LightNVM: The Linux Open-Channel SSD Subsystem

Matias Bjørling[†*] Javier González[†] Philippe Bonnet[*]

[†]*CNEX Labs, Inc.* [*]*IT University of Copenhagen*

Abstract

As Solid-State Drives (SSDs) become commonplace in data-centers and storage arrays, there is a growing demand for predictable latency. Traditional SSDs, serving block I/Os, fail to meet this demand. They offer a high-level of abstraction at the cost of unpredictable performance and suboptimal resource utilization. We propose that SSD management trade-offs should be handled through *Open-Channel SSDs*, a new class of SSDs, that give hosts control over their internals. We present our experience building *LightNVM*, the Linux Open-Channel SSD subsystem. We introduce a new Physical Page Address I/O interface that exposes SSD parallelism and storage media characteristics. LightNVM integrates into traditional storage stacks, while also enabling storage engines to take advantage of the new I/O interface. Our experimental results demonstrate that LightNVM has modest host overhead, that it can be tuned to limit read latency variability and that it can be customized to achieve predictable I/O latencies.

1 Introduction

Solid-State Drives (SSDs) are projected to become the dominant form of secondary storage in the coming years [18, 19, 31]. Despite their success due to superior performance, SSDs suffer well-documented shortcomings: log-on-log [37, 57], large tail-latencies [15, 23], unpredictable I/O latency [12, 28, 30], and resource underutilization [1, 11]. These shortcomings are not due to hardware limitations: the non-volatile memory chips at the core of SSDs provide predictable high-performance at the cost of constrained operations and limited endurance/reliability. It is how tens of non-volatile memory chips are managed within an SSD, providing the same block I/O interface as a magnetic disk, which causes these shortcomings [5, 52].

A new class of SSDs, branded as *Open-Channel SSDs*, is emerging on the market. They are an excellent platform for addressing SSD shortcomings and managing trade-offs related to throughput, latency, power consumption, and capacity. Indeed, open-channel SSDs expose their internals and enable a host to control data placement and physical I/O scheduling. With open-channel SSDs, the responsibility of SSD management is shared between host and SSD. Open-channel SSDs have been used by Tier 1 cloud providers for some time. For example, Baidu used open-channel SSDs to streamline the storage stack for a key-value store [55]. Also, Fusion-IO [27] and Violin Memory [54] each implement a host-side storage stack to manage NAND media and provide a block I/O interface. However, in all these cases the integration of open-channel SSDs into the storage infrastructure has been limited to a single point in the design space, with a fixed collection of trade-offs.

Managing SSD design trade-offs could allow users to reconfigure their storage software stack so that it is tuned for applications that expect a block I/O interface (e.g., relational database systems, file systems) or customized for applications that directly leverage open-channel SSDs [55]. There are two concerns here: (1) a block device abstraction implemented on top of open-channel SSDs should provide high performance, and (2) design choices and trade-off opportunities should be clearly identified. These are the issues that we address in this paper. Note that demonstrating the advantages of application-specific SSD management is beyond the scope of this paper.

We describe our experience building LightNVM, the Open-Channel SSD subsystem in the Linux kernel. LightNVM is the first open, generic subsystem for Open-Channel SSDs and host-based SSD management. We make four contributions. First, we describe the characteristics of open-channel SSD management. We identify the constraints linked to exposing SSD internals, discuss the associated trade-offs and lessons learned from the storage industry.

Second, we introduce the *Physical Page Address* (PPA) I/O interface, an interface for Open-Channel SSDs, that defines a hierarchical address space together with control and vectored data commands.

Third, we present LightNVM, the Linux subsystem that we designed and implemented for open-channel SSD management. It provides an interface where application-specific abstractions, denoted as *targets*, can be implemented. We provide a host-based Flash Translation Layer, called *pblk*, that exposes open-channel SSDs as traditional block I/O devices.

Finally, we demonstrate the effectiveness of LightNVM on top of a first generation open-channel SSD. Our results are the first measurements of an open-channel SSD that exposes the physical page address I/O interface. We compare against state-of-the-art block I/O SSD and evaluate performance overheads when running synthetic, file system, and database system-based workloads. Our results show that LightNVM achieves high performance and can be tuned to control I/O latency variability.

2 Open-Channel SSD Management

SSDs are composed of tens of storage chips wired in parallel to a controller via so-called *channels*. With open-channel SSDs, channels and storage chips are exposed to the host. The host is responsible for utilizing SSD resources in time (I/O scheduling) and space (data placement). In this section, we focus on NAND flash-based open-channel SSDs because managing NAND is both relevant and challenging today. We review the constraints imposed by NAND flash, introduce the resulting key challenges for SSD management, discuss the lessons we learned from early adopters of our system, and present different open-channel SSD architectures.

2.1 NAND Flash Characteristics

NAND flash relies on arrays of floating-gate transistors, so-called cells, to store bits. Shrinking transistor size has enabled increased flash capacity. SLC flash stores one bit per cell. MLC and TLC flash store 2 or 3 bits per cell, respectively, and there are four bits per cell in QLC flash. For 3D NAND, increased capacity is no longer tied to shrinking cell size but to flash arrays layering.

Media Architecture. NAND flash provides a read-/write/erase interface. Within a NAND package, storage media is organized into a hierarchy of die, plane, block, and page. A die allows a single I/O command to be executed at a time. There may be one or several dies within a single physical package. A plane allows similar flash commands to be executed in parallel within a die.

Within each plane, NAND is organized in blocks and pages. Each plane contains the same number of blocks, and each block contains the same number of pages. Pages are the minimal units of read and write, while the unit of erase is a block. Each page is further decomposed into fixed-size sectors with an additional out-of-bound area, e.g., a 16KB page contains four sectors of 4KB plus an out-of-bound area frequently used for ECC and user-specific data.

Regarding internal timings, NAND flash memories exhibit an order of magnitude difference between read and write/erase latency. Reads typically take sub-hundred microseconds, while write and erase actions take a few milliseconds. However, read latency spikes if a read is scheduled directly behind a write or an erase operation, leading to orders of magnitude increase in latency.

Write Constraints. There are three fundamental programming constraints that apply to NAND [41]: (i) a write command must always contain enough data to program one (or several) full flash page(s), (ii) writes must be sequential within a block, and (iii) an erase must be performed before a page within a block can be (re)written. The number of program/erase (PE) cycles is limited. The limit depends on the type of flash: 10^2 for TLC/QLC flash, 10^3 for MLC, or 10^5 for SLC.

Additional constraints must be considered for different types of NAND flash. For example, in multi-level cell memories, the bits stored in the same cell belong to different write pages, referred to as lower/upper pages. The upper page must be written before the lower page can be read successfully. The lower and upper page are often not sequential, and any pages in between must be written to prevent write neighbor disturbance [10]. Also, NAND vendors might introduce any type of idiosyncratic constraints, which are not publicly disclosed. This is a clear challenge for the design of cross-vendor, host-based SSD management.

