ZTL: Enabling Zoned Namespace Support for File Systems

Jan Sass[1][0009-0006-3132-4544], André Brinkmann[1][0000-0003-3083-2775], Xubin He[2][0000-0002-5071-2861], Matias Bjørling[3][0000-0003-1657-4052], and Reza Salkhordeh[1][0000-0003-3786-7102]

[1] Johannes Gutenberg University, Mainz, Germany
{sass,brinkman,salkordeh}@uni-mainz.de
[2] Temple University, Philadelphia, Pennsylvania, U.S.A.
xubin.he@temple.edu
[3] Western Digital Corporation, Copenhagen, Denmark
matias.bjorling@wdc.com

Abstract. A Flash Translation Layer (FTL) hides the intrinsic flash properties of SSDs and provides a block interface to the host. FTLs completely embedded in the firmware of SSDs must therefore duplicate host functionality, such as the translation between logical and physical blocks performed by a file system, introducing performance unpredictability and increasing device cost. Zoned Namespace (ZNS) SSDs overcome these drawbacks by only implementing limited FTL functionality inside the SSD and by exposing a more flash-friendly interface to the host. However, moving away from the block interface means that the storage stack inside the host must be modified for ZNS devices. This requires considerable effort, which is one reason why F2FS is the only Linux file system with reliable ZNS support today.
This paper discusses how the Linux storage stack can be extended to simplify the process of adapting file systems to ZNS devices. It then proposes the host-side Zoned Translation Layer (ZTL), which provides abstractions and functionalities required by many file systems to support ZNS devices. We demonstrate the feasibility of ZTL by providing the first EXT4 implementation for ZNS devices and by comparing our implementation of ZNS support for F2FS with the native ZNS support of F2FS.

1 Introduction

Compared to hard disk drives (HDDs), SSDs offer increased throughput and reduced latency for read and write operations [6, 8, 13]. Read and write accesses are performed at the granularity of pages, which are grouped into blocks. Each block must be written sequentially. SSDs do not allow in-place updates and erase operations must be performed at block granularity. A dedicated firmware component, the Flash Translation Layer (FTL), hides these flash-specific characteristics and provides a block interface that can be accessed in the same way as HDDs [6, 13, 31]. The main functions of the FTL include mapping between

logical block numbers requested by the host and physical block numbers on the SSD, garbage collection to reclaim invalidated pages, and wear-leveling to reduce the impact of erase operations on the SSD's limited lifetime.

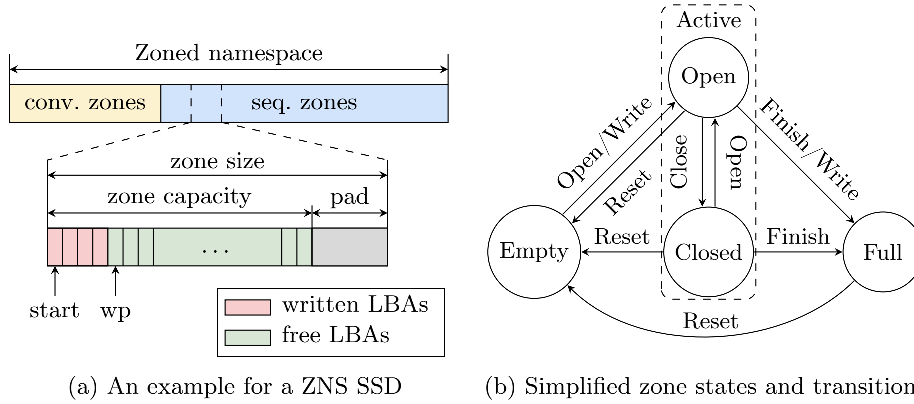
FTLs introduce their own overhead by requiring additional device resources and they operate without semantic information about the pages, resulting in suboptimal performance and increased costs [2, 32]. For example, their logical-to-physical page mapping introduces an additional layer of indirection when coupled with a file system, as the file system already maps its contents to logical blocks, which are then again mapped by the FTL to physical pages on the SSD. This page mapping consumes a significant amount of the scarce memory resources on SSDs. In addition, the block interface cannot semantically distinguish between data and metadata of a file system and also cannot infer the hotness of the data, resulting in increased write amplification [22, 23].

The Zoned Namespace (ZNS) specification defines a novel way for the host to directly interface with the flash medium, removing the need for a full on-device FTL. Compared to conventional SSDs, ZNS SSDs allow for reduced cost, more consistent performance, and prolonged device lifetime [2, 23]. The integration of ZNS SSDs into the storage stack requires moving parts of the FTL into the host. Generic host-side FTL implementations, such as dm-zoned [27] or dm-zap [18], are reimplementations of SSD FTLs and do not utilize the benefits of ZNS. On the other hand, FTLs included in file systems, such as F2FS [12] and BTRFS [24, 25], lack portability and spread out development over multiple implementations.

