RAIL: Predictable, Low Tail Latency for NVMe Flash

HEINER LITZ, University of California
JAVIER GONZALEZ, Samsung
ANA KLIMOVIC, ETH Zürich
CHRISTOS KOZYRAKIS, Stanford University

Flash-based storage is replacing disk for an increasing number of data center applications, providing orders of magnitude higher throughput and lower average latency. However, applications also require predictable storage latency. Existing Flash devices fail to provide low tail read latency in the presence of write operations. We propose two novel techniques to address SSD read tail latency, including *Redundant Array of Independent LUNs* (RAIL) which avoids serialization of reads behind user writes as well as *latency-aware hot-cold separation* (HC) which improves write throughput while maintaining low tail latency. RAIL leverages the internal parallelism of modern Flash devices and allocates data and parity pages to avoid reads getting stuck behind writes. We implement RAIL in the Linux Kernel as part of the LightNVM Flash translation layer and show that it can reduce read tail latency by 7× at the 99.99th percentile, while reducing relative bandwidth by only 33%.

CCS Concepts: • **Information systems** → **Flash memory**

Additional Key Words and Phrases: SSD, OpenChannel, LightNVM, Linux

1 INTRODUCTION

Flash-based storage devices are replacing disks for an increasing number of applications in data centers. Transistor scaling, multi-level cell technology, and 3D integration have delivered a continuous increase in capacity, while new Flash controllers have leveraged high degrees of architectural parallelism and new software interfaces such as NVMe to significantly increase performance [6, 55]. As a result, Flash devices now provide up to one million **I/O operations per second** (**IOPS**) and read latencies as low as 70μs [39, 64]. However, data center applications also require *predictable* performance. Low read tail latency is particularly important for high fan out applications that access thousands of servers to process a single request [22]. Existing Flash devices fail
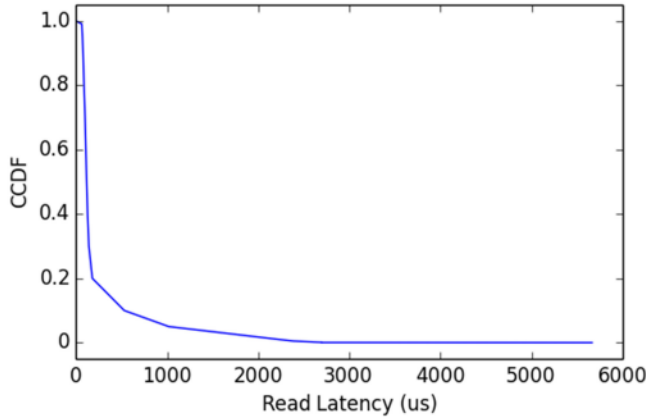
Fig. 1. Read latency CCDF for read/write mix.

to provide predictable tail read latency [30, 42]. Figure 1 shows the read latency **complimentary cumulative distribution function (CCDF)** for a mixed read/write workload on an NVMe based **solid state disk (SSD)** from CNEX Labs.[1] While most of the reads complete in under $100\mu s$ there is also a long tail with some reads completing at just under 6ms resulting in over 50× difference between the average and 99.99th percentile latency.

Prior work has focused on reducing high tail latency by optimizing **garbage collection (GC)**[67, 78, 80], interference between applications[28, 35] or by over-provisioning Flash capacity by up to 30–50% [56]. While these techniques improve performance to some degree, they fail to enforce strict tail latency guarantees and, furthermore, introduce significant overheads in terms of capacity and bandwidth. For instance, TinyTail [80] leverages a RAID [58] approach to avoid interference between reads and GC, addressing the 6ms tail shown in Figure 1. However, TinyTail does not protect against the common **read-after-write (RaW)** serialization, increasing tail latency by up to 20×. Furthermore, TinyTail introduces significant write overheads to achieve isolation between reads and GC.

To address these challenges, we present *Redundant Array of Independent LUNs* (RAIL), an SSD device-level technique that eliminates the possibility of reads being stalled by any high latency operation. RAIL leverages redundancy to provide an alternative read path in the case a particular NAND chip is temporarily inaccessible due to performing a high latency operation. Unlike previous work, RAIL reduces tail latency at all percentiles, in particular, by 7× over existing approaches for the 99.99th percentile and for our tests always completes reads in under 1ms. To reduce the write bandwidth overheads, RAIL introduces *latency-aware hot-cold separation* (HC) to separate hot user writes and cold GC writes into independent physical flash chips. With this technique in place, we show that avoiding RaGC serialization, such as implemented by prior work becomes obsolete and can be skipped entirely. As a result, RAIL-HC also reduces write amplification by 2× and increases GC write bandwidth by 4× over prior work. Furthermore, in contrast to prior works that relied on simulation, we present a full Linux kernel software implementation leveraging OpenChannel SSDs.

---

[1]We analyzed a set of SSDs including CNEX Westlake, Intel P3600, Intel 750, and Samsung PM1725 all showing similar behavior.

Table 1. RAIL vs. Prior Work

Technique	Avoid RaW	Avoid RaGC	Avoid RaWL	Detect Stall	WR Overhead
Purity [20]	yes	no	no	react	high
Toleraid [30]	yes	no	no	proact	high
F-on-R [70]	yes	no	no	exact	high
TinyTail [80]	no	yes	no	exact	high
RAIL-HC	yes	yes	yes	exact	low

2 BACKGROUND

Prior Work: Prior work has leveraged redundancy to reduce tail read latency when accessing storage. Flash-on-rails [70], ToleRAID [30], and Purity [20] use parity across drives to improve tail ready latency by only writing to a subset of the drives at a time. Whenever a read is slow (timeouts) another read is triggered to recompute the data from parity. While simple to implement, these approaches can only prevent RaW serialization, however, they cannot guard against reads being serialized behind operations such as GC and wear leveling, as those are triggered by the SSD itself. Furthermore, waiting for a timeout at least doubles the read latency in average. TinyTail [80] isolates reads from GC, however, fails to prevent RaW serialization which occurs frequently. TinyTail also utilizes the internal copyback operation to move data which can lead to errors because NAND chips do not have error detection and correction internally, whereas RAIL re-computes error correction bits during GC on the host. TinyTail also has only been simulated, ignoring many of the NAND specific challenges of real designs discussed in Section 4. RAIL has passed an extensive two man year validation and verification phase to reach Linux kernel stability. The validation suite is implemented in QEMU [5] and emulates read/write/erase errors as well as static and grown bad blocks in a random constrained way to automatically test RAIL in different configurations.

All related works above [20, 30, 70, 80] achieve a tail latency reduction by throttling write bandwidth. By combining RAID with HC, RAIL-HC significantly reduces the negative impact on write performance. Table 1 compares the capabilities of prior approaches against RAIL. It shows whether a technique is capable of preventing RaW, **read-after-GC (RaGC)**, and **read-after-wear-leveling (RaWL)** stalls. It also shows whether the techniques require timeouts (react) to detect a RaW stall or if they always assume a stall (proact) or if they can determine whether a read will be stalled (exact). The last column shows the bandwidth overheads introduced by the approaches for reducing read latency. As we will show in this article, to guarantee low read latency at the very tail, all sources of high latency need to be addressed. We find that this can only be achieved by controlling SSD device properties on the hardware level exposing us to the intricate details of NAND Flash such as bad blocks, managing meta information, and handling write errors. RAIL overcomes these challenges by leveraging OpenChannel SSDs providing strict tail latency guarantees up to the 99.99th percentile.

Internal device parallelism: Modern Flash devices have multiple levels of internal parallelism [1, 16]. A Flash controller interfaces with multiple channels which are shared by multiple NAND dies. We refer to units of parallelism on Flash as **logical units (LUNs)**. On a typical Flash device, a LUN typically corresponds to a die, since each NAND die typically supports one outstanding operation at a time.[2] Tables 2 and 3 show the device specification for the SSD we use in our experiments. The device contains 128 LUNs in total, thus the number of concurrent operations is

---

[2]Most devices support multi-plane operations, which allow multiple identical operations per die, increasing throughput but not latency.