Failure Modes. NAND Flash might fail in various ways [7, 40, 42, 49]:

- Bit Errors. The downside of shrinking cell size is an increase in errors when storing bits. While error rates of 2 bits per KB were common for SLC, this rate has increased four to eight times for MLC.

- Read and Write Disturb. The media is prone to leak currents to nearby cells as bits are written or read. This causes some of the write constraints described above.

- Data Retention. As cells wear out, data retention capability decreases. To persist over time, data must be rewritten multiple times.

- Write/Erase Error. During write or erase, a failure can occur due to an unrecoverable error at the block level. In that case, the block should be retired and data already written should be rewritten to another block.
- Die Failure. A logical unit of storage, i.e., a die on a NAND chip, may cease to function over time due to a defect. In that case, all its data will be lost.

2.2 Managing NAND

Managing the constraints imposed by NAND is a core requirement for any flash-based SSD. With open-channel SSDs, this responsibility is shared between software components running on the host (in our case a Linux device driver and layers built on top of it) and on the device controller. In this section we present two key challenges associated with NAND management: write buffering and error handling.

Write Buffering. Write buffering is necessary when the size of the sector, defined on the host side (in the Linux device driver), is smaller than the NAND flash page size, e.g., a 4KB sector size defined on top of a 16KB flash page. To deal with such a mismatch, the classical solution is to use a cache: sector writes are buffered until enough data is gathered to fill a flash page. If data must be persisted before the cache is filled, e.g., due to an application flush, then padding is added to fill the flash page. Reads are directed to the cache until data is persisted to the media. If the cache resides on the host, then the two advantages are that (1) writes are all generated by the host, thus avoiding interference between the host and devices, and that (2) writes are acknowledged as they hit the cache. The disadvantage is that the contents of the cache might be lost in case of a power failure.

The write cache may also be placed on the device side. Either the host writes sectors to the device and lets the device manage writes to the media (when enough data has been accumulated to fill a flash page), or the host explicitly controls writes to the media and lets the device maintain durability. With the former approach, the device controller might introduce unpredictability into the workload, as it might issue writes that interfere with host-issued reads. With the latter approach, the host has full access to the device-side cache. In NVMe, this can be done through a *Controller Memory Buffer* (CMB) [43]. The host can thus decouple (i) the staging of data on the device-side cache from (ii) the writing to the media through an explicit flush command. This approach avoids controller-generated writes and leaves the host in full control of media operations. Both approaches require that the device firmware has power-fail techniques to store the write buffer onto media in case of a power loss. The size of the cache is then limited by the power-capacitors available on the SSD.

Error Handling. Error handling concerns reads, writes, and erases. A read fails when all methods to recover data at sector level have been exhausted: ECC, threshold tuning, and possibly parity-based protection mechanisms (RAID/RAIN) [13, 20].

To compensate for bit errors, it is necessary to introduce Error Correcting Codes (ECC), e.g., BCH [53] or LDPC [16]. Typically, the unit of ECC encoding is a *sector*, which is usually smaller than a page. ECC parities are generally handled as metadata associated with a page and stored within the page's out-of-band area.

The bit error rate (BER) can be estimated for each block. To maintain BER below a given threshold, some vendors make it possible to tune NAND threshold voltage [7, 8]. Blocks which are write-cold and read-hot, for which BER is higher than a given threshold, should be rewritten [47]. It might also be necessary to perform *read scrubbing*, i.e., schedule read operations for the sole purpose of estimating BER for blocks which are write-cold and read-cold [9].

Given that manual threshold tuning causes several reads to be executed on a page, it may be beneficial to add RAID techniques to recover data faster, while also enable SSD to recover from die failures.

Note that different workloads might require different RAID configurations. Typically, high read workloads require less redundancy, because they issue fewer PE cycles. This is an argument for host-based RAID implementation. Conversely, for high write workloads, RAID is a source of overhead that might be compensated by hardware acceleration (i.e., a hardware-based XOR engine [14, 48]).

In the case of write failures, due to overcharging or inherent failures [51], recovery is necessary at the block level. When a write fails, part of a block might already have been written and should be read to perform recovery. Early NAND flash chips allow reads on partially written blocks, but multi-level NAND [10] requires that a set of pages (lower/upper) be written before data can be read, thus preventing reads of partially written blocks in the general case. Here, enough buffer space should be available to restore the contents of partially written blocks.

If a failure occurs on erase, there is no retry or recovery. The block is simply marked bad.

2.3 Lessons Learned

Open-channel SSDs open up a large design space for SSD management. Here are some restrictions on that design space based on industry trends and feedback from early LightNVM adopters.

1. Provide device warranty with physical access. Warranty to end-users is important in high-volume markets. A traditional SSD is often warrantied for either three or five years of operation. In its lifetime, enough good flash media must be available to perform writes. Contrary to spinning hard-drives, the lifetime for NAND media heavily depends on the number of writes to the media. Therefore, there is typically two types of guarantees for flash-based SSDs: Year warranty and Drive Writes Per Day (DWPD) warranty. DWPD guarantees that the drive can sustain X drive writes per day. Providing low thousands of PE cycles to NAND flash media, the number of writes per day is often limited to less than ten and is lower in consumer drives.

If PE cycles are managed on the host, then no warranty can be given for open-channel SSDs. Indeed, SSD vendors have no way to assess whether a device is legitimately eligible for replacement, or if flash simply wore out because of excessive usage. To provide warranty, PE cycles must be managed on the device. See Figure 1 for an illustration.

2. Exposing media characterization to the host is inefficient and limits media abstraction. Traditional SSD vendors perform media characterization with NAND vendors to adapt their embedded Flash Translation Layer to the characteristics of a given NAND chip. Such in-house NAND characterization is protected under IP. It is neither desirable nor feasible to let application and system developers struggle with the internal details of a specific NAND chip, in particular threshold tuning or ECC. These must be managed on the device. This greatly simplifies the logic in the host and lets the open-channel SSD vendor differentiate their controller implementation.

3. Write buffering should be handled on the host or the device depending on the use case. If the host handles write buffering, then there is no need for DRAM on the device, as the small data structures needed to maintain warranty and physical media information can be stored in device SRAM or persistent media if necessary. Power consumption can thus be drastically reduced. Managing the write buffer on the device, through a CMB, efficiently supports small writes but requires extra device-side logic, together with power-capacitors or similar functionality to guarantee durability. Both options should be available to open-channel SSD vendors.

4. Application-agnostic wear leveling is mandatory. As NAND ages, its access time becomes longer. Indeed, the voltage thresholds become wider, and more time must be spent to finely tune the appropriate voltage to read or write data. NAND specifications usually report both a typical access latency and a max latency. To make sure that latency does not fluctuate depending on the age of the block accessed, it is mandatory to perform wear-leveling independently from the application workload, even if it introduces an overhead.