We propose the *Zoned Translation Layer* (ZTL), a host-side flash translation layer that abstracts generic parts of the FTL, such as garbage collection and ZNS management, and provides an interface that allows file systems to control data placement on the device and garbage collection scheduling. ZTL requires no additional translation layer as it integrates tightly with the file system and significantly reduces the effort required to adapt general-purpose file systems to ZNS devices compared to dedicated FTL implementations. We have integrated ZTL with EXT4 and F2FS and evaluated our implementation on physical hardware.

2 Technical Background

Flash pages are grouped into blocks, which need to be written sequentially. Once a block has been fully written, it can only be rewritten after erasing it. The FTL transforms host write operations into sequential write patterns to satisfy these requirements. It operates transparently and employs a mapping to translate the logical block address (LBA) specified by the host into its physical counterpart [6, 8, 31]. A page on the device is considered invalid when it is rewritten to a new location or denoted as deleted by the host. In order to reclaim invalid pages, the FTL executes a periodic garbage collection. During this task, valid pages of a selected *victim* block are rewritten to different blocks and then the victim block is erased. The lack of semantic information on the device leads to suboptimal data placement and increased garbage collection overheads [9, 14]. Combined with



(a) An example for a ZNS SSD (b) Simplified zone states and transitions

the duplicated mapping efforts, these shortcomings invite alternative concepts in order to replace the FTL [2, 10, 15].

ZNS SSDs were introduced to overcome the limitations of standard SSDs by moving most of the FTL to the host [3]. According to the NVMe ZNS specification [21], the LBA range of a ZNS SSD is partitioned into zones of equal size. Each zone must be written sequentially and can only be erased as a whole. Each zone has a write pointer (wp), denoting the next sector to be written. To align the starting sector of every zone to a power-of-two value, address padding is employed. ZNS SSDs require a host-side driver similar to the FTL on conventional SSDs: The host has to align writes with the write pointer of a zone, perform garbage collection to reclaim invalidated pages, maintain metadata for validity information and mapping tables, and handle zone state changes.

Aligning write operations with a write pointer requires the host to enforce strict request ordering throughout its storage stack, reducing flexibility and potentially throughput while increasing implementation complexity [5, 16]. The *zone append* operation was added, so that the host can simply append data pages to a selected zone. The device then advances the writer pointer accordingly and returns the physical location in a callback to the host [3, 7, 21].

Each sequential write zone represents a state machine (Figure 1b). On an empty device, each zone is in the *Empty* state and its write pointer points to its start sector. Issuing write operations transitions zones implicitly from *Empty* to *Open* and ultimately to *Full*. *Full* zones may be rewritten after an explicit *Reset* operation. While most transitions are performed implicitly by the device, the host can issue explicit transitions to enforce a stricter state policy. The number of concurrently active zones is limited due to device resources [3, 11, 21].

3 Design and Implementation

ZTL is a kernel space FTL implementation that integrates with file systems, enabling them to utilize the benefits of the ZNS. We therefore assume that a random-write area is available for metadata operations of the file system. This

area can either consist of conventional zones on the ZNS or dedicated storage. We see this as a sensible assumption as writing metadata to append-only logs incurs excessive overheads [12]. F2FS, the only stable file system with ZNS support, follows the same approach [26].

3.1 ZNS Support for File Systems

ZTL transparently implements all file system-independent tasks in the block layer. This includes managing zones, transforming write requests into sequential patterns, and providing a garbage collection routine.

Append-only logs and zone representation. ZTL abstracts physical zones using append-only logs. The file system can configure the number of logs to match its write patterns, e.g., to separate data with different expected lifetimes. Each log points to an active zone, which ZTL advances automatically once the zone becomes full. Towards the file system, a log is represented by a small integer value, e.g., $l \in \{0, 1, \ldots, n-1\}$ where n is the number of logs.

ZTL obtains zone information (LBA range of the sequential area and zone states) from the device at setup time. Additional metadata is provided by the file system, i.e., validity bitmaps used by the garbage collection, log-to-zone mappings from previous mount times, and each log's migration target.

Transparent transformation of write requests. ZTL attaches to the ZNS block device and integrates with functions inside the block layer that are used by file systems when writing to a device. Upon request creation, the request is limited to the maximum number of pages for a zone append operation. When adding the first page to the request, ZTL redirects the request to the zone currently represented by the selected log. When further pages are added, ZTL clamps the request to zone boundaries. During the request callback after a successful write operation, ZTL state is updated.

The behavior of these functions is fully transparent to the file system, which only needs to configure the request to perform the zone append operation and supply the log number in place of an LBA. ZTL retains the idiomatic workflow of adding pages to a request until either the request is full or, specific to ZNS, a zone boundary is encountered. File systems already align with this routine by submitting the current request and starting a new one. Hence, the submission workflow requires minimal modifications within the file system.