Table 2. Open-Channel SSD

SSD Properties	
Channels	16
LUNs per Channel	8
Total LUNs	128
Channel Bandwidth	280 MB/s

Table 3. LUN Properties

LUN Properties	
Sector Size	4 KB
Page Size	64 KB
Blocks	1067
Block Size	256 pages

Table 4. Per-LUN Performance

Op.	Bandwidth	Latency	Size
Read	280 MB/s	$65\mu s$	32 KB
Write	25 MB/s	$1,700\mu s$	32 KB
Erase	N/A	$6,000\mu s$	4 MB

limited to 128. With 64 KB pages, 256 pages per block and 1,067 blocks per LUN, each LUN has over 16 GB of capacity.

NAND properties: In NAND technology, erase operations are performed at the granularity of blocks, writes at the granularity of pages, and reads at the granularity of sectors. Writes can only set bits in a single direction (from one to zero). Thus, pages cannot be updated in place; they must first be erased. Table 4 shows the performance of read, write, and erase operations per LUN. Although SSDs deploy battery backed DRAM caches to complete writes to software instantaneously, a write can only be sent to a chip if all prior writes have persisted. Although, erases take 8 times as long as writes, they are a more efficient operation since they operate on 4 MB of data. Due to the asymmetric read, write, and erase latencies and the serialization of operations, read latency increases significantly when the requested data page resides on a LUN occupied by a write or erase operation.

Flash Translation Layer (FTL): The FTL is an essential layer in the Flash storage stack, managing L2P address mapping, GC, and wear-leveling while presenting a simple block interface to the operating system. User applications submit read and write requests to **logical block addresses (LBAs)**. Each logical block in the address space represents a sector-sized data segment (usually 4 KB) which the FTL maps to a sector of a physical page on Flash. Policies for physical data layout and GC (choosing which blocks to erase and when) in the FTL directly influence read/write performance of user applications on Flash.

LightNVM and OpenChannel SSDs: OpenChannel SSDs do not implement the FTL in firmware, but instead, expose the internal parallelism of SSDs to the host, enabling the operating system to manage physical storage [8]. The Linux kernel is an example of an operating system that supports OpenChannel SSDs through an abstraction layer called LightNVM [9, 11]. The LightNVM subsystem is an open-source host-based FTL that provides a generic media manager for wear-leveling and bad block management, uses a **physical page address (PPA)** I/O interface to communicate with the SSD (still over the standard NVMe specification), and exposes the Open-Channel SSD as a traditional block I/O device to user applications. We develop RAIL as part of LightNVM.

3 DESIGN

RAIL enforces strict tail latency guarantees by eliminating reads being stalled behind high latency operations such as writes and erases. Therefore, RAIL maintains redundant parity data for recomputing sectors, that reside on a LUN that currently serves a high latency operation. RAIL draws
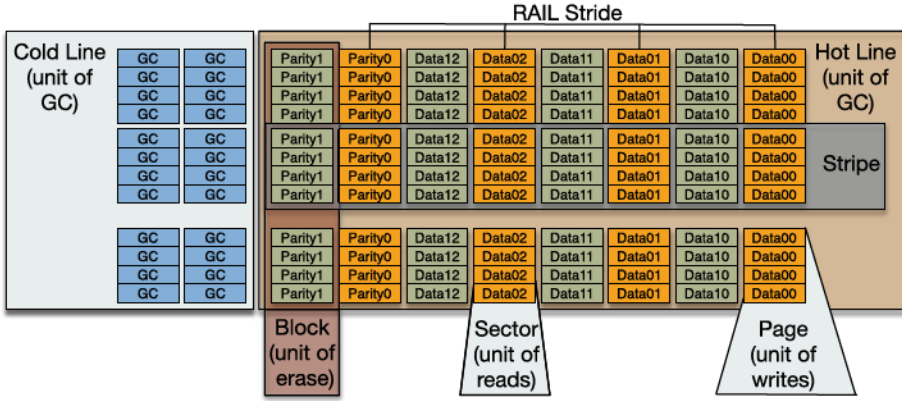
Fig. 2. SSD sector placement.

from techniques such as RAID [58], however, utilizes redundancy for latency reduction in addition to improving fault tolerance. RAIL parity computation is performed within **strides**, where a stride of size S consists of one parity element and S−1 data elements. Data elements are XOR'ed together to compute the parity element. Each element of a stride resides on a separate LUN. In the case where a LUN, for instance, LUN_0, is serving a high latency operation, reads targeting LUN_0 are served by reading the corresponding sectors from LUN_1, LUN_2, and LUN_3 and computing the original sector residing on LUN_0. We refer to such multi-LUN reads as a **RAIL read**. Note that block devices provide no ordering guarantees for reads and writes in the absence of flushes, enabling RAIL reads to complete before prior writes.[3] To enforce tail latency guarantees, RAIL ensures that only one LUN within a stride serves a high latency operation at all times, impacting write bandwidth.

3.1 Parity Sector Placement (RAIL)

In this section, we describe how RAIL manages sectors and places data and parity on the SSD. SSDs use device specific unit sizes for reads, writes, and erases. For instance, CNEX performs reads at the unit of sectors (4 KB), writes at the unit of pages (64 KB), and erases at the unit of blocks (1,024 KB). Each LUN contains multiple blocks, enumerated by an ID. All blocks with the same ID, for instance, the first block of all LUNs, are combined to form a **line**. Furthermore, all write unit (page) sized elements that have the same page ID and block ID are defined as a **stripe**. RAIL needs to allocate sectors for parity data to enable RAIL reads. One option would be to maintain additional information as part of the **logical to physical (L2P)** block translation table, which enables to find the other sectors and parity data in case of a RAIL read. As the L2P already consumes multiple GB in host memory for TB sized SSDs, we place parity data to fixed locations within a stripe using simple modulo operations to determine the target physical addresses of a RAIL read. We distribute RAIL strides over stripes to maximize write bandwidth and place parity data on the high order LUNs of a stripe as parity can only be computed after all data sectors of a stride have been written. Figure 2 shows how RAIL manages sectors, blocks, stripes, and lines as well as how it places data and parity elements. In the example, two RAIL strides are shown, where stride 0 consists of Data00, Data01, Data02, and Parity0.

---

[3]Ordering needs to be enforced by the filesystem or application.

3.2 Latency-Aware Hot-Cold Separation (HC)

Vertical hot-cold separation [24, 48, 72] reduces write amplification by separating frequently written LBAs (hot) and rarely written LBAs (cold) into separate flash blocks. This increases the probability that all sectors within hot blocks are overwritten by user writes before the block is garbage collected, minimizing the number of valid blocks that need to be moved by GC. Hot-cold separation is implemented by maintaining two open blocks per NAND chip whereas user writes are applied to the hot block and GC writes are applied to the cold block. We introduce horizontal hot-cold separation, separating user and GC writes for the purpose of reducing tail latency. In contrast to prior approaches that applied GC and user writes to separate blocks, our technique allocates hot and cold partitions from separate NAND chips guaranteeing that user and GC writes do not utilize the same LUN. For workloads with a zipfian distribution, reads are likely to access hot data and hence the probability is low that reads are serialized behind cold writes. As a result, it is no longer mandatory to throttle writes to only one LUN per stride for the cold block, eliminating the write overhead introduced by avoiding RaGC serialization. Horizontal hot-cold separation introduces no capacity or bandwidth overheads and hence represents a more effective technique than prior approaches [67, 78, 80] focusing on the read-GC interference. Enabling horizontal hot-cold separation is challenging as the ratio of the number of hot to cold blocks depends on the dynamic write amplification factor. Therefore, RAIL-HC continuously monitors the user and GC writes and, every 1M total writes, based on the the write-amplification factor, it assigns LUNs to user and GC lines accordingly, rounding-up the number of LUNs assigned to GC. To avoid high tail read latency when re-assigning LUNs between hot and cold lines, RAIL-HC supports LBA live migration. For instance, if a user LUN is re-assigned to a GC line, the old data continues to be accessible via RAIL reads, while new written pages are allocated according to the most recent partitioning scheme. As blocks are invalidated and erased over time, all blocks within a LUN gradually converge to the new partitioning scheme. Consequently, blocks within the same LUN may be part of a different line configuration. Therefore, every line contains less than 100 bits of additional meta information defining the stride width and GC/user configuration enabling the correct line-specific RAIL read access pattern. Note that migrating LUNs occurs rarely (after millions of writes) as even an abrupt change of the user write pattern, for instance, from random to sequential, suffers from inertia. In particular, a large fraction of the SSD needs to be overwritten by a new user write pattern before the GC behavior starts to change. Furthermore, a temporally non-optimal allocation of user and GC LUNs only reduces write bandwidth but does not affect tail read performance. For instance, if there are not enough cold LUNs, GC will need to be performed on hot lines reducing user write bandwidth, however, without affecting user read latency.