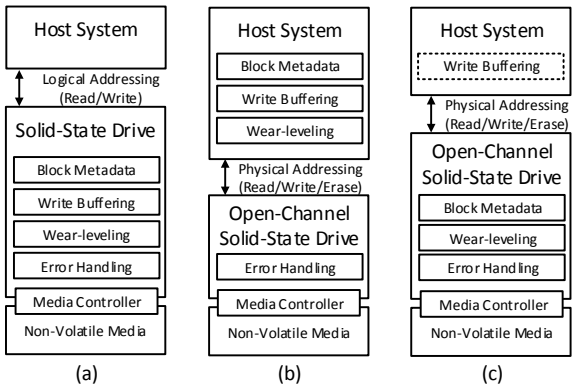

Figure 1: Core SSD Management modules on (a) a traditional Block I/O SSD, (b) the class of open-channel SSD considered in this paper, and (c) future open-channel SSDs.

It must be possible, either for the host or the controller, to pick free blocks from a die in a way that (i) hides bad blocks, (ii) implements dynamic wear leveling by taking the P/E cycle count into account when allocating a block, and possibly (iii) implements static wear leveling by copying cold data to a hot block. Such decisions should be based on metadata collected and maintained on the device: P/E cycle per block, read counts per page, and bad blocks. Managing block metadata and a level of indirection between logical and physical block addresses incurs a significant overhead regarding latency and might generate internal I/Os (to store the mapping table or due to static wear leveling) that might interfere with an application I/Os. This is the cost of wear-leveling [6].

2.4 Architectures

Different classes of open-channel SSDs can be defined based on how the responsibilities of SSD management are shared between host and SSD. Figure 1 compares (a) traditional block I/O SSD with (b) the class of open-channel SSDs considered in this paper, where PE cycles and write buffering are managed on the host, and (c) future open-channel SSDs that will provide warranties and thus support PE cycle management and wear-leveling on the device. The definition of the PPA I/O interface and the architecture of LightNVM encompass all types of open-channel SSDs.

3 Physical Page Address I/O Interface

We propose an interface for open-channel SSDs, the *Physical Page Address (PPA) I/O interface*, based on a hierarchical address space. It defines administration commands to expose the device geometry and let the host

take control of SSD management, and data commands to efficiently store and retrieve data. The interface is independent of the type of non-volatile media chip embedded on the open-channel SSD.

Our interface is implemented as a vendor-specific extension to the NVM Express 1.2.1 specification [43], a standard that defines an optimized interface for PCIe-attached SSDs.

3.1 Address Space

We rely on two invariants to define the PPA address space:

1. *SSD Architecture*. Open-channel SSDs expose to the host a collection of channels, each containing a set of *Parallel Units* (PUs), also known as LUNs. We define a PU as the unit of parallelism on the device. A PU may cover one or more physical die, and a die may only be a member of one PU. Each PU processes a single I/O request at a time.

2. *Media Architecture*. Regardless of the media, storage space is quantized on each PU. NAND flash chips are decomposed into blocks, pages (the minimum unit of transfer), and sectors (the minimum unit of ECC). Byte-addressable memories may be organized as a flat space of sectors.

The controller can choose the physical representation for the PUs. This way the controller can expose a performance model, at the PU level, that reflects the performance of the underlying storage media. If the controller chooses a logical definition for PUs (e.g., several NAND dies accessed through RAID) then the performance model for a PU must be constructed based on storage media characteristics and controller functionality (e.g., XOR engine acceleration). A logical representation might be beneficial for byte-addressable memories, where multiple dies are grouped together to form a single sector. In the rest of this paper, we assume that a PU corresponds to a single physical NAND die. With such a physical PU definition, the controller exposes a simple, well-understood, performance model of the media.

PPAs are organized as a decomposition hierarchy that reflects the SSD and media architecture. For example, NAND flash may be organized as a hierarchy of plane, block, page, and sector, while byte-addressable memories, such as PCM, is a collection of sectors. While the components of the SSD architecture, channel and PU, are present in all PPA addresses, media architecture components can be abstracted. This point is illustrated in Figure 2.

Each device defines its bit array nomenclature for PPA addresses, within the context of a 64-bit address. Put differently, the PPA format does not put constraints on the maximum number of channels per device or the maximum number of blocks per PU. It is up to each device to define these limitations, and possibly ignore some media-specific components. This flexibility is a major difference between the PPA format and the hierarchical CHS (Cylinder-Head-Sector) format introduced for early hard drives.

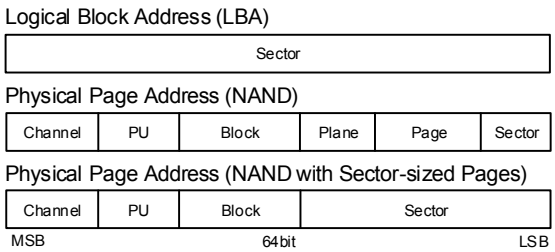

Figure 2: Logical Block Addresses compared to Physical Page Addresses for NAND flash.

Encoding the SSD and media architecture into physical addresses makes it possible to define hardware units, embedded on an open-channel SSD, that map incoming I/Os to their appropriate physical placement. The PPA format is also well-suited for a representation where the identifiers are independent variables as a power of two. This way, operations on the name-space (e.g., get next page on the next channel, or get next page on a different PU) are efficiently implemented by shifting bits.

The PPA address space can be organized logically to act as a traditional logical block address (LBA), e.g., by arranging NAND flash using "block, page, plane, and sector" [34]. This enables the PPA address space to be exposed through traditional read/write/trim commands. In contrast to traditional block I/O, the I/Os must follow certain rules. Writes must be issued sequentially within a block. Trim may be issued for a whole block, so that the device interprets the command as an erase. It is implementation specific whether a single read may cross multiple blocks in a single I/O.

In comparison with a traditional, linear LBA space, the PPA address space may contain invalid addresses, where I/Os are not accepted. Consider for example that there are 1067 available blocks per PU, then it would be represented by 11 bits. Blocks 0–1066 would be valid, while blocks 1067–2047 would be invalid. It is up to the controller to return an error in this case. In case the media configuration for each level in the hierarchy is not a power of two, then there will be such holes in the address space.

3.2 Geometry and Management

To let a host take control of SSD management, an open-channel SSD must expose four characteristics:

1. *Its geometry*, i.e., the dimensions of the PPA address space. How many channels? How many PUs within a channel? How many planes per PU? How many blocks per plane? How many pages per block? How many sectors per page? How large is the out-of-bound region per page? We assume that PPA dimensions are uniform for a given address space. If an SSD contains different types of storage chips, then the SSD must expose the storage as separate address spaces, each based on similar chips.