Garbage collection. ZTL tracks page invalidations in per-zone counters and bitmaps. A background thread periodically checks the page counters of full zones and starts a garbage collection if a zone contains a large fraction (e.g., 75 %) of invalid pages. In addition, ZTL can perform a foreground garbage collection when under pressure, i.e., when less than 5 % of the zones are empty during log advancements, or when the file system chooses to invoke it.

During garbage collection, the remaining valid pages of a zone are read from the device and written to a different zone using the zone append operation. Afterwards, ZTL triggers an update of the file system metadata and invalidates the old page locations on the victim zone of the garbage collection. Once all

pages have been migrated and their old locations have been invalidated, the victim zone can be reset.

The file system may specify the log target of page migrations in order to prevent repeated page migrations. For example, a file system that uses logs $\{0, 1, 2\}$ to represent cold data, hot data, and directories, may specify a mapping such as $\{0 \rightarrow 0, 1 \rightarrow 0, 2 \rightarrow 2\}$ to indicate migration targets. To ensure garbage collection can always succeed, a small number of zones (depending on the log configuration) are reserved for garbage collection purposes.

3.2 Enabling ZNS benefits for file systems

ZTL allows the file system to schedule a garbage collection and provide its own implementation of victim selection strategies. This allows the file system to leverage the advantages of ZNS and reduce write amplification as well as the impact of garbage collection on read and write latencies. By selecting a ZTL log to append to, the file system can attain the physical separation of data with different expected lifetimes, further augmenting this benefits.

3.3 Eliminate Duplicate Translation

The file system maintains a mapping from logical blocks belonging to files or directories to blocks on the device. Traditional FTL implementations, being detached from the host system, perform an additional translation from logical block addresses to flash pages. This extra layer of indirection can be removed by using the integration of ZTL with file systems as no additional translation is required by ZTL. Furthermore, ZTL utilizes the NVMe metadata features [20] to place the physical-to-logical mapping on the device, which is required by the garbage collection to update the file system metadata after page migrations.

3.4 Sustain File System Consistency

A ZTL host must delay metadata updates until a write operation completes and its callback is triggered (Section 4.2). If a system failure occurs *after* the write completes and *before* the file system updates its metadata, the pages are already written to the ZNS device while the file system has no information about their location. However, this does not harm consistency, as pages are only marked up-to-date after the metadata update of the file system. The impact of a fault during a zone append operation is the same as with conventional writes. Fault handling at any other time during the lifetime of the file system remains unchanged.

4 File System Integration

Some functionality necessarily remains dependent on the file system and cannot be abstracted by ZTL. In this section, we will describe the required adaptations. Listing 1.1 displays parts of the interface exposed by ZTL.

```
struct ztl_operations {
    int (*gc_bio_begin)(struct block_device *bdev, struct bio *bio,
                unsigned int from_log);
    int (*gc_bio_end)(struct block_device *bdev, struct bio *bio,
                unsigned int to_log);
    int (*fg_find_victim)(struct block_device *bdev, size_t nr_zones,
                struct ztl_zone **zones, struct ztl_zone **victim);
    int (*bg_find_victim)(struct block_device *bdev, size_t nr_zones,
                struct ztl_zone **zones, struct ztl_zone **victim);
    bool (*bg_gc_may_run)(struct block_device *bdev);
};

struct ztl_config {
    log_t nr_logs;              /* Nr of logical logs */
    sector_t *log_zones;        /* Each log's current zone */
    log_t *log_migration_zones; /* Map of logs to GC target */
    log_t min_op_zones;         /* Minimum OP zones */
    unsigned int gc_inteval;    /* Interval for background GC, in secs */
    bitmap_t **blk_validity;    /* Per-zone validity bitmaps */
};
int ztl_init_context(struct super_block *sb, struct block_device *bdev,
                     struct ztl_operations *ops, struct ztl_config *cfg);
void ztl_destroy_context(struct block_device *bdev);
int ztl_run_gc(struct block_device *bdev);
int ztl_invalidate(struct block_device *bdev, ino_t ino, ztl_blk_t block,
                   ztl_blk_t count);
```

Listing 1.1: Excerpt of ZTL interface structures, prototypes, and functions. Only the prototype `gc_bio_end` is mandatory.