3.3 RAIL Implementation

RAIL is implemented within the Linux kernel's PBLK [11] subsystem in 1,618 C lines of code. Figure 3 provides an overview of PBLK and the RAIL modifications in blue.

Write path: Write requests updating a particular LBA are inserted into a single shared ring buffer by all blk-mq [10] kernel threads. Written sectors are immediately completed to the block I/O layer and buffered in DRAM until enough sectors are available to write a whole page. In the case of a flush (sync) operation, available sectors are padded with empty sectors to form a page and then mapped to a PPA. PBLK then sends the write operation to the Flash hardware controller over the standard NVMe interface [55] using the PPA as the address. Writes are performed round-robin across LUNs within a stripe in the hot line to maximize LUN parallelism and bandwidth.

To implement RAIL, we add a PPA to write buffer entry (P2B) mapping table (16 KB) which references all write buffer entries forming a RAIL stride. The P2B is required, as there is no static
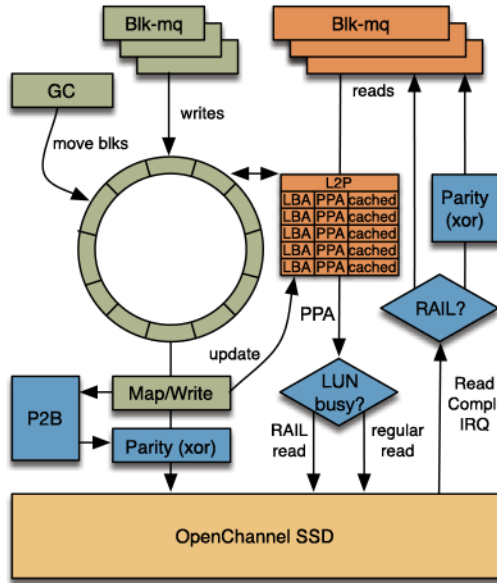
Fig. 3. RAIL architecture.

fixed mapping between write buffer entries and hardware sectors because of existing bad blocks, padded sectors, and meta data. RAIL extends the PPA mapping mechanism such that, whenever, a parity LUN (e.g., `Parity0` in Figure 2) is to be mapped, the RAIL parity engine is triggered to generate a parity write. The RAIL parity engine queries the P2B, computes the parity from the cached sectors, and generates the PPA for writing the parity to the SSD. RAIL introduces stride semaphores to enforce that at most one LUN per RAIL stride is written or erased at a time reducing write bandwidth to $1 \div S$. However, this only applies to hot writes. By leveraging hot-cold separation, RAIL-HC does not require to throttle cold writes, minimizing the negative impact on write bandwidth. Furthermore, trading-off write bandwidth for low tail latency is acceptable for the following reasons: (1) Many datacenter workload studies [18, 32, 36, 43, 62, 63] have shown read to write ratios of 4 to 1 and higher. As our SSD provides the same maximal read and write bandwidth of 1.6 GB/s, applications are generally limited by read and not write bandwidth. (2) Due to shared resources (PCIe, channels, and controller) the maximum bandwidth is shared between reads and writes. Hence, even for a 1 to 1 read/write ratio, RAIL only reduces write bandwidth by 50% effectively. (3) RAIL detects write-mostly (<1K read IOPS) workloads and automatically disables LUN throttling. (4) As shown in Section 5 applications such as MongoDB and RocksDB are not performance limited by write bandwidth.

Read path: Reads are served by looking up the LBAs (multiple LBAs in the case of multi-sector reads) in the L2P to determine the PPA. Since blk-mq threads directly serve read requests, multiple threads can have multiple asynchronous read requests in flight at the same time. To integrate RAIL, we check, for each PPA, whether the target LUN currently serves a high latency operation. In this case, we perform a RAIL read by transforming the PPA into its corresponding $S-1$ RAIL PPAs. We issue asynchronous read requests for all PPAs and, in the interrupt handler, complete the I/O by copying the recomputed reverse-parity of the RAIL PPAs into the original kernel block I/O (`struct bio`).

Garbage collection and Wear Leveling: In addition to managing userspace I/O, PBLK generates write and erase operations to implement GC and WL. PBLK performs GC at the granularity of

lines by reading all valid 4K sectors from a line and then writing them into the same write buffer into which user writes are placed. From the write buffer they are written to the next open line as any other user write. To implement hot-cold separation we tag the write buffer entries so that GC'ed sectors are written to the cold line and user sectors to the hot line. We also implemented an alternative approach utilizing separate buffers for GC and user writes which performed equally well. PBLKs original GC mechanism greedily determines the line with the least amount of valid sectors, moves them to a new line and then erases all blocks within the line. With hot-cold separation it becomes beneficial to prefer cold lines over hot lines [72]. RAIL does not require further modification of the GC path, except for marking parity sectors as invalid so that they are not moved by GC. Note that overwriting LBAs does not affect parity computation of other sectors within the same stripe. As sectors are never updated in place, the physical sectors can still be used for parity computation although they have been logically invalidated.

3.4 RAIL Overheads

RAIL enforces low tail latency at the cost of storage capacity and bandwidth. However, RAIL also offers fault tolerance guarantees over a non-RAID approach.

Capacity overheads: RAIL induces capacity overheads of $1 \div S$, where S is the stride size, to store parity data. For instance, for S=4 the capacity overhead is 25%, for S=16 the capacity overhead is 6.25%. Note that RAIL enables fault tolerance as provided by RAID. In applications where RAID needs to be deployed anyways, RAIL's capacity overheads can be zero. RAIL-HC only requires parity blocks for hot data, hence, depending on the distribution of hot and cold data the capacity overheads are reduced, often by 2× as shown in Figure 6. Furthermore, Section 5.1.5 shows that RAIL allows to reduce over-provisioning compensating RAIL's capacity overheads.

Bandwidth overheads: RAIL affects total device bandwidth (TotalBW) which is shared among reads, writes. and erases and also effects read and write bandwidth individually. TotalBW is reduced by $UserWrBW \div S$ as additional parity data needs to be written to the SSD. For RAIL (but not for RAIL-HC), TotalBW is reduced by $GcWrBW \div S$ as additional parity data needs to be written for garbage collected sectors. UserWrBw is limited to $1 \div S$ to ensure that only a single LUN is serving a high latency operation at a time. Note that for mixed read-write workloads this is generally not an issue as the remaining bandwidth $TotalBW - 1 \div S$ can be used for reads. If maximum write bandwidth is required such as for preconditioning or bulk-loading data, RAIL automatically disables LUN throttling in the presence of <1K read IOPS while maintaining parity computation. Effective user read bandwidth is reduced by the read amplification of RAIL reads. We quantify the effect of read amplification in Section 5.1.4.

4 IMPLEMENTATION CHALLENGES

Implementing RAIL on real hardware has been a challenging endeavor due to the technology specific properties of NAND Flash. In the following sections, we list the most challenging issues we faced and addressed.

4.1 Bad Blocks

NAND Flash is an inherently unreliable storage medium. New devices generally contain a number of bad blocks that are unusable and, furthermore, write and erase operations wear out NAND memory over time, increasing the number of bad blocks over time. The number of erase cycles before a block wears out is technology dependent and determined, for instance, by the number of voltage levels (single level vs. multi level cells). LightNVM supports bad block management by maintaining a list of bad blocks to guarantee that LBAs are never mapped to a bad sector. Bad blocks and, in particular, grown bad blocks are problematic, as RAIL utilizes a fixed mapping between the

data/parity sectors of a RAIL stride and the LUNs. As a result, whenever performing a RAIL read, the read path needs to determine the number of bad sectors within a stride so that it can issue the correct number of sector reads. Similarly, on the write path RAIL needs to be aware of bad blocks such that they can be skipped during parity computation. The P2B on the write path described in Section 3 maintains all valid PPA to write buffer entry and invalid (bad block) mappings to compute the correct parity in the presence of strides with fewer than S sectors.

4.2 Meta Data

LightNVM maintains meta data to store information about bad sectors, overwritten sectors that can be GC'ed, sequence numbers and other information required for recovery in the case of a power cycle. Meta data is stored on Flash within the start and end sectors of a line and hence meta sectors are not available for storing data. Even worse, due to potential bad blocks at the start or end of a line, the location of meta sectors is not fixed. There exist many corner cases in the presence of bad and meta sectors which need to be considered to compute correct parity in all cases both on the read and write path.

4.3 Flushes

Block devices support flushes to enforce the required consistency guarantees of filesystems and databases. For instance, in ACID databases, durability can be enforced by issuing a flush operation before committing a transaction. Flushes require LightNVM to immediately issue all writes in prior to the flush to the storage device by padding the flushed sectors with zeros to form a full page, the unit of writes. As the padded sectors do not traverse the write buffer, RAIL maintains additional information to correctly compute parity for the padding sectors.

4.4 Reverse LBA Map

The logical block (LBA) to physical sector (PPA) mapping table (L2P) is resident in main memory and lost during a power cycle. LightNVM recovers the L2P during boot up from the recovery data stored as part of the meta section on the device. To store the actual LBA that maps to a particular sector, LightNVM utilizes out-of-band memory on Flash to store a reverse L2P. In addition to recovery, LightNVM leverages the out-of-band LBA data for verification purposes. On every read, the out-of-band LBA information is compared to the expected LBA from the memory resident L2P. For RAIL reads, this verification check fails as the sectors utilized to re-compute the original page are mapped to different LBAs. We address this issue by computing a parity LBA for each RAIL parity page by XOR'ing all LBAs of a RAIL stride. On the read path, this enables to re-compute the original LBA for read verification. XOR'ed RAIL LBAs are skipped during L2P recovery.

4.5 Multi-Sector Reads

To support different read sizes, PBLK allows to read multiple sequential LBAs as part of a single access, although they might be mapped to non-consecutive PPAs. To implement this capability, PBLK utilizes scatter operations composed of a vector of PPAs. In the case of RAIL, it is possible that some of the sectors need to be read utilizing conventional reads, some sectors are read using RAIL reads and some sectors are read from the write cache. To reduce the number of permutations and corner cases, we re-factored the code to offer three different code paths reflecting the potential location of a sector. We then scan the entire multi-sector request before emitting up to three separate asynchronous read requests which finally get assembled to complete the original block I/O (bio) request.

4.6 Locking and Synchronization

Linux kernel developers need to deal with various concurrency issues. LightNVM leverages the blk-mq interface to enable concurrent access of multiple readers to the NVMe block device. For performance reasons, each reader can issue multiple overlapping asynchronous reads and also needs to handle completion events (interrupts) that can preempt the read path at any time. While the CNEX OC driver and SSD can process multiple outstanding reads at a time, it can only process one write or erase operation which needs to be enforced via per LUN semaphores. RAIL has been carefully designed to comply with all the locking and synchronization policies built into LightNVM and has been validated using an extensive verification suite.

4.7 Write Buffer Races

The write buffer introduces a potential race condition which occurs when transferring sectors from the buffer to the device in the presence of concurrent reads to the same sectors. In particular, it is possible that some sectors of a RAIL stride are persisted to the SSD while some are still in the write buffer. Additionally, NAND chips specify the concept of upper and lower pages, distributed within a block at a certain distance whereas the upper page can only be read after both the lower and upper page have been written. We guarantee consistent reads under all these conditions by delaying the L2P update such that reads are only served from the device if all above conditions are met.

5 EVALUATION

We evaluate RAIL on the CNEX Westlake Open-Channel SSD [8] with the properties shown in Table 2. The SSD is connected over a PCIe ×8 interface to the host server, an Intel Xeon Broadwell E5-2630 with 20 cores, and 40 SMT threads running at 2.2 GHz with 64 GB of DRAM. Our system runs Ubuntu 18.4 Linux with a 5.1 kernel for the unmodified PBLK baseline. We utilize nvme-cli to initialize LightNVM's media manager and expose the Open-Channel SSD as a block device.

We compare RAIL and RAIL-HC against two baselines: PBLK which represents the unmodified LightNVM subsystem and LinuxTinyTail, our Linux re-implementation of TinyTail [80]. Linux-TinyTail conceptually resembles TinyTail, in that it leverages redundancy to reduce tail latency in the presence of GC. In particular, whenever the GC mechanism issues writes or erases, it accesses only one unit of the RAID array at a time to avoid GC overheads impacting read tail latency. Linux-TinyTail is entirely implemented in software on top of LightNVM and hence our implementation lacks the (simulated) hardware acceleration of the original TinyTail proposal. We evaluate RAIL with microbenchmarks using the **Flexible I/O** tester (**FIO**) [40] and two database applications: RocksDB and MongoDB. We also evaluated Twitter's Fatcache [60] but omit the results for brevity (RAIL shows a 4× tail latency reduction over PBLK for Fatcache). For all tests, we precondition the SSD with sequential writes and then 4K random writes. For all tests, if not mentioned otherwise we utilize $S = 4$ for both RAIL and RAIL-HC.

5.1 Flexible I/O Tester

We perform a series of microbenchmarks using FIO. All tests are performed directly on the block device without a filesystem and page cache (O_DIRECT). We use a thread count of 40, and low queue depth of 2 for all tests. We enable the Kyber [65] I/O scheduler for all tests and set it to a target read latency of 500μs. While the SSD device can be saturated by two threads, high queue depth has a detrimental effect on latency due to request batching. We leave optimizing the Linux block I/O layer for low tail latency as future work.
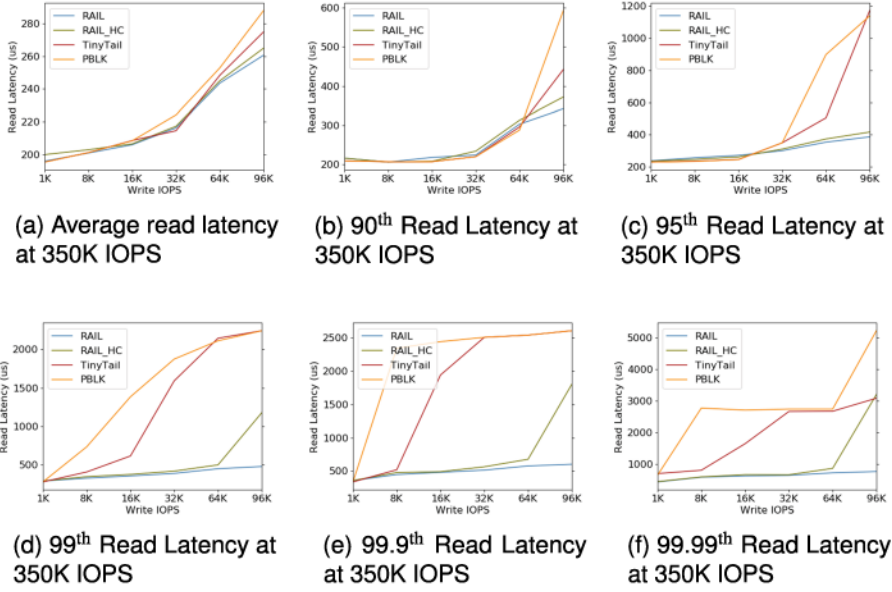
Fig. 4. RAIL-HC provides predictable, low tail read latency at high write throughput.