2. *Its performance*, i.e., statistics that capture the performance of data commands, channel capacity and controller overhead. The current version of the specification captures typical and max latency for page read, page write, and erase commands and the maximum number of in-flight commands addressed to separate PUs within a channel.

3. *Media-specific metadata*. For instance, NAND flash-specific metadata includes the type of NAND flash on the device, whether multi-plane operations are supported, the size of the user-accessible out-of-bound area, or page pairing information for MLC and TLC chips. As media evolves, and becomes more complex, it may be advantageous to let SSDs handle this complexity.

4. *Controller functionalities*. As we have seen in Section 2, a controller might support write buffering, failure handling, or provisioning. Each of these capabilities might be configured (e.g., RAID across PUs). If the controller supports write buffering, then a flush command enables the host to force the controller to write the contents of its buffer to the storage media.

3.3 Read/Write/Erase

The data commands directly reflect the read, write, and erase interface of NAND flash cells. The erase command is ignored for media that does not support it.

Vectored I/Os. Data commands expand upon traditional LBA access. A read or write command is no longer defined by a start LBA, some sectors to access, and a data buffer. Instead, a read or write is applied to a vector of addresses to leverage the intrinsic parallelism of the SSD. For example, let us consider 64KB of application data. Assuming a page size of 4KB, this data might be striped with a write command applied to 16 sectors simultaneously, thus efficiently supporting scattered access.

Concretely, each I/O is represented as an NVMe I/O read/write command. We replace the start LBA (SLBA) field with a single PPA address or a pointer to an address list, denoted *PPA list*. The PPA list contains an LBA for each sector to be accessed. Similarly, we utilize the NVMe I/O metadata field to carry out-of-band metadata. The metadata field is typically used for end-to-end data consistency (T10-PI/DIF/DIX [25, 43]). How to gracefully combine end-to-end and PPA metadata is a topic for future work.

When a data command completes, the PPA interface returns a separate completion status for each address. This way, the host can distinguish and recover from failures at different addresses. For the first iteration of the specification, the first 64 bits of the NVMe I/O command completion entry are used to signal the completion status. This limits the number of addresses in the PPA list to 64.

We considered alternatives to the PPA list. In fact, we evaluated three approaches: (i) NVMe I/O command, (ii) grouped I/Os, and (iii) Vectored I/Os. An NVMe I/O command issues commands serially. When a full page buffer is constituted, it is flushed to the media. Each command rings the doorbell of the controller to notify a new submission. With grouped I/Os, several pages constitute a submission, the doorbell is only rung once, but it is up to the controller to maintain the state of each submission. With vectored I/Os, an extra DMA is required to communicate the PPA list. We opt for the third option, as the cost of an extra DMA mapping is compensated by simplified controller design.

Media specific. Each I/O command provides media-specific hints, including plane operation mode (single, dual, or quad plane), erase/program suspend [56], and limited retry. The plane operation mode defines how many planes should be programmed at once. The controller may use the plane operation mode hint to efficiently program planes in parallel, as it accesses PUs sequentially by default. Similarly, the erase-suspend allows reads to suspend an active write or program, and thus improve its access latency, at the cost of longer write and erase time. Limited retry allows the host to let the controller know that it should not exhaust all options to read or write data, but instead fail fast to provide a better quality of service, if data is already available elsewhere.

4 LightNVM

LightNVM is the open-channel SSD subsystem in Linux. In this section, we give an overview of its architecture, and we present the pblk target in detail.

4.1 Architecture

LightNVM is organized in three layers (see Figure 3), each providing a level of abstraction for open-channel SSDs:

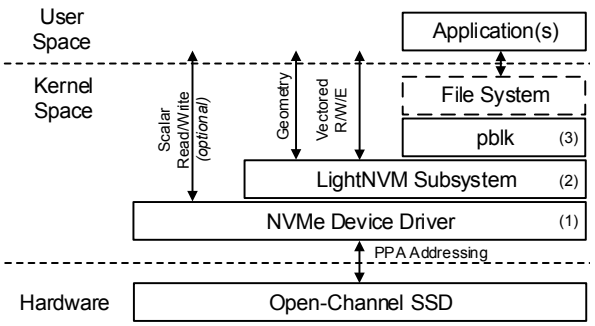

Figure 3: LightNVM Subsystem Architecture

1. **NVMe Device Driver.** A LightNVM-enabled NVMe device driver gives kernel modules access to open-channel SSDs through the PPA I/O interface. The device driver exposes the device as a traditional Linux device to user-space, which allows applications to interact with the device through ioctls. If the PPA interface is exposed through an LBA, it may also issue I/Os accordingly.

2. **LightNVM Subsystem.** An instance of the subsystem is initialized on top of the PPA I/O-supported block device. The instance enables the kernel to expose the geometry of the device through both an internal *nvm_dev* data structure and sysfs. This way FTLs and user-space applications can understand the device geometry before use. It also exposes the vector interface using the blk-mq [4] device driver private I/O interface, enabling vector I/Os to be efficiently issued through the device driver.

3. **High-level I/O Interface.** A *target* gives kernel-space modules or user-space applications access to open-channel SSDs through a high-level I/O interface, either a standard interface like the block I/O interface provided by pblk (see Section 4.2), or an application-specific interface provided by a custom target.

4.2 *pblk*: Physical Block Device

The *Physical Block Device* (pblk) is a LightNVM target implementing a fully associative, host-based FTL that exposes a traditional block I/O interface. In essence, pblk's main responsibilities are to (i) deal with controller- and media-specific constraints (e.g., caching the necessary amount of data to program a flash page), (ii) map logical addresses onto physical addresses (4KB granularity) and guarantee the integrity—and eventual recovery in the face of crashes—of the associated mapping table (L2P), (iii) handle errors, and (iv) implement garbage collection (GC). Since typical flash page sizes are bigger than 4KB, pblk must also (v) handle *flushes*. A flush forces pblk's in-flight data to be stored on the device before it completes. It might be required by a file system or an application (i.e., fsync).

4.2.1 Write Buffering

The pblk target is based on the architecture described in Section 2.4, where write buffering is managed on the host. The write buffer is managed as a circular ring buffer. It is internally decoupled into two buffers: a data buffer storing 4KB user data entries (4KB corresponds to the size of a sector), and a context buffer storing per-entry metadata. The size of the buffer is the product of flash page size (FPSZ), the number of flash pages to write (lower/upper pages), and the number of PUs (N). For example, if $FPSZ = 64KB, PP = 8, N = 128$, the write buffer is 64MB.

The write buffer is accessed by several producers and a single consumer:

Producers. Both pblk users and pblk's own garbage collector insert I/Os as entries into the write buffer. When a new entry is written, the L2P table is updated with the entry line and the write is acknowledged. If the buffer is full, the write is re-scheduled. In case that a mapping already exists for the incoming logical address, the old entry is invalidated.