4.1 Context Lifetime and Super Block Changes

A ZTL context is required for each ZNS used by a file system. Accordingly, ZTL should be set up and removed by the file system when mounting and unmounting a ZNS device (`ztl_init_context` and `ztl_destroy_context`). On initialization, function prototypes and configuration options can be specified by the file system (Listing 1.1). The configuration allows the file system to specify the number of write logs (`nr_logs`), their underlying zones (`log_zones`), and their corresponding garbage collection targets (`log_migration_zones`). If `log_zones` is not provided, ZTL will autonomously select suitable zones. The overprovisioned zones (`min_op_zones`), and the garbage collection interval (`gc_interval`) can additionally be configured. Lastly, zone validity bitmaps from a previous mount time may be supplied (`blk_validity`). ZTL maintains no persistent metadata itself as it does not own any block device itself. Hence, the file system needs to hold ZTL related metadata in order to allow remounts (e.g., block validity information, requried for garbage collection).

Additionally, the file system must keep information about the ZNS and its logical block range to maintain consistency. For example, the total available blocks are required to account for the capacity of the device. Lastly, the user space tools that create file system partitions (`mkfs`) may be needed to be modified to consider ZTL integration. However, this is out of the scope of this work.

4.2 Zone Append Operation

The file system determines which write operations are designated for the random-write area, i.e., metadata or data that relies on a fixed block location. Typically, this data can be identified by examining the inode type and number to distinguish between file data, directories, or special inodes (e.g., containing metadata or journals). All other write operations are designated for the sequential-write area of the ZNS. Some file systems, such as F2FS, already categorize pages into metadata, nodes, and data.

When issuing write operations to the conventional area, no or very little changes to the write workflow are required. For example, some file systems place metadata relative to the data on the device and may need to adjust the target sector for metadata writes accordingly. Despite our efforts to minimize the intrusiveness of ZTL, creation and submission of write operations to the sequential area require minor modifications. First, on request creation, the file system is to set the operation flag to zone append and select the log to append to instead of providing an LBA. Second, the file system needs to update its metadata after the operation, as the physical location of the written pages is provided during the request callback. Lastly, some global file system tasks, such as checkpointing or flushing of journals, may need to wait for the writeback operation on pages to finish in order to prevent deadlocks.

4.3 Block Validity and Garbage Collection

Garbage collection migrates valid pages to a different zone. This task is performed by ZTL, which afterwards notifies the file system about the new physical locations of migrated pages. To this end, the file systems must provide an implementation for the `gc_bio_end` prototype. ZTL allows the file system to perform additional preparations by providing a definition for the `gc_bio_begin` prototype. For the file system, the metadata update after a migration is very similar to the update after a non-migration write (Section 4.2), allowing the file system to aggregate both update functions.

ZTL holds information about the validity of pages in each zone for use by the garbage collection. The file system must call the `ztl_invalidate` function to notify ZTL whenever a physical address is no longer used by the file system, e.g., when deleting a file. Otherwise, the garbage collection would migrate pages the file system considers as deleted and trigger a file system metadata update based on outdated information.

The file system can provide implementations for the remaining function prototypes in Listing 1.1. `fg_find_victim` and `bg_find_victim` are used to select

a zone during foreground and background garbage collections. If no implementation is given, ZTL uses a variant of the greedy victim selection (Section 3.1). In order to postpone the garbage collection dynamically, the file system can use the prototype `bg_gc_may_run` which is called by ZTL at the start of every garbage collection task. The file system can use `ztl_run_gc` to manually trigger a garbage collection.

5 Case studies

In this section, we describe the integration of ZTL into EXT4 and F2FS. EXT4 does not natively support ZNS. F2FS, on the other hand, already supports ZNS SSDs, which allows us to compare the effort of our implementation against the native approach.

5.1 EXT4

EXT4 utilizes extents for allocating data on storage devices. Extents only require the starting LBA, the corresponding file offset and the number of consecutive blocks, effectively reducing the metadata overhead and allocation time compared to mapping single blocks [19, 30].

During write operations, EXT4 performs four steps. First, for a given page, it traverses its inode's extent tree to look for a previously mapped physical location. If a previous block number is found, allocation stops. Otherwise, a new physical location is selected. The in-memory metadata is then updated for future requests. Second, the page is added to a write request, which is created if it does not already exist. Third, if no more pages are to be written or subsequent pages are not adjacent in the inode's address space, the write request is submitted. Finally, a callback step commits the in-memory metadata to the journal and marks the written pages as up-to-date. EXT4 always only writes pages that belong to a single inode, and all pages in a write request are contiguous in the inode's address space.

We implemented all changes according to Section 4. First, we integrated the ZNS device into the EXT4 metadata and added its zones to the available block range of the current EXT4 mount.On context creation, EXT4 registers ZTL with three logs: one for directories, one for data, and a garbage collection target. This extends the standard EXT4 implementation by an approach to reduce the garbage collection impact, based on previous works [12, 15]. We provided an implementation for `gc_bio_end` and left the remaining ZTL settings to the default.

Second, to align with the zone append operation, we modified the write routines and directed data to the sequential part of the ZNS. We set the request operation flag accordingly and selected the appropriate log for the pages to be written. To decide which write operations are designated for the sequential area of the zoned device, we used the inode number of the pages to be written and

directed metadata and journal pages to the conventional area. For other pages, we inspected the inode type to distinguish file data from directories.

Third, we integrated metadata updates into the after-write work queue of EXT4, which also frees resources and sets pages up-to-date. Lastly, we added calls to inform ZTL of invalid LBAs when an extent is deleted during removal or overwrite operations.

5.2 F2FS

The Flash Friendly File System (F2FS) [12] is designed to align file system tasks with the FTL as closely as possible and to extend the lifetime of flash storage while improving performance over previous file systems. F2FS appends write operations to different logs, based on the type of data and its expected lifetime. A random-write area is used to store metadata. F2FS requires a garbage collection routine similar to that of the FTL. The unit of garbage collection, called a *segment*, is aligned to the physical blocks of the underlying SSDs. By additionally issuing the *discard* operation to the device, F2FS aims to perform the garbage collection at file system level and leverage host insights to decrease the impact of the FTL. Because of its log-structured design and the use of write-buffers, F2FS does not support direct I/O.

F2FS uses inode indirection blocks to denote the location of data on the device. Inodes and indirection blocks are written to dedicated logs. F2FS uses a node address table to store the location of node blocks in the log. In order to avoid recursive updates, this table is kept in the random-write area next to other metadata [4, 12].

When a ZNS device is detected, F2FS adapts the garbage collection unit to fit the size of the zones of the ZNS SSD, replaces the *discard* operation with a zone reset command, and disables in-place updates which are used when the available device capacity is below a given threshold [11]. Additionally, F2FS handles zone states, acquires zone metadata, and translates the device geometry into internal data structures.

The default ZNS support in F2FS does not utilize the zone append operation. Instead, it relies on ordered scheduling throughout the storage stack on the host. Hence, F2FS requires the `mq-deadline` scheduler to be enabled for the zoned device. `mq-deadline` has previously been adapted to submit write requests to zoned devices in ascending order by their target sector [5, 7, 11]. Support for zone write plugging [16] is currently not implemented in F2FS.

We integrated ZTL within F2FS by setting up the FTL context and related data structures. We configure ZTL logs similar to the native ZNS support and adapted the write workflow to make use of the zone append operation. As data block updates need to be denoted in node blocks, node updates need to be delayed until the corresponding data blocks are written. We handle the request callbacks for node and data separately using dedicated work queues. Lastly, we disabled in-place updates and the F2FS-intrinsic garbage collection and, instead, use the ZTL GC implementation.

F2FS performs periodic checkpointing to maintain file system consistency when a system outage occurs. While doing this, all other writeback operations are temporarily paused. The checkpoint procedure flushes the write buffers to the backing device and causes the associated callbacks to take place. A checkpoint is then written to the metadata region of the device. When a checkpoint is started before a write request is mapped, up-to-date node information is not available and the file system is at risk of being corrupted. Hence, we delayed checkpointing while write operations take place.

In contrast to the strict write ordering that F2FS applies for native ZNS support, our approach uses the zone append operation and removes the previously imposed scheduling restriction.

6 Experimental results

In this section, we evaluate the impact of ZTL integration in terms of file system throughput, latencies and potential processing overheads. We compare ZTL integrated into EXT4 and F2FS (refered to as EXT4-ZTL and F2FS-ZTL) with stock F2FS and BTRFS on a physical ZNS SSD. While we included BTRFS in our evaluation, we note that tests run with fio [1] produced device errors during execution (i.e., trying to exceed the limit of concurrently opened zones). Despite numerous errors, BTRFS maintained its consistency. We did not compare against host-side FTLs such as dm-zap, dm-zoned, or SPDK as they pursue different goals.

We run all experiments on a machine with a 20 core Intel Xeon Gold 5215 CPU at 2.50 GHz and 128 GiB memory. We used an 8 TB Western Digital Ultrastar DC ZN540 as a physical ZNS SSD. The conventional area of this device is not sufficient for file system metadata and we therefore used a Samsung 983 DCT Series Enterprise SSD for the conventional area. We expect that future generations of ZNS devices come with a larger conventional area and more flexible device partitioning.

The zones of the ZNS are 2,048 MiB in size and 1,077 MiB in capacity. F2FS, F2FS-ZTL and EXT4-ZTL used the conventional SSD for metadata while BTRFS ran exclusively on the ZNS device. We used Linux kernel version 6.6 for all tests. Garbage collection was enabled during all experiments. F2FS and BTRFS employ their own implementation while F2FS-ZTL and EXT-ZTL utilize the implementation described in Section 3.1. Both variants perform the same steps, i.e., reading and writing pages from the ZNS and updating page locations in the file system metadata. Further investigation of garbage collection is out of the scope of this work.

6.1 Microbenchmarks

We conducted microbenchmarks using fio with different request sizes, measuring throughput, latency and CPU load during file system workloads. We tested read and write in random or sequential order (*read-rand*, *read-seq*, *write-rand* and

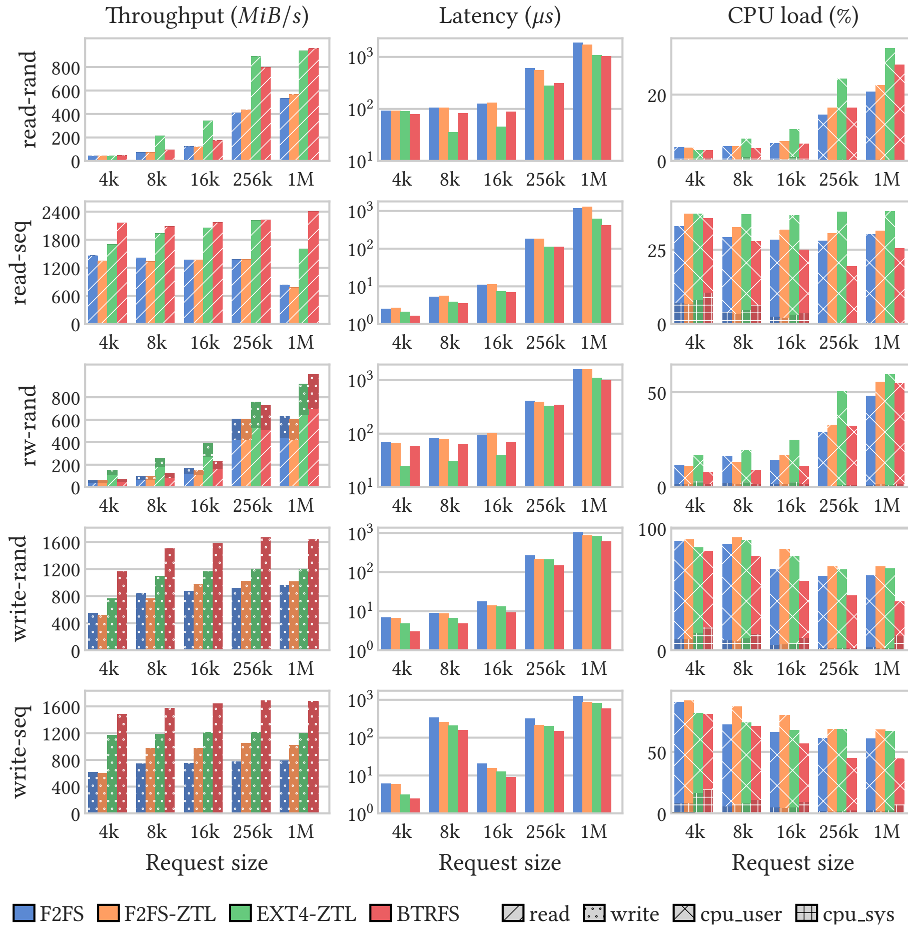


Fig. 2: Microbenchmarks of throughput, average latencies and CPU load (a load of 100 % equals one fully-utilized CPU core). CPU load is divided into user space load (cpu_user) and kernel space load (cpu_sys).

write-seq), and a mixed workload of 30 % random writes and 70 % random reads (*rw-rand*). The results are presented in Figure 2.

We observed significant performance improvements on write operations when using F2FS-ZTL compared to the native F2FS implementation. The latter employs conventional writes, requiring additional scheduling. By using the zone append operation, F2FS-ZTL circumvents scheduling restrictions and exceeds the write throughput of native F2FS by up to 37 % for sequential write workloads, while maintaining compatible or better performance for all other benchmarks.

EXT4 achieves higher throughput than both native F2FS and F2FS-ZTL in all workloads and with all request sizes, due to its more compact metadata layout. BTRFS achieves the highest throughput of all file systems as it maintains

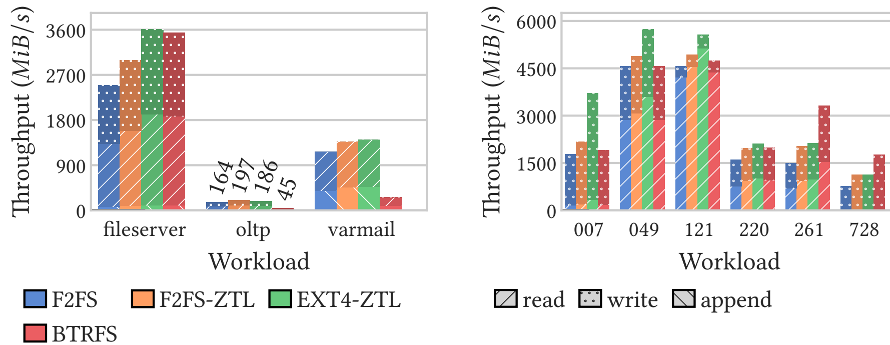
the largest number of concurrently opened zones, more effectively saturating the device bandwidth than other file systems. Similar to EXT4, BTRFS also maintains its metadata in extent trees for better cache utilization.

The average latencies measured during the tests show that the zone append workflow and the prolonged writeback period of pages do not incur additional overheads. Instead, due to increased throughput in some cases, the latencies are reduced. During the read-seq workload, all file systems perform readahead operations that lead to average latency values below the access time of the physical device for small request sizes.

CPU usage scales with number of requests processed and hence correlates with increased throughput of file systems. However, we observe that the CPU load increases for F2FS-ZTL even when the performance does not increase compared to native F2FS. This is an expected result of the zone append write workflow and the metadata update after zone append. Optimizing a file system for zone append and towards a lower CPU load is out of the scope of this work.

6.2 Filebench

We used the fileserver, oltp, and varmail personalities of the Filebench file system benchmarking framework [28]. The results of the experiments are shown in Figure 3a. We observe that F2FS-ZTL yields improvements over native F2FS of 19.8 % for fileserver, 20.3 % for oltp, and 16.9 % for varmail. EXT4 performs similar to F2FS-ZTL for oltp and varmail and better (20.8 % compared to F2FS-ZTL) for fileserver. BTRFS fails to maintain competitive throughput for oltp and varmail as frequent fsync calls lead to large metadata overheads and the resulting tree traversals slow the file system down. Additionally, BTRFS does not use a conventional device for metadata, further amplifying the effect of fsync.



(a) Filebench workloads.

(b) Selection of Alibaba block traces [17].

6.3 Block Traces

To evaluate the overall file system performance with ZTL integration, we ran a selection of *Alibaba Block Traces* [17, 29]. Block traces are evaluated using fio on a single file, representing the target block device. This evaluation reflects virtual machines workloads. We deliberately select traces with differing working set size (WSS) ratios to evaluate ZTL regarding caching and the writeback period. Our selection is displayed in Table 1. The results of these tests are shown in Figure 3b.

#	**Device Size**	**WSS Ratio**	**R:W Ratio**	**I/O Size**
007	500 GiB	5.77 %	5:95	2,328 GiB
049	40 GiB	11.33 %	51:49	1,226 GiB
121	60 GiB	15.28 %	78:22	3,396 GiB
220	160 GiB	66.12 %	69:31	1,212 GiB
261	100 GiB	93.64 %	28:72	1,744 GiB
728	300 GiB	93.37 %	4:96	1,190 GiB

Table 1: Selected Alibaba Traces [29]. The WSS (Working Set Size) ratio denotes the relative amount of pages used from the device size.

Traces with small WSS ratio show large throughput due to caching effects. With larger WSS, the device is accessed more frequently, resulting in less throughput. We notice that F2FS-ZTL shows better overall performance than native F2FS. EXT4 allows better utilization of the page cache compared to F2FS and its performance is better during workloads with a small WSS. During write-heavy workloads, BTRFS shows higher throughput than all other file systems tested, as this allows BTRFS to better utilize the bandwidth of the ZNS.

6.4 Integration Effort

We integrated ZTL into F2FS and EXT4 and changed 547 and 534 lines of code during the process. The native ZNS support for F2FS accumulates to 648 lines changed, however using traditional writes. To adapt to the zone append paradigm, more extensive modifications would have to be made. Native ZNS support for BTRFS required more than 2,500 additional lines of code, despite being currently unstable.The implementation of ZTL contains 4,000 lines of code.

While these numbers only give an estimate of the implementation effort, we note that even a traditional file system like EXT4 can utilize the advantages offered by ZTL and ZNS. Furthermore, only ZTL needs to be adapted when the kernel API or the ZNS specification evolve, not the file system itself.

7 Conclusions

In this paper, we have proposed ZTL, a compatibility layer that allows arbitrary file systems to implement ZNS support with minimal effort. Our experiments

show that ZTL reduces the effort required to add ZNS support while providing in the case of F2FS even better performance than the native ZNS implementation. Additionally, we provided the first ZNS implementation for EXT4, which exceeds the performance of F2FS.

Acknowledgement

This research has been funded by the Federal Ministry of Education and Research within the project "ScalNEXT: Optimierung des Datenmanagements und des Kontrollflusses von Rechenknoten für Supercomputing" (16ME0688).

References

[1] Jens Axboe. *fio - Flexible I/O tester* rev. 3.36. URL: https://fio.readthedocs.io/en/latest/fio_doc.html.

[2] Matias Bjørling et al. "The Necessary Death of the Block Device Interface". In: *Sixth Biennial Conference on Innovative Data Systems Research (CIDR), January 6-9*. 2013.

[3] Matias Bjørling et al. "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs". In: *2021 USENIX ATC, July 14-16*. 2021.

[4] Neil Brown. *An f2fs teardown*. Oct. 2012. URL: https://lwn.net/Articles/518988/.

[5] Jonathan Corbet. *4.16 Merge window part 1*. Feb. 2018. URL: https://lwn.net/Articles/746129/.

[6] Michael Cornwell. "Anatomy of a solid-state drive". In: *Commun. ACM* 55.12 (2012).

[7] Jake Edge. *Zoned storage and filesystems*. May 2023. URL: https://lwn.net/Articles/932748/.

[8] Eran Gal and Sivan Toledo. "Algorithms and data structures for flash memories". In: *ACM Comput. Surv.* 37.2 (2005).

[9] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings". In: *Proceedings of ASPLOS 2009, March 7-11*.

[10] Jesung Kim et al. "A space-efficient flash translation layer for CompactFlash systems". In: *IEEE Trans. Consumer Electron.* 48.2 (2002).

[11] Damien Le Moal et al. *Zoned Storage*. URL: https://zonedstorage.io.

[12] Changman Lee et al. "F2FS: A New File System for Flash Storage". In: *Proceedings of the 13th USENIX FAST Conference, February 16-19*. 2015.

[13] Sungjin Lee et al. "Application-Managed Flash". In: *14th USENIX FAST Conference, February 22-25*. 2016.

[14] Sungjin Lee et al. "Exploiting Sequential and Temporal Localities to Improve Performance of NAND Flash-Based SSDs". In: *ACM Trans. Storage* 12.3 (2016).

[15] Sungjin Lee et al. "LAST: locality-aware sector translation for NAND flash memory-based storage systems". In: *ACM SIGOPS* 42.6 (2008).

[16] Damien LeMoal. *Zone write plugging.* Mar. 2024. URL: https://lwn.net/ml/linux-block/20240328004409.594888-1-dlemoal@kernel.org/.

[17] Jinhong Li et al. "An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production". In: *IEEE International Symposium on Workload Characterization, October 27-30.* 2020, pp. 37–47.

[18] Dennis Maisenbacher et al. *dm-zap.* Aug. 2022. URL: https://github.com/westerndigitalcorporation/dm-zap.

[19] Avantika Mathur et al. "The new ext4 filesystem: current status and future plans". In: *Proceedings of the Ottawa Linux Symposium, June 27 - 30.* 2007.

[20] *NVM Express® Base Specification* v2.1. Aug. 2024. URL: https://nvmexpress.org/specifications/.

[21] *NVM Express® Zoned Namespace Command Set Specification* v1.2. Aug. 2024. URL: https://nvmexpress.org/specifications/.

[22] Seonggyun Oh et al. "MIDAS: Minimizing Write Amplification in Log-Structured Systems through Adaptive Group Number and Size Configuration". In: *22nd USENIX FAST Conf., February 27-29.* 2024.

[23] Devashish R. Purandare et al. "Append is Near: Log-based Data Management on ZNS SSDs". In: *12th Conference on Innovative Data Systems Research (CIDR), January 9-12.* 2022.

[24] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-Tree Filesystem". In: *ACM Trans. Storage* 9.3 (2013).

[25] Marta Rybczyńska. *Btrfs on zoned block devices.* Apr. 2021. URL: https://lwn.net/Articles/853308.

[26] Dongjoo Seo et al. "Is Garbage Collection Overhead Gone? Case study of F2FS on ZNS SSDs". In: *Proceedings of the 15th ACM/USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 9 July.* 2023, pp. 102–108.

[27] Mike Snitzer and Hannes Reinecke. *dm-zoned.* Feb. 2023. URL: https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-zoned.html.

[28] Vasily Tarasov, Erez Zadok, and Spencer Shepler. "Filebench: A Flexible Framework for File System Benchmarking". In: *login Usenix Mag.* 41.1 (2016). URL: https://www.usenix.org/publications/login/spring2016/tarasov.

[29] Qiuping Wang and Chao Shi. *Alibaba Block Traces.* Feb. 2022. URL: https://github.com/alibaba/block-traces.

[30] Derrick J. Wong et al. *ext4 General Information.* Apr. 2023. URL: https://www.kernel.org/doc/html/latest/admin-guide/ext4.html.

[31] Michael Wu and Willy Zwaenepoel. "eNVy: A Non-Volatile, Main Memory Storage System". In: *Sixth ASPLOS conference, October 4-7.* 1994.

[32] Jingpei Yang et al. "Don't Stack Your Log On My Log". In: *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW), October 5.* 2014.