5.1.1 Read Latency vs. Write IOPS. As described in Section 3, high read tail latency is caused by reads being serialized behind high latency operations such as writes, erases, and GC. Our first series of microbenchmarks shown in Figure 4 plots read latency against write IOPS on the X-Axis. For this test, we utilize the entire SSD capacity by preconditioning all LBAs using a standard over-provisioning ratio of 12%. When executing non-sequential write workloads on a full SSD, GC kicks in immediately. The resulting write amplification ranges from $1\times - 10\times$ depending on the write distribution, over-provisioning ratio and hot-cold separation technique. For realistic zipfian distributions, the observed write amplification ratio generally ranges between $1.8\times$ and $4.5\times$ [23, 33, 73]. To run our tests, we execute 40 threads that issue 100% 4K sized random reads with a zipfian distribution and one write thread that issues 4K sized random writes with the same zipfian distribution. The test utilizes a fixed read bandwidth of 350 KIOPS which is close to the SSD's peak read only throughput of 380K IOPS and we sweep the target write IOPS from 1K to 96K IOPS. Note that, because of write throttling, the four approaches support different maximum write performances. In particular, for this setup, the approaches reach a maximum write IOPS (in the absence of reads) of 169K for PBLK, 80K for RAIL-HC, 49K for LinuxTinyTail, and 41K for RAIL. We evaluate write performance in more detail in Section 5.1.3. As shown in Figure 4(a), for all approaches read latency generally increases in the presence of writes. RAIL provides a 20% average latency reduction over PBLK at maximum write IOPS. For the 90th percentile tail latency, shown in Figure 4(b), RAIL reduces latency from $600\mu s$ to below $340\mu s$. For the 99.9th percentile, RAIL again shows a 5x latency reduction already for low write bandwidth such as 8K IOPS. At the 99.99th percentile, shown in Figure 4(f), RAIL provides a read latency of below $900\mu s$ for all write rates, whereas for PBLK, tail latency increases to $5300\mu s$ for maximum write IOPS. Figure 4 shows that LinuxTinyTail successfully addresses read after erase stalls that in PBLK introduce up to $6ms$ latency (Table 4), however, it only provides little performance improvement over the PBLK baseline as it cannot avoid RaW serialization. This shows that avoiding GC interference is not enough; RaW stalls need to be addressed as well. RAIL-HC performs almost on par with RAIL. The zipfian distribution of the test workload ensures that most reads hit the hot data partition and only few
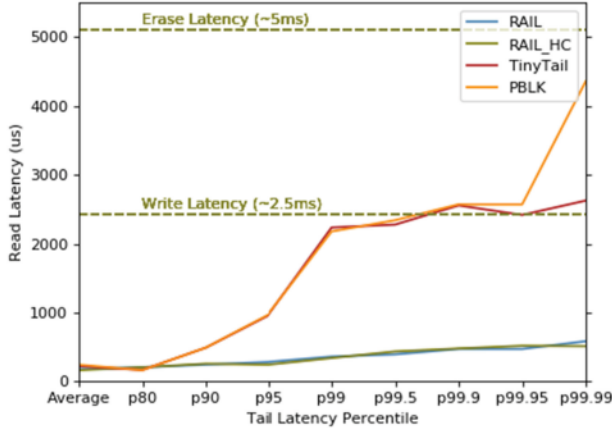
Fig. 5. Tail latency without GC.

reads are served from the cold partition. As a result, RaGC serialization is almost non-existent and hence RAIL-HC enables low tail latency without throttling write performance as significantly. Note that RAIL-HC only works well for workloads that follow a zipfian distribution. For completely uniform distributions, there exists a high probability that reads hit cold data in which case RAIL-HC does not provide sufficient read-write isolation. For such workloads, RAIL is the only approach that enforces low tail latency.

5.1.2 Read Latency vs. Percentile Latency (No GC). We also evaluated tail latency for the case where there exists sufficient free space (75%) on the SSD. In this case, there exist 4× as many physical than logical blocks on the SSD which delays GC significantly and hence reduces write amplification to 1.05×. As the GC traffic is significantly reduced, LinuxTinyTail provides very little latency benefits over PBLK, while the RAIL approaches continue to maintain low tail latency. Figure 5 shows the results of the non-GC workloads in condensed form. We use the same setup as in Section 5.1.1 while only filling the SSD to 25%. RAIL consistently provides lower latency ranging from $100\mu s$ in average to $585\mu s$ at the 99.99th percentile. In contrast, PBLK shows a tail latency of $4358\mu s$ and LinuxTinyTail shows a 99.99th percentile latency of $2834\mu s$. RAIL outperforms the baselines at all percentiles whereas LinuxTinyTail can only avoid the rare read being stalled behind erase scenarios while it suffers from the much more frequent RaW stalls. Due to the lack of GC traffic, RAIL, and RAIL-HC perform almost identical in terms of read latency and write throughput.

5.1.3 Write Performance and Request Size. We omit write latency graphs as, in absence of flushes, writes are immediately completed when they enter the DRAM write buffer. As a result, writes generally complete in less than $20\mu s$ independent of the evaluated mechanism. Write throughput as measured by FIO is reduced by all latency avoidance techniques. The write overhead, hereby, depends on the GC write amplification, the stride width S and whether reads are isolated only from GC writes, from user writes or from all operations. We evaluate the write overhead of the different techniques for $S = 4$ and different write amplification factors using a write workload with zipfian distribution. Using FIO, we vary *theta-zipf* to generate write amplification factors of 2.3, 3.2, 4.1, and 5.0 which match our observations of real applications as well as prior work on write amplification analysis [23, 33, 73]. Figure 6 shows the achieved maximum write throughput of the different approaches normalized to PBLK. RAIL provides the strictest tail latency guarantees at the cost of reducing bandwidth to approximately $1 \div S$. TinyTail and RAIL-HC only throttle
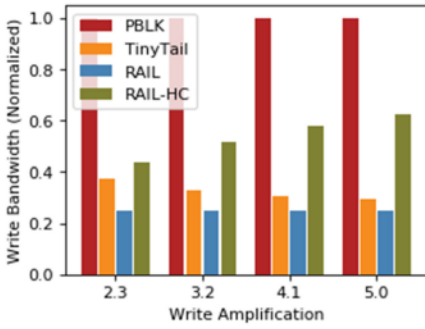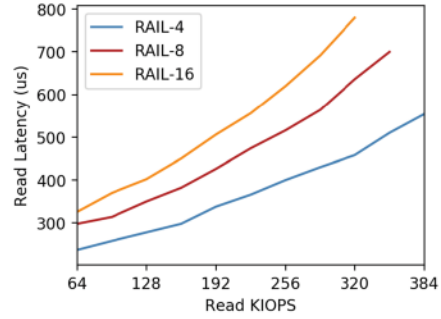
Fig. 6. Write bandwidth overhead.

Fig. 7. Capacity vs. latency.

cold GC, respectively, hot user write traffic and hence limit their impact on write performance. For a write amplification ratio of 5× RAIL-HC can maintain 62% of PBLK's original bandwidth, whereas TinyTail achieves only 29%, while RAIL-HC enforces much stricter tail latency guarantees than TinyTail as shown in Section 5.1.1. We will show in Sections 5.2 and 5.3 that overall, write throttling has a small effect on end-to-end application performance.

We also evaluated the performance of RAIL and PBLK for different request sizes. In PBLK, writes are always performed at the granularity of 64K, independent of the size of the original request, by slicing and reassembling segments in the write buffer. For reads, in the case of RAIL, we could not see any significant performance impact of larger requests and hence omit the results here for brevity. The reason is that read requests are always scattered into a series of 4 KB requests and RAIL emits RAIL reads for any of the 4K sector reads that would be blocked by a high latency operation. In the case of PBLK, tail latency in fact increases for larger request sizes as the probability increases that one of the 4 KB reads is serialized behind a write operation. For instance, for 256 KB sequential reads PBLK already shows a latency of greater than 2*ms* for the 80th percentile.

5.1.4 RAIL Stride Width. As described in Section 3.3 the stride width S refers to the number of sectors that are used to compute a parity sector. A high stride width is advantageous as it reduces the capacity overhead of RAIL. For instance, a stride width of 16 (RAIL-16) only introduces a capacity overhead of 6.25% to store redundant parity information. On the other hand, RAIL-16 also reduces maximum write bandwidth by 16x and increases read amplification, as on a RAIL read 15 sectors have to be read to recompute the original sector. Figure 7 compares three RAIL configurations and their impact on read IOPS and 90th percentile latency. We run 1 write thread issuing 30K IOPS and 32 read threads that sweep their aggregate IOPS from 32K to 384K IOPS. Due to read amplification, RAIL-16 reduces the maximum read IOPS by 64K and increases the tail latency by 220*μs*. Configuring stride width enables users to trade-off capacity, IOPS, and latency in an application specific way.

5.1.5 Over-provisioning vs. Tail Latency. SSD vendors over-provision NAND Flash memory to compensate for GC overheads, by exposing fewer capacity to the user than available on the SSD. The amount of capacity reserved for GC determines the number of valid sectors that need to be moved between blocks, significantly affecting the read-write interference [52, 71, 73]. Data center operators commonly reserve up to 30–50% of raw SSD capacity for space over-provisioning to improve predictability in the presence of random writes [56]. We evaluate the impact of over-provisioning for RAIL by comparing three over-provisioning factors: 20%, 11%, and 7%. As Figure 8 shows, RAIL completely eliminates read stalls even for very low over-provisioning factors and operators may be able to choose lower over-provisioning factors reducing capacity overheads.
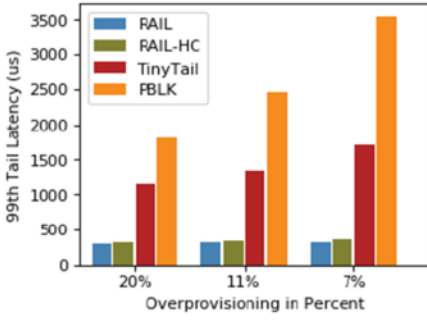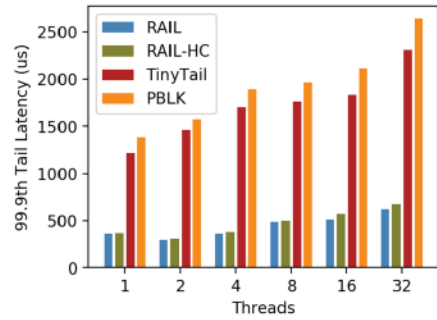
Fig. 8. Latency vs. over-provision.

Fig. 9. RocksDB latency.

LinuxTinyTail improves tail latency as it does not suffer from the increased GC overheads induced by low over-provisioning factors but tail latency is still affected by RaW stalls.

5.2 RocksDB

We evaluate the performance of RAIL using the RocksDB key-value store database in version 5.10.3. To run RocksDB, we create an Ext4 filesystem to place both the database and write ahead log onto the mounted SSD. As load generator we utilize db_bench configured with the default parameters. We *Bulkload* 100M key/value pairs into the database with 20 byte keys and 400 byte values which completes in 4,300 seconds for both PBLK and RAIL. *Bulkload* generates a write bandwidth of 233 MB/s which is below the 400 MB/s RAIL supports in an $S = 4$ configuration and well below the 1.6 GB/s supported by PBLK and RAIL in write-only mode. Figure 9 shows the tail latency of db_bench's ***ReadWhileWriting (RWW)*** and ***ReadWhileMerging (RWM)*** workload for RAIL and PBLK utilizing 1 through 32 threads. We omit db_bench's other benchmarks as they are either read-only or write-only. RAIL shows 4× lower tail latency at the 99.9th percentile than PBLK for less than 32 threads whereas for high thread counts software queuing increases latency for both approaches. Figure 10 lists the **queries per second (QPS)** that can be achieved with both approaches. 140K QPS translate into 330 MB/s which is below the throttled bandwidth of RAIL which explains that all approaches are able to achieve similar performance. TinyTail performs similar as PBLK (less than 10% latency improvement) while RAIL-HC closely follows the performance of RAIL.

5.3 MongoDB

We evaluate the performance of RAIL with the document-oriented NoSQL database MongoDB [3, 19]. We utilize MongoDB version 3.4.7, placing the database file on the Open-Channel SSD, utilizing the XFS filesystem as recommended by MongoDB. As a workload generator, we leverage **Yahoo Cloud Server Benchmark (YCSB)** [21]. We first load the database with 1 billion entries of $1KB$ in size to generate a $1.2TB$ backing file, filling up the SSD to 75% capacity. As most benchmarks utilize random writes, this fill level leads to significant GC. We run $1M$ transactions of each of the workloads defined by YCSB. Figure 11 shows the 99.9th percentile read latency as reported by YCSB for PBLK, LinuxTinyTail, and RAIL. Workloads B, C, D are read-heavy (95% reads) and hence all three implementations perform well providing sub millisecond tail latency. RAIL outperforms the two baselines for workload A (50% reads/50% writes) and F whereas F shows read and F2 shows the read-modify-write latency performed by workload F. We omit the results for workload E for brevity which performs scans that take $30ms$ to complete for all 3 approaches. Figure 12 shows the aggregated read/write throughput measurements for the same set of workloads. All three approaches achieve the same QPS and benchmark execution time.
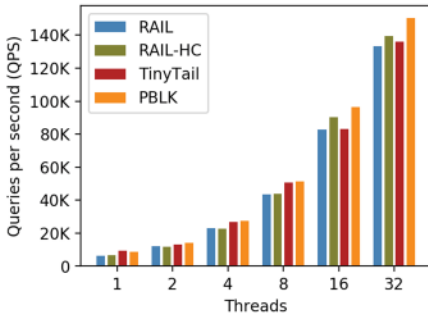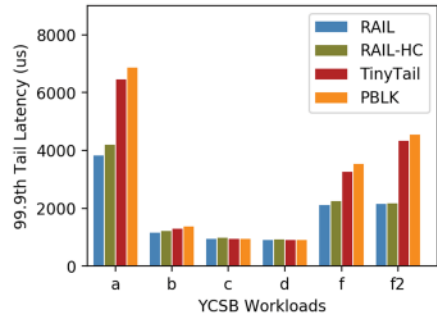
Fig. 10. RocksDB throughput.
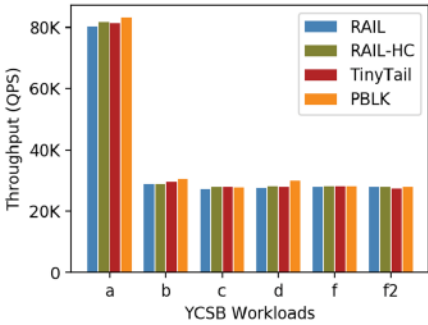

Fig. 11. MongoDB latency.
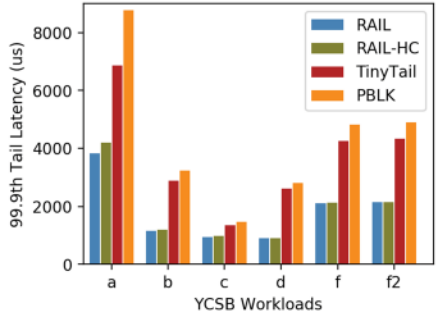

Fig. 12. MongoDB throughput.

Fig. 13. MongoDB interference.

We now study application interference by running the same MongoDB workload as above while running an FIO workload concurrently that generates a steady write workload of 10K IOPS. The performance of MongoDB suffering from application interference is shown in Figure 13. While RAIL performance is unaffected, tail latency increases by up to 3× for PBLK in the presence of application interference. This shows that RAIL cannot only enforce low tail latency within a single application, but that it also reliably avoids application interference.

6 DISCUSSION

6.1 FTL-Application Trade-offs

While RAIL could be implemented in hardware, the benefit of our host-based, software FTL approach is to expose design trade-offs to users. Based on application requirements and device characteristics, users can tune the stride size and over-provisioning factor, to trade-off capacity, bandwidth, and latency. For instance, if write bandwidth is more important than minimizing capacity overheads, RAIL can be configured in RAID-6 mode which utilizes two parity LUNs per stride. RAID-6 mode does not change the tail latency behavior, however, it further increases write bandwidth (as 2 LUNs can be written at a time) at the cost of additional capacity. In particular, in a configuration with three data LUNs and two parity LUNs, write throughput and capacity overheads both increase by 1.6×.

6.2 Tail Latency Aware OSes

RAIL enforces predictable low tail latency for Flash accesses. Nevertheless, independent software layers can introduce high tail latency, jeopardizing RAIL's effectiveness. While the Linux block I/O

layer has significantly improved scalability and throughput to support high performance NVMe storage devices, achieving low end-to-end tail latency remains a challenge. We provide three insights, that we believe are crucial to achieve low tail latency storage access on existing Linux systems.

Filesystem Bypass: Running the same workload in FIO over an Ext4 filesystem increases tail latency by $2 - 3\times$ over directly accessing the I/O device. As a result, to achieve low tail latency, applications need to operate on block devices directly bypassing the filesystem layer. Block device virtualization techniques such as provided by PBLK, enables sharing devices among multiple applications without the need of a filesystem.

Read/Write Separation: RAIL eliminates reads being stalled by independent writes. In addition, developers need to guarantee to never serialize any reads behind writes on the application layer. This design principle can be implemented with relative ease for databases or caches that have limited consistency and atomicity requirements but becomes challenging when implementing a database with strict ACID properties.

Over-provisioning of cores is costly, although, data plane operating systems such as Arrakis [59], IX [4], and ReFlex [46] have shown that low tail latency can be achieved without sacrificing throughput and efficiency while user level networking (DPDK) and storage (SPDK) stacks have also shown to provide predictable high performance. POSIX OSes such as Linux need to be re-architected with tail latency as a first order concern to enable end-to-end low latency systems.

6.3 Hardware Acceleration

Open-Channel SSDs move compute cycles from the SSD controller to the host processor in order to increase flexibility and programmability. As host processors represent a costly resource, this may negatively affect the **total cost of ownership** (**TCO**). We believe that both programmability and cost efficiency can be achieved by an Open-Channel design that leverages the following techniques.

Parity Acceleration: RAIL consumes 10% of the compute cycles of a modern Intel Skylake class processor core for parity computation. We propose a new NVMe command that, given a set of source PPAs and a destination PPA, computes parity of the source sectors and writes it into the destination sector. This hardware offload would also reduce PCIe bandwidth as the parity sector would not have to be transferred from the host to the SSD, while maintaining the flexibility of performing the sector mapping in software.

Garbage Collection: GC overheads can be reduced by introducing an NVMe memcpy command that copies the sector of one PPA to another. With this approach, the over-provisioning factor, GC aggressiveness, remapping strategy, and line selection remains fully programmable in software while PCIe bandwidth and host CPU cycles are reduced significantly.

Flash Architecture: When designing a Flash chip, NAND architects need to trade-off cost, capacity, bandwidth, and latency. For instance, to amortize the high cost of erases, they are performed on a large block granularity, sacrificing latency for throughput. Writes are also batched (see Section 3) to increase throughput, however, batching is limited as it can lead to high tail latency when a read is stalled behind a write. With RAIL, write latency becomes irrelevant and hence very large pages can be supported that can improve write throughput. Furthermore, with RAIL, SSDs no longer need to support a large amount of LUNs to reduce RaW serialization.

7 RELATED WORK

Data center level techniques have been proposed to improve storage tail latency performance for maintenance [2], video serving [7], and remote storage access [46] while Limplock [25] analyzes the performance impact of unreliable storage hardware.These approaches are application specific and cannot provide the same guarantees as RAIL. Many Flash-based systems account

for asymmetrical read-write latency in I/O scheduling decisions [57, 66, 68]. LOCS, a key-value store database implemented directly on an Open-Channel SSD, schedules read, write, and erase operations using a least-weight-queue-length policy to maximize Flash utilization and throughput [76]. ParaFS also weighs requests as part of its parallelism-aware scheduling algorithm [82]. Prefetching [14] has been proposed to reduce tail latency. In contrast to prior techniques, RAIL is the first device-level technique that supports strict tail latency guarantees.

Redundancy: Storage systems use replication or erasure coding to improve reliability [13, 26, 34, 37, 38, 45, 50, 51, 69]. Several systems and studies have shown that redundancy is an effective way to reduce tail latency [22, 27, 75, 79]. EC-Cache uses erasure codes and late binding of redundant requests to reduce tail latency and improve load balance for in-memory cache systems [61]. These approaches requires application-level changes while RAIL provides stronger guarantees while being transparent to the application layer.

Data placement: Prior work has examined various page allocation schemes on Flash to leverage internal device parallelism [17, 41, 74]. Gordon [12] uses a 2-D striping scheme to leverage channel and die-level parallelism, increasing throughput. OFSS is an object-based FTL co-designed in hardware/software to reduce write amplification [53]. Chopper [31] and F2Fs [47] are two file system proposals that improve performance for Flash based storage devices. Autostream [81] separates writes into streams to improve data placement. RAIL differs from these systems by making data placement decisions based on reducing the probability of read-write conflict, thus improving tail latency.

Garbage Collection: Several systems [15, 20, 29, 44, 49, 54, 77, 80] have determined GC as the culprit for high tail latency and try to reduce or eliminate its effect. While GC arguably has a strong impact on tail latency, we showed that addressing read-write interference is even more important. User writes increase read latency by an order of magnitude and so far have been ignored by prior work.

8 CONCLUSION

We described RAIL, a Flash management technique that relies on redundancy to improve the tail read latency in the presence of high latency operations. We showed that RAIL's page placement algorithm and parity-based read datapath eliminates the possibility of a read operation getting stuck behind writes and erases, allowing RAIL to achieve 7× lower tail read latency than a conventional SSD. We implemented RAIL within Linux PBLK, a host-side, software FTL whose design parameters can be tuned by users to balance trade-offs between tail latency QoS, bandwidth, capacity, and fault-tolerance according to application requirements and device properties. RAIL contributes over prior approaches by avoiding RaW serialization, enforcing stricter tail latency guarantees and by reducing write overheads by leveraging HC.

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*. 57–70.

[2] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. 2015. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 457–473.

[3] Kyle Banker. 2011. *MongoDB in Action*. Manning Publications Co.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the USENIX Operating Systems Design and Implementation*. 49–65. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[5] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, Vol. 41, 46.

[6]   R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. 2003. Introduction to flash memory. *Proceedings of the IEEE* 91, 4
      (April 2003), 489–502.
[7]   Yitzhak Birk. 1997. Random RAIDs with selective exploitation of redundancy for high performance video servers. In
      *Proceedings of the IEEE 7th International Workshop on Network and Operating System Support for Digital Audio and
      Video.* IEEE, 13–23.
[8]   Matias Bjorling. 2016. Getting Started with Open-Channel-SSD with CNEX SSD(CNEX Labs Westlake ASIC).
      Retrieved on 06 Jan., 2022 from http://lightnvm.io/.
[9]   Matias Bjorling. 2016. Open-Channel Solid State Drives NVMe Specification. Retrieved 01 Jan., 2021 from http:
      //lightnvm.io.
[10] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: Introducing multi-queue SSD
      access on multi-core systems. In *Proceedings of the International Systems and Storage Conference.* ACM, 22.
[11] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The linux open-channel SSD subsystem. In
      Proceedings of the USENIX File and Storage Technologies, FAST'17. USENIX Association, 2017.
[12] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. 2009. Gordon: Using flash memory to build fast, power-
      efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural
      Support for Programming Languages and Operating Systems.* ACM, 217–228. DOI : https://doi.org/10.1145/1508244.
      1508270
[13] Chandranil Chakraborttii and Heiner Litz. 2020. Improving the accuracy, adaptability, and interpretability of SSD
      failure prediction models. In *Proceedings of the 11th ACM Symposium on Cloud Computing.* 120–133.
[14] Chandranil Chakraborttii and Heiner Litz. 2020. Learning I/O access patterns to improve prefetching in SSDs. In
      Proceedings of the European Conference on Machine Learning.
[15] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. 2004. Real-time garbage collection for flash-memory storage systems of
      real-time embedded systems. *ACM Transactions on Embedded Computing Systems* 3, 4 (2004), 837–863.
[16] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. *ACM
      Transactions on Storage* 12, 3 (May 2016), Article 13, 39 pages. DOI : https://doi.org/10.1145/2818376
[17] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory
      based solid state drives in high-speed data processing. In *Proceedings of the IEEE 17th International Symposium on High
      Performance Computer Architecture.* IEEE Computer Society, 266–277.
[18] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and
      Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM
      SIGMOD Record* 39, 3 (2011), 5–10.
[19] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly Media, Inc..
[20] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachhara-
      jani, and Feng Wang. 2015. Purity: Building fast, highly-available enterprise flash storage from commodity compo-
      nents. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* ACM, 1683–1694.
[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud
      serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing.* ACM, 143–154.
[22] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Communication of the ACM* 56, 2 (Feb. 2013), 74–80.
      DOI : https://doi.org/10.1145/2408776.2408794
[23] Peter Desnoyers. 2012. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International
      Systems and Storage Conference.* ACM, 12.
[24] Peter Desnoyers. 2014. Analytic models of SSD write performance. *ACM Transactions on Storage* 10, 2 (2014), 8.
[25] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tirat Patana-anake, and Haryadi S. Gunawi. 2013.
      Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Sym-
      posium on Cloud Computing.* ACM, 14.
[26] Y. Du, Y. Zhang, N. Xiao, and F. Liu. 2014. CD-RAIS: Constrained dynamic striping in redundant array of independent
      SSDs. In *Proceedings of the 2014 IEEE International Conference on Cluster Computing.* 212–220.
[27] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. 2015. Reducing latency via
      redundant requests: Exact analysis. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement
      and Modeling of Computer Systems.* ACM, 347–360. DOI : https://doi.org/10.1145/2745844.2745873
[28] Javier González and Matias Bjørling. 2017. Multi-tenant I/O isolation with open-channel SSDs. In *Proceedings of the
      Nonvolatile Memory Workshop.*
[29] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. 2011. Leveraging value locality
      in optimizing NAND flash-based SSDs. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies.*
      91–103.
[30] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi.
      2016. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *Proceedings of the 14th
      USENIX Conference on File and Storage Technologies.* USENIX Association, 263–276.

[31] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Reducing file system tail latencies with chopper. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, Vol. 15. 119–133.

[32] Windsor W. Hsu and Alan Jay Smith. 2003. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal* 42, 2 (2003), 347–372.

[33] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 10.

[34] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX Association, 2–2.

[35] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. 375–390.

[36] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proceedings of the 2010 IEEE 26th International Conference on Data Engineering Workshops*. IEEE, 41–51.

[37] Soojun Im and Dongkun Shin. 2011. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Transactions on Computers* 60, 1 (2011), 80–92.

[38] Micron Technology Inc. 2011. NAND Flash Media Management Through RAIN. Retrieved 01 Jan., 2021 from https://www.micron.com/~/media/documents/products/technical-marketing-brief/brief_ssd_rain.pdf.

[39] Intel Corp. 2015. Intel Solid-State Drive DC P3608 Series. Retrieved 01 Jan., 2021 from http://www.intel.com/content/www/us/en/solid-state-drives/ssd-dc-p3608-spec.html.

[40] Jens Axboe. 2015. FIO: Flexible I/O Tester. Retrieved 01 Jan., 2021 from https://github.com/axboe/fio.

[41] Myoungsoo Jung and Mahmut Kandemir. 2012. An evaluation of different page allocation strategies on high-speed SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX.

[42] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*.

[43] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production windows servers. In *Proceedings of the IEEE International Symposium on Workload Characterization*. IEEE, 119–128.

[44] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO complying SSDs through OPS isolation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 183–189.

[45] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2013. Improving SSD reliability with RAID via elastic striping and anywhere parity. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 1–12.

[46] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote flash == local flash. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*.

[47] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 273–286.

[48] Jongsung Lee and Jin-Soo Kim. 2013. An empirical study of hot/cold data separation policies in solid state drives. In *Proceedings of the 6th International Systems and Storage Conference*. ACM, 12.

[49] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. 2011. A semi-preemptive garbage collector for solid state drives. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 12–21.

[50] Sehwan Lee, Bitna Lee, Kern Koh, and Hyokyung Bahn. 2011. A lifespan-aware reliability scheme for RAID-based flash storage. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 374–379.

[51] Yangsup Lee, Sanghyuk Jung, and Yong Ho Song. 2009. FRA: A flash-aware redundancy array of flash storage devices. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis*. ACM, 163–172.

[52] Seagate Technology LLC. 2017. SSD Over-provisioning and Its Benefits. Retrieved 01 Jan., 2021 from http://www.seagate.com/tech-insights/ssd-over-provisioning-benefits-master-ti/.

[53] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*. USENIX, 257–270. Retrieved from https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou.

[54] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. 12.

[55] NVM Express Inc. 2015. NVM Express: The Optimized PCI Express SSD Interface. Retrieved 01 Jan., 2021 from http://www.nvmexpress.org.

[56] Jian Ouyang, Shiding Lin, Jiang Song, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. 471–484. DOI : https://doi.org/10.1145/2541940.2541959

[57] Stan Park and Kai Shen. 2012. FIOS: A fair, efficient flash I/O scheduler. In *Proceedings of the USENIX File and Storage Technologies*. USENIX Association, 13. Retrieved from http://dl.acm.org/citation.cfm?id=2208461.2208474

[58] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A case for redundant arrays of inexpensive disks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 109–116. DOI : https://doi.org/10.1145/50202.50214

[59] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems* 33, 4 (2015), 11.

[60] Manju Rajashekhar and Yao Yue. [n.d.]. Fatchache. Retrieved 06 Jan., 2021 from https://github.com/twitter/fatcache.

[61] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 401–417.

[62] Alma Riska and Erik Riedel. 2006. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference, General Track*, Vol. 2006. 97–102.

[63] Alma Riska and Erik Riedel. 2009. Evaluation of disk-level workloads at different time-scales. In *IEEE International Symposium on Workload Characterization*. IEEE, 158–167.

[64] Samsung Electronics Co. 2015. Samsung PM1725 NVMe PCIe SSD. Retrieved 01 Jan., 2021 from http://www.samsung.com/semiconductor/global/file/insight/2015/11/pm1725-ProdOverview-2015-0.pdf.

[65] Omar Sandoval. [n.d.]. Kyber Multi-queue i/o Scheduler. Retrieved from https://lwn.net/Articles/720071/.

[66] Kai Shen and Stan Park. 2013. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 67–78. Retrieved from https://www.usenix.org/conference/atc13/technical-sessions/presentation/shen.

[67] Woong Shin, Myeongcheol Kim, Kyudong Kim, and Heon Y. Yeom. 2015. Providing QoS through host controlled flash SSD garbage collection and multiple SSDs. In *2015 International Conference on Big Data and Smart Computing*. IEEE, 111–117.

[68] David Shue and Michael J. Freedman. 2014. From application requests to virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the European Conference on Computer Systems*. 17:1–17:14. DOI : https://doi.org/10.1145/2592798.2592823

[69] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the IEEE Mass Storage Systems and Technologies*. IEEE Computer Society, 1–10. DOI : https://doi.org/10.1109/MSST.2010.5496972

[70] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. 2014. Flash on rails: Consistent flash performance through redundancy. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 463–474.

[71] Kent Smith. 2013. Understanding SSD over-provisioning. *EDN Network* (2013). Retrieved on 01 Jan., 2020 from https://www.vlsiresearch.com/understanding-ssd-over-provisioning-PFEL1SE1QFXVBZX#page1.

[72] Radu Stoica and Anastasia Ailamaki. 2013. Improving flash write performance by using update frequency. *Proceedings of the VLDB Endowment* 6, 9 (2013), 733–744.

[73] Hui Sun, Xiao Qin, Fei Wu, and Changsheng Xie. 2013. Measuring and analyzing write amplification characteristics of solid state disks. In *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 212–221.

[74] Arash Tavakkol, Pooyan Mehrvarzy, Mohammad Arjomand, and Hamid Sarbazi-Azad. 2016. Performance evaluation of dynamic page allocation strategies in SSDs. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 1, 2 (June 2016), Article 7, 33 pages. DOI : https://doi.org/10.1145/2829974

[75] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. 2013. Low latency via redundancy. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 283–294.

[76] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the European Conference on Computer Systems*. ACM, Article 16, 14 pages. DOI : https://doi.org/10.1145/2592798.2592804

[77] Guanying Wu and Xubin He. 2012. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. USENIX Association, 10–10. Retrieved from http://dl.acm.org/citation.cfm?id=2208461.2208471

[78] Suzhen Wu, Yanping Lin, Bo Mao, and Hong Jiang. 2016. GCaR: Garbage collection aware cache management with improved performance for flash-based SSDs. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 28.

[79] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. 2015. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 543–557. Retrieved from https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/wu.

[80] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. 15–28.

[81] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference*. Article 3, 11 pages.

[82] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 87–100.