Consumer. A single thread consumes buffered entries either when there is enough data to fill a flash page or when a flush command has been issued. If multi-plane programming is used then the number of planes must also be considered (e.g., 16KB pages with quad plane programming requires 64KB chunks for a single write). At this point, logical addresses are mapped to physical ones. By default, pblk's mapping strategy targets throughput and stripes data across channels and PUs at a page granularity. Other data placement strategies can be used. After mapping takes place, a vector write command is formed and sent to the device. Note that in case of a flush, if there is not enough data to fill a flash page, pblk adds padding (i.e., unmapped data) in the write command before it is sent to the device.

In order to respect the lower/upper page pairs (Section 2.1), the L2P table is not modified as pages are mapped. This way, reads are directed to the write buffer until all page pairs have been persisted. When this happens, the L2P table is updated with the physical address. L2P recovery is discussed in Section 4.2.2.

The number of channels and PUs used for mapping incoming I/Os can be tuned at run-time. We refer to them as active PUs. For example, let us consider 4 active PUs on an open-channel SSD with 4 channels and 8 PUs per channel. To start with, PU_0, PU_8, PU_{16}, and PU_{24} are

active. Pages are written on those PUs in a round-robin fashion. When a block fills up on PU_0, then that PU becomes inactive and PU_1 takes over as the active PU. At any point in time, only 4 PUs are active, but data is still striped across all available PUs at a page granularity.

When an application or file system issues a flush, pblk ensures that all outstanding data is written to the media. The consumer thread empties the write buffer and uses padding to fill up the last flash page if necessary. As data is persisted, the last write command holds an extra annotation that indicates that it must complete before the flush is successful.

4.2.2 Mapping Table Recovery

The L2P mapping table is essential for data consistency in a block device. Thus, we persist a redundant version of the mapping table in three forms: First, as a snapshot, which is stored (i) on power-down, in form of a full copy of the L2P, and (ii) periodically as checkpoints in form of an FTL log that persists operations on blocks (allocate and erase). Second, as block-level metadata, on the first and last page of each block. When a block is opened, the first page is used to store a block sequence number together with a reference to the previous block. When a block is fully written, the last pages are used to store (1) an FTL-log consisting of the portion of the L2P map table corresponding to data in the block, (2) the same sequence number as in the first page in order to avoid an extra read during recovery, and (3) a pointer to the next block. The number of pages needed depends on the size of the block. This strategy allows us to recover the FTL in an ordered manner and prevent old mappings from overwriting new ones. Finally, a portion of the mapping table is kept (iii) within the OOB area of each flash page that is written to the device. Here, for each persisted flash page, we store the logical addresses that correspond to physical addresses on the page together with a bit that signals that the page is valid. Since blocks can be reused as they are garbage collected, all metadata is persisted together with its CRC and relevant counters to guarantee consistency.

Any initialization of the device will trigger a full recovery. If an L2P mapping table snapshot is available (e.g., due to a graceful shutdown), then the mapping table is directly retrieved from disk and loaded into memory. In the case of a non-graceful shutdown, the mapping table must be recovered. We designed a two-phase recovery process.

To start with, we scan the last page of all available blocks and we classify them into free, partially written, and fully written. We can reduce the scanning by looking at the sequence numbers and only recovering written blocks. In the first phase, fully written blocks are ordered using the sequence number. The L2P table is then updated with the map portions stored on each last page. Similarly, in the second phase, partially written blocks are ordered. After this, blocks are scanned linearly until a page with an invalid bit on the OOB area is reached. Each valid mapping triggers an update in the L2P table. To ensure data correctness, it is paramount that half-written lower/upper pages are padded before reads can be issued. If the controller counts on a super capacitor, padding can be done in the device on ungraceful power-down. Otherwise, padding must be implemented on the second phase of recovery, as partially written blocks are recovered.

4.2.3 Error Handling

Unlike a traditional FTL that deals with read, write, and erase failures, pblk deals only with write and erase errors. As discussed in Section 2.2, ECC and threshold tuning are enforced by the device. If a read fails, then data is irrecoverable from the device's perspective; recovery must be managed by the upper layers of the system, above pblk.

When a write fails, pblk initiates two recovery mechanisms. First, the blocks corresponding to sectors on which a write failed are identified using the per-sector completion bits encoded in the command completion entry. These failed sectors are remapped and re-submitted to the device directly. They are not inserted in the write buffer because of the flush guarantee provided by pblk. In case a flush is attached to the failed command, subsequent writes will stop until the pointed I/O completes. Writes preceding that flush must be persisted before forward progress can be made. The second mechanism starts when the block corresponding to the failed sectors is marked as bad. Here, the remaining pages are padded and the block is sent for GC.

In the case of erase failures, the block is directly marked as bad. Since no writes have been issued at this point, there is no data to recover.

4.2.4 Garbage Collection

As any log-structured FTL, pblk must implement garbage collection. Blocks are re-purposed by garbage collecting any valid pages and returning blocks for new writes. Wear-leveling is assumed to happen either on the device or within the LightNVM core (Section 2.3). Therefore, pblk simply maintains a valid page count for each block, and selects the block with the lowest number of valid sectors for recycling.

The reverse logical to physical mapping table is not stored in host memory. To find a reverse mapping, we leverage the fact that a block is first recycled when it

is fully written. Thus, we can use the partial L2P table stored for recovery on the last pages of the block. In case a page in that block is still valid, it is queued for rewrite. When all pages have been safely rewritten, the original block is recycled.

To prevent user I/Os from interfering with garbage collection, pblk implements a PID controlled [44] rate-limiter, whose feedback loop is based on the total number of free blocks available. When the number of free blocks goes under a configurable threshold, GC starts. Note that GC can also be managed from *sysfs*. In the beginning, both GC and user I/Os compete equally for the write buffer. But as the number of available blocks decreases, GC is prioritized in order to guarantee the consistency of already persisted data. The feedback loop ensures that incoming I/Os and GC I/Os move towards a steady state, where enough garbage collection is applied given the user I/O workload. The rate-limiter uses write buffer entries as a natural way to control incoming I/Os; entries are reserved as a function of the feedback loop. If the device reaches its capacity, user I/Os will be completely disabled until enough free blocks are available.

5 Experimental Evaluation

The purpose of our experimental evaluation is threefold. First, we verify the correctness of the LightNVM stack, and we evaluate the overhead it introduces. Second, we characterize pblk on top of a first generation open-channel SSD (OCSSD) and compare it to a state-of-the-art NVMe SSD in terms of throughput, latency, and CPU utilization. We rely on *fio* [2] and application workloads for this study. Finally, we show how explicit PU write provisioning can be used to optimize I/O scheduling and achieve predictable latencies.

Our experimental setup consists of a server equipped with an Intel Xeon E5-2620v3, 32 GB of DDR4 RAM, an Open-Channel SSD (CNEX Labs Westlake SDK) with 2TB NAND MLC Flash, denoted OCSSD in the rest of this section, and an NVMe SSD (Intel P3700) with 400GB storage, denoted NVMe SSD. Both SSDs are datacenter/enterprise SSDs using MLC NAND, which makes them comparable in terms of hardware raw performance. A new instance of pblk is used for each run on the OCSSD; the NVMe SSD is formatted to 4K sector size and is low-level formatted before each run. The host runs Ubuntu 15.04 with Linux Kernel 4.8-rc4 and pblk patches applied.

The entire LightNVM stack amounts to approximately 10K LOC; pblk is responsible for almost 70% of that code.

Open-Channel Solid-State Drive	
Controller	CNEX Labs Westlake ASIC
Interface	NVMe, PCI-e Gen3x8
Channels	16
PUs per Channel	8 (128 total)
Channel Data Bandwidth	280MB/s
Parallel Unit Characteristics	
Page Size	16K + 64B user OOB
Planes	4
Blocks	1,067
Block Size	256 Pages
Type	MLC
Bandwidths	
Single Seq. PU Write	47MB/s
Single Seq. PU Read	105MB/s (4K), 280MB/s (64KB)
Single Rnd. PU Read	56MB/s (4K), 273MB/s (64KB)
Max Write	4GB/s
Max Read	4.5GB/s
pblk Factory Write (no GC)	4GB/s
pblk Steady Write (GC)	3.2GB/s

Table 1: Solid-State Drive Characterization.

5.1 Sanity Check

Table 1 contains a general characterization for the evaluated Open-Channel SSD. Per-PU sequential read and write bandwidth were gathered experimentally through a modified version [3] of fio that uses the PPA I/O interface and issues vector I/Os directly to the device. The pblk factory state and steady state (where garbage collection is active) are measured experimentally through standard fio on top of pblk. Note that we leave the detailed characterization of pblk for future work and only prove that the implementation works as expected. Unless specified otherwise, each experiment is conducted in factory state with pblk's rate-limiter disabled.

In terms of CPU utilization, pblk introduces an overhead of less than 1% CPU overhead for reads with 0.4μs additional latency (2.32μs with, and 1.97μs without, a difference of 18%). While for writes, it adds 4% CPU overhead with an additional 0.9μs latency (2.9μs with, and 2μs without, a difference of 45%). Overhead on the read path is due to an extra lookup into the L2P table and the overhead on the write path is due to buffer and device write I/O requests management. CPU overhead is measured by comparing the time it takes with and without pblk on top of a null block device [4] and does not include device I/O timings.

5.2 Uniform Workloads

Figure 4 captures throughput and latency for sequential and random reads issued with fio on 100GB of data. The preparation for the test has been performed with pblk using the full bandwidth of the device (128 PUs). This means that sequential reads are more easily parallelized internally by the controller since sequential logical addresses are physically striped across channels and PUs on a per-page basis.

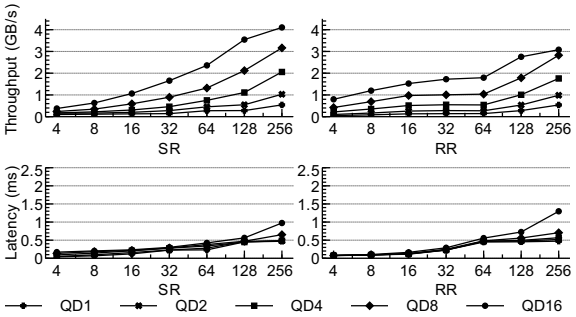

Figure 4: Throughput and corresponding latencies for sequential and random read workloads as a function of queue depths (QD) and block sizes (x axis in KB).

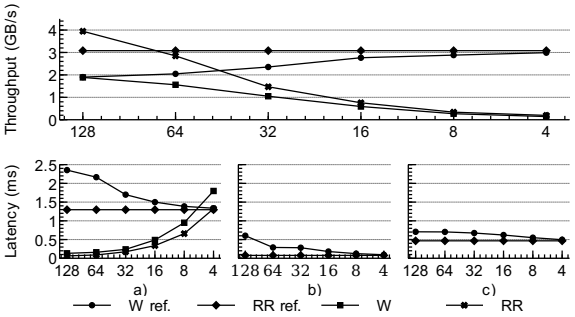

Figure 5: R/W throughput and latencies as a function of active write PUs configurations. Top graph, and a: throughput and corresponding latency (Writes: 256KB, QD1; Reads: 256KB, QD16); b: read latency (Writes: 256KB, QD1; Reads: 4KB, QD1); c: read latency (Write: 256KB, QD1, rate-limited at 200MB/s; Reads: 256KB, QD1).

We see that the OCSSD is capable of reaching 4GB/s using sequential reads at an average latency of 970μs using 256KB request size and a queue depth of 16. The 99^{th} percentile latency is reported at $1,200\mu$s. Similarly, we measure throughput and latency for 4KB reads using a queue depth of 1. Maximum throughput is 105MB/s, with 40μs average access latency and 99^{th} percentile at 400μs. The average access latency is lower than a single flash page access because the controller caches the flash page internally. Thus, all sectors located on the same flash page will be served from the controller buffer instead of issuing a new flash page read. Also, read throughput is limited by the flash page access time, as we only perform one read at a time.

Pure read and write workloads can be used to calibrate queue depths to reach full bandwidth. They show the optimal case, where reads and writes do not block each other. Let us now discuss mixed workloads, which are much more challenging for SSDs.

5.3 Mixed Workloads

For a mixed workload, we use the same write preparation as in the previous experiment (stripe across all PUs with a 100GB dataset).

Then, we proceed to write with an offset of 100GB, while we read from the first 100GB. We repeat this experiment, varying stripe size (number of active write PUs) for new writes. The hypothesis is that as the stripe size decreases, read performance predictability should increase as the probability of a read being stuck behind a write lowers.

Figure 5 depicts the behavior of pblk when reads and writes are mixed. In Figure 5(a), we show throughput for both writes and random reads together with their reference value, represented by 100% writes (4GB/s–200MB/s) and 100% random reads (3GB/s), respectively; Figure 5(b) depicts its latencies. The experiment consists of large sequential 256KB writes at queue depth 1, and 256KB random reads at queue depth 16. The write queue depth is 1, as it is enough to satisfy the full write bandwidth (defined by the capacity of the pblk write buffer). Note that reads are issued at queue depth 16 so that enough parallelism can be leveraged by the controller. This allows us to better visualize the worst-case latencies and the effect of fewer writing PUs.

We observe that when new writes are striped across all 128 PUs, throughput is halved for both reads and writes compared to the reference value, while average latency doubles for reads (maximum latency is increased by $4\times$). Write latencies are close to not being affected because they are buffered. This represents the typical case on a traditional SSD: reads are stacked behind writes, thus affecting read performance; host and controller queues are filled with read requests, thus affecting write performance. However, as soon as we start limiting the number of active write PUs, we observe how reads rapidly recover. For this experiment, we configured one block to be fully written on an active PU before switching to a different PU. Writes are still striped across all 128 PUs, but instead of being striped at page granularity, they are striped at block granularity. This lowers the probability of reads being issued to the same PU as new writes (because, reads and writes are striped at different granularities). If we lower the number of write-active PUs to 4, we see that reads are very close to the reference read workload, while still writing at 200MB/s.

Figure 5(c) shows latency for 4K reads at queue depth 1. Here, we emphasize the impact of a read being blocked by a write. As in Figure 5(b), latency variance reduces as we decrease the number of active write PUs. With 4 active write PUs, the maximum latency for random reads in the 99^{th} percentile is only 2μs higher than in the average case.

Figure 5(d) shows the same experiment as in a) and b), with the difference that writes are rate-limited to

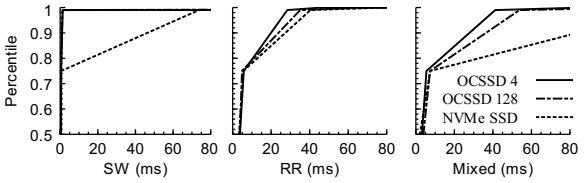

Figure 6: Latencies for RocksDB sequential writes, random reads and mixed workloads on OCSSD and NVMe SSD

	NVMe SSD	OCSSD 128	OCSSD 4
SW	276	396	80
RR	5064	5819	5319
Mixed	2208	3897	4825

Table 2: Throughput (MB/s) for RocksDB sequential writes, random reads, and mixed workloads on OCSSD and NVMe SSD

200MB/s. The motivation for this experiment is the expectation of consistent writes in next-generation SSDs. Note that current SSDs already define the maximal sustained write bandwidths over a three-year period. Examples are write-heavy (e.g., Intel DC P3608, 1.6GB, 5 DWPD) and read-heavy (e.g., Samsung 960 Pro, 2TB, 2.2 DWPD) SSDs, where the limits are 95MB/s and 26MB/s, respectively. The interesting output of this experiment is that even when writes are rated, the variance of reads is still very much affected by the number of active write PUs.

More generally, the experiments with mixed workloads show that informed decisions based on the actual workload of an application can be leveraged to optimize a traditional block device interface, without requiring an application-specific FTL.

5.4 Application Workloads

We evaluate pblk with a NoSQL database, and MySQL with both OLTP and OLAP workloads. The NoSQL database relies on an LSM-tree for storage and leans towards fewer flushes (sync is enabled to guarantee data integrity), while MySQL has tight bounds on persisting transactional data to disk. We evaluate both using the NVMe SSD and the OCSSD using 128 and 4 active write PUs.

NoSQL. For this experiment, we ran RocksDB [17] on top of an Ext4 file system and made use of RocksDB's *db_bench* to execute three workloads: sequential writes, random reads, and mixed (RocksDB *read-while-writing* test). Figure 6 shows user throughput and latencies for the three workloads. We show latency for the 95^{th}, 99^{th} and 99.9^{th} percentile of the latency distribution. Note that internally RocksDB performs its own garbage collection (i.e., sstable compaction). This consumes device bandwidth, which is not reported by db_bench.

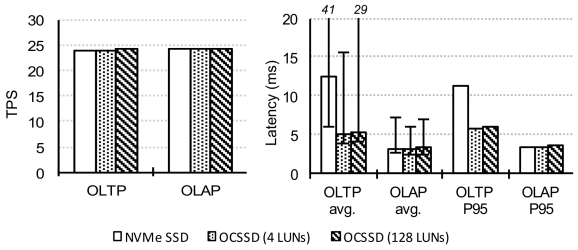

Figure 7: Transactions per second and latencies for OLTP and OLAP on NVMe SSD and OCSSD.

The user write throughput is 276MB/s for the NVMe SSD, 396MB/s for the OCSSD with 128 active PUs, and 88MB/s with 4 active PUs. The fewer active PUs clearly show that the write performance is limited. The performance of the random reads workload is comparable for both SSDs. There is a significant difference when writes are involved. First, both SSDs expose the same behavior for sequential workloads until we reach the 99.9^{th} percentile, where the OCSSD provides a lower latency, by a factor of two. Second, for mixed workload, the OCSSD provides a much lower latency (approximately a factor of three) already for the 99^{th} percentile. This is because reads get much more often stuck after writes on the NVMe SSD and that the OCSSD has more internal parallelism that can be leveraged by writes.

OLTP and OLAP. Figure 7 shows Sysbench's [32] OLTP and OLAP workloads on top of the MySQL database system and an Ext4 file system. The latency error bounds show the min/max for the workloads as well.

Both workloads are currently CPU bound and thus similar for all SSDs. When writing, however, the OLTP workload exhibits significant flush overheads. For 10GB write, 44,000 flushes were sent, with roughly 2GB data padding applied. For OLAP (as for RocksDB), only 400 flushes were sent, with only 16MB additional padding. Thus, for a transactional workload, a device-side buffer would significantly reduce the amount of padding required.

The latency results show the same trend as RocksDB. In the 95^{th} percentile, latency increases a lot for write-heavy OLTP on the traditional SSD compared to the average case, while the increase is insignificant for the open-channel SSD. For OLAP, the results are similar both in terms of throughput and latency due to the workload being CPU-intensive, and mostly read-only. Thus, there is no interference between reads and writes/erases. Tuning SQL databases for performance on open-channel SSDs is an interesting topic for future work (possibly via a KV-based storage engine [39]).

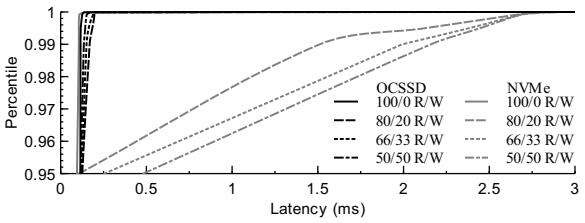

Figure 8: Latency comparison of OCSSD and NVMe SSD showing how writes impact read latencies. Note that the y-axis does not start at 0, but at 0.95.

5.5 Predictable Latency

This experiment illustrates the potential benefits of application-specific FTLs. We use again the modified version of fio to run two concurrent streams of vector I/Os directly to the device. One thread issues 4KB random reads at queue depth 1, while another thread issues 64K writes at the same queue depth. The streams for the OCSSD are isolated to separate PUs, while the NVMe SSD mixes both reads and writes. We measure the workload over five seconds and report the latency percentiles in Figure 8. We report the latency granularities as $100/0, 80/20, 66/33$, and $50/50$. As writes increase, performance remains stable on the OCSSD. While the NVMe SSDs has no method to separate the reads from the writes, and as such higher read latency are introduced even for light workloads (20% writes).

Our point is that the PPA I/O interface enables application developers to explicitly manage the queue for each separate PU in an SSD and thus achieve predictable I/O latency. Characterizing the potential of application-specific FTLs with open-channel SSDs is a topic for future work.

6 Related Work

As SSD shortcomings become apparent [12, 15, 30, 37], research has focused on organizing the cooperation between host and SSDs. One form of cooperation consists of passing hints from hosts to embedded FTLs. The work on multi-streaming falls in this category [28]. Other forms of cooperation consist in bypassing the FTL [24, 35, 50], or designing the upper layers of the system around the properties of a given FTL [29, 33, 36, 38, 46]. Finally, host-based FTLs let the host control data placement and I/O scheduling. This is the approach we have taken with LightNVM.

Host-side FTLs have been implemented by both FusionIO DFS [27] and Violin Memory [54], each moving the FTL into the host in order to expose a block I/O SSD. Similarly, Ouyang et al. [45] proposed Software-Defined Flash that allows a key-value store to integrate with the underlying storage media. Also, Lu et al. [37] defined a host-based object-based FTL (OFTL) on top of raw flash devices that correspond to Baidu's open-channel SSDs. In contrast, we specify a cross-vendor interface for open-channel SSDs, and we show how a tunable block I/O target can reduce read latency variability.

Lee et. al. [34] proposed a new SSD interface, compatible with the legacy block device interface, that exposes error-free append-only segments through read/write/trim operations. This work is based on a top-down approach, which shows how a state-of-art file system (F2FS) can be implemented on top of the proposed append-only interface. Our paper, in contrast, describes a bottom-up approach where the PPA interface reflects SSD characteristics independently of the upper layers of the system. First, our PPA interface allows write and erase errors to propagate up to the host for increased I/O predictability. We also define a vector I/O interface, allowing the host to leverage the device parallelism. Finally, we explicitly expose the read and write granularity of the media to the host so that write buffering can be placed on the host or the device. Demonstrating the benefits of application-specific SSD management with LightNVM is a topic for future research. Initial results with RocksDB [22] or multi-tenant I/O isolation [21, 26] are promising.

7 Conclusion

LightNVM is the open-channel SSD subsystem in the Linux kernel. It exposes any open-channel SSD to the host through the PPA I/O interface. LightNVM also provides a partition manager and a tunable Block I/O interface. Our experimental results show that LightNVM provides (i) low overhead with significant flexibility, (ii) reduced read variability compared to traditional NVMe SSDs, and (iii) the possibility of obtaining predictable latency. Future work includes characterizing the performance of various open-channel SSD models (products from three vendors have been announced at the time of writing), devising tuning strategies for relational database systems and designing application-specific FTLs for key-value stores.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd, Erez Zadok, whose suggestions helped improve this paper. We also thank Alberto Lerner, Mark Callaghan, Laura Caulfield, Stephen Bates, Carla Villegas Pasco, Björn Þór Jónsson, and Ken McConlogue for their valuable comments during the writing of this paper. We thank the Linux kernel community, including Jens Axboe and Christoph Hellwig, for providing feedback on upstream patches, and improving the architecture.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference (ATC)* (2008), pp. 57–70.

[2] AXBOE, J. Fio - Flexible I/O tester. *URL http://freecode. com/projects/fio* (2014).

[3] BJØRLING, M. fio LightNVM I/O Engine. *URL https://github.com/MatiasBjorling/lightnvm-fio* (2016).

[4] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)* (2013).

[5] BJØRLING, M., BONNET, P., BOUGANIM, L., AND DAYAN, N. The necessary death of the block device interface. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)* (2013), pp. 1–4.

[6] BONNET, P., AND BOUGANIM, L. Flash device support for database management. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings* (2011), pp. 1–8.

[7] CAI, Y., HARATSCH, E. F., MUTLU, O., AND MAI, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2012), IEEE, pp. 521–526.

[8] CAI, Y., HARATSCH, E. F., MUTLU, O., AND MAI, K. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe* (San Jose, CA, USA, 2013).

[9] CAI, Y., LUO, Y., GHOSE, S., AND MUTLU, O. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2015), IEEE, pp. 438–449.

[10] CAI, Y., MUTLU, O., HARATSCH, E. F., AND MAI, K. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *IEEE 31st International Conference on Computer Design (ICCD)* (2013).

[11] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *IEEE 17th International Symposium on High Performance Computer Architecture* (2011), IEEE, pp. 266–277.

[12] CHEN, F., LUO, T., AND ZHANG, X. CAFTL : A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. *9th USENIX Conference on File and Storage Technologies (FAST)* (2011).

[13] CRUCIAL. The Crucial M550 SSD. `http://www.crucial.com/usa/en/storage-ssd-m550`, 2013.

[14] CURRY, M. L., SKJELLUM, A., WARD, H. L., AND BRIGHTWELL, R. Accelerating reed-solomon coding in raid systems with gpus. In *IEEE International Symposium on Parallel and Distributed Processing* (April 2008), pp. 1–6.

[15] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.

[16] DONG, G., XIE, N., AND ZHANG, T. On the use of soft-decision error-correction codes in NAND flash memory. *IEEE Transactions on Circuits and Systems I: Regular Papers 58*, 2 (2011), 429–439.

[17] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STUMM, M. Optimizing Space Amplification in RocksDB. *8th Biennial Conference on Innovative Data Systems Research (CIDR)* (2017).

[18] FILKS, V., UNSWORTH, J., AND CHANDRASEKARAN, A. Magic Quadrant for Solid-State Arrays. Tech. rep., Gartner, 2016.

[19] FLOYER, D. Flash Enterprise Adoption Projections. Tech. rep., Wikibon, 2016.

[20] FUSION-IO. Introduces "Flashback" Protection Bring Protective RAID Technology to Solid State Storage and Ensuring Unrivaled Reliability with Redundancy. Businesswire, 2008.

[21] GONZÁLEZ, J., AND BJØRLING, M. Multi-Tenant I/O Isolation with Open-Channel SSDs. *Nonvolatile Memory Workshop (NVMW)* (2017).

[22] GONZÁLEZ, J., BJØRLING, M., LEE, S., DONG, C., AND HUANG, Y. R. Application-Driven Flash Translation Layers on Open-Channel SSDs. *Nonvolatile Memory Workshop (NVMW)* (2014).

[23] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D., CHIEN, A. A., AND GUNAWI, H. S. The tail at store: a revelation from millions of hours of disk and SSD deployments. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 263–276.

[24] HARDOCK, S., PETROV, I., GOTTSTEIN, R., AND BUCHMANN, A. Noftