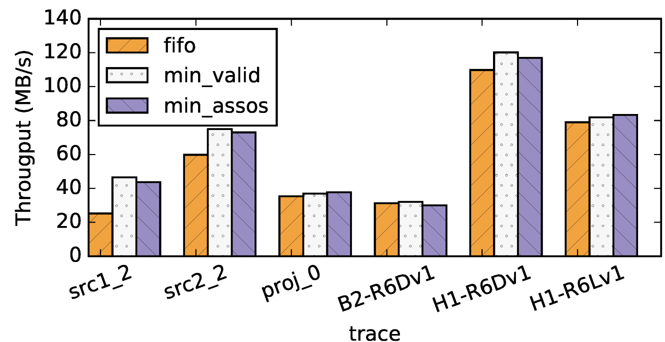
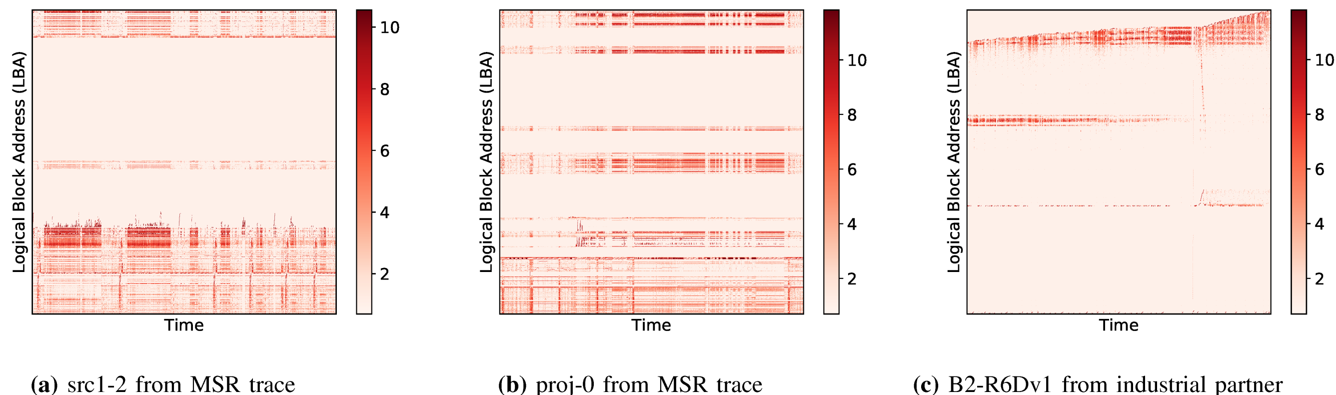


Fig. 14: TL1 performance, varied workloads, FIFO, min_valid and min_assoc cleaning.



(a) src1-2 from MSR trace

(b) proj-0 from MSR trace

(c) B2-R6Dv1 from industrial partner

Fig. 15: Heatmap of LBA accesses over time (every 32 operations) for MSR Cambridge and our industrial partner trace. Graphs represent 3D histograms of LBA accesses: LBAs are identified by time of their access (X axis), actual LBA is on Y axis, and count by color intensity. Graphs (a) and (b) count LBA accesses by MSR traces; (c) counts LBA accesses for archival pattern by Industrial partner.

have been written to, min_valid, which chooses a zone with minimum live data (also known as Greedy), and min_assoc, based on the cleaning strategy in Cho's K-Associative Sector Translation [21], selecting the cache zone with valid data spanning the minimum number of independent data zones.

Results are seen in Figure 14. FIFO cleaning is seen to under-perform by amounts ranging from tiny (`H1-R6Lv1`) to a factor of 2 (`src1_2`), while min_valid and min_assoc are essentially tied. This is despite the fact that cleaning work in TL1 is actually proportional to zone associativity rather than valid sectors; for an explanation of this we examine the access patterns of several of the traces in Figure 15. Access patterns are seen to be highly skewed, with a small range of hot LBAs, and infrequent accesses to a wide range of other addresses. In effect in these cases both min_valid and min_assoc (unlike FIFO) are effective at leaving hot data in the cache, and any further gains to be found are minimal.

VI. Related Work

Research on Shingled Magnetic Recording drives can be categorized into three following groups:

Modeling and characterization: Skylight [7] proposes a framework by which a device-managed SMR drive can be characterized using simple fio tests. These characterizations, however, are limited to underlying structures of the drive such as band size and cache location and do not cover drive's dynamics. Shafaei et. al [27], [19] address this shortcoming by proposing a modeling approach by which a drive's underlying algorithms such as cleaning algorithms can also be discovered in detail. The proposed model then is used to provide an SMR drive simulator which gets a trace as its input and estimates the latency of each IO as if the trace gets replayed on the modeled drives. Such an accurate model can be used to explore different design choices and policies. Moreover, it is useful in studying the behavior of the drive for a set of workloads by avoiding high cleaning delay between different runs skip cleaning delays while testing the drive's behavior for different workloads. Niu et. al [28] use queueing based analytical modeling to provide meaningful design insights for SMR drives.

SMR specific file systems and object stores: The basic idea behind these efforts is to either customize the existing file systems for SMR drives considering their limitations or to use the metadata information in the file system such as file creation/deletion to reduce cleaning overheads. SFS [18] divides the disk space into 64MB zones, separates sequential and random streams and forwards metadata to the first free blocks in random zones. HiSMRfs [29] buffers metadata in a tree like data structure in memory and uses an SSD for persisting them. Ext4-lazy [30] modifies journaling in ext4 to improve performance of drive-managed SMR. SMORE [31] is a log structured based object store for SMR drives which stripes data and erasure codes across zones on different disks to protect the data against disk failures. SMRDB [32] is a key-value store based on log structured merge tree which has been optimized for SMR disks.

Shingling translation layers: Much SMR research is focused on data mapping schemes and translation layers. Cassuto et. al [3] propose using well-known techniques such as set associative cache and circular buffers along with S-blocks to manage the performance degradations in device managed SMR drives. (note that both the set-associative and S-block-based translation layers from this work may be implemented with FSTL) Tan et al [2] evaluate a number of translation layers in a simulation environment and provide some rough comparisons among them in terms of their performance and space utilizations. Hall et. al [4] mitigate SMR performance degradations caused by random writes by transforming the host's random writes into sequential writes and high-queue-depth random reads. He and Du [33], [34] propose several static and dynamic mapping schemes for SMR. Their recent

work, SMART [34] proposes using track-based dynamic mapping. Shafaei et. al propose Virtual Guard [5] a track-based static translation layer in which unlike the traditional STLs the cache space is used for keeping the data at risk instead of updates. This makes the cache usage on the drive a function of write footprints rather than the number of writes and therefore avoids cleaning for all available real world traces tested. As both SMART and Virtual Guard are track-based and do not perform out-of-place writes, they cannot be implemented with the current version of FSTL. Lin et. al [35] reduce the cleaning overheads by hot and cold data segregation; Jones et. al [36] propose using the write history (frequency) of data blocks to reduce the data movements due to compaction or cleaning. Both of these translation layers may be implemented in FSTL.

VII. Conclusion

FSTL provides a flexible and high performance framework for translation layer research. Our work demonstrates that it may be used to create simple but robust translation layers which rival the performance of existing drive-managed algorithms, while supporting mechanisms to allow more powerful translation layers to be created. We have made it available under an open source license, and hope that it stimulates additional research in this area.

References

[1] S. N. Piramanayagam, "Perpendicular recording media for hard disk drives," *Journal of Applied Physics*, vol. 102, no. 1, p. 011301, Jul. 2007.

[2] S. Tan, W. Xi, Z. Ching, C. Jin, and C. Lim, "Simulation for a Shingled Magnetic Recording Disk," *IEEE Transactions on Magnetics*, vol. 49, no. 6, pp. 2677–2681, Jun. 2013.

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

[4] D. Hall, J. Marcos, and J. Coker, "Data Handling Algorithms For Autonomous Shingled Magnetic Recording HDDs," *IEEE Transactions on Magnetics*, vol. 48, no. 5, pp. 1777–1781, May 2012.

[5] M. Shafaei and P. Desnoyers, "Virtual guard: A track-based translation layer for shingled disks," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, 2017.

[6] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding Manycore Scalability of File Systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 71–85.

[7] A. Aghayev, M. Shafaei, and P. Desnoyers, "Skylight—a window on shingled disk operation," *ACM Transactions on Storage*, vol. 11, no. 4, pp. 16:1–16:28, Oct. 2015.

[8] S. Tancheff, "Seagate zdm device mapper," https://github.com/Seagate/ZDM-Device-Mapper.

[9] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[10] E. Krevat, J. Tucek, and G. R. Ganger, "Disks are like snowflakes: no two are alike," in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*. USENIX Association, 2011, pp. 14–14.

[11] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds." in *FAST*, 2011, pp. 91–103.

[12] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 327–338.

[13] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009, p. 10.

[14] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance." in *USENIX Annual Technical Conference*, vol. 8, 2008, pp. 57–70.

[15] "Advanced format technology brief," HGST, Technical Report, Mar. 2014.

[16] G. Tyndall, "Why Specify Workload?" Western Digital Technologies, Inc, Technical Report 2579-772003-A00, Jun. 2013. [Online]. Available: http://www.wdc.com/wdproducts/library/other/2579-772003.pdf

[17] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, 2002.

[18] D. Le Moal, Z. Bandic, and C. Guyot, "Shingled file system host-side management of shingled magnetic recording disks," in *Consumer Electronics (ICCE), 2012 IEEE International Conference on*. IEEE, 2012, pp. 425–426.

[19] M. Shafaei, M. H. Hajkazemi, P. Desnoyers, and A. Aghayev, "Modeling Drive-Managed SMR Performance," *ACM Trans. Storage*, vol. 13, no. 4, pp. 38:1–38:22, Dec. 2017.

[20] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *13th ACM symposium on Operating systems principles*. Pacific Grove, California, United States: ACM, 1991, pp. 1–15.

[21] H. Cho, D. Shin, and Y. I. Eom, "KAST: K-associative sector translation for NAND flash memory in real-time systems," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09.*, 2009, pp. 507–512.

[22] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: practical power management for enterprise storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. San Jose, California: USENIX Association, 2008, pp. 1–15.

[23] J. Axboe, "fio," https://github.com/axboe/fio.

[24] S. Shepler, E. Kustarz, and A. Wilson, "Filebench," San Jose, California, Feb. 2008.

[25] C. Mason, "Compilebench," http://oss.oracle.com/~mason/compilebench.

[26] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Washington, DC, USA: ACM, 2009, pp. 229–240.

[27] M. Shafaei, M. H. Hajkazemi, P. Desnoyers, and A. Aghayev, "Modeling smr drive performance," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, ser. SIGMETRICS '16. New York, NY, USA: ACM, 2016, pp. 389–390.

[28] J. Niu, J. Xu, and L. Xie, "Analytical modeling of smr drive under different workload environments," in *2017 13th IEEE International Conference on Control Automation (ICCA)*, July 2017, pp. 1113–1118.

[29] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim, "HiSMRfs: A high performance file system for shingled storage array," in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, Jun. 2014, pp. 1–6.

[30] A. Aghayev, T. Ts'o, G. Gibson, and P. Desnoyers, "Evolving ext4 for shingled disks," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 105–120.

[31] P. Macko, X. Ge, J. John Haskins, J. Kelley, D. Slik, K. A. Smith, and M. G. Smith, "Smore: A cold data object store for smr drives," in *2017 33rd Symposium on Mass Storage Systems and Technologies (MSST)*, 2017.

[32] 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. New York, NY, USA: ACM, 2015, pp. 18:1–18:11.

[33] W. He and D. H. Du, "Novel address mappings for shingled write disks," in *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*. USENIX Association, 2014, pp. 5–5.

[34] ——, "Smart: An approach to shingled magnetic recording translation," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 121–134. [Online].

Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/he
[35] C.-I. Lin, D. Park, W. He, and D. Du, "H-SWD: Incorporating Hot Data Identification into Shingled Write Disks," in *2012 IEEE 20th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2012, pp. 321–330.
[36] S. N. Jones, A. Amer, E. L. Miller, D. D. Long, R. Pitchumani, and C. R. Strong, "Classifying data to reduce long term data movement in shingled write disks," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–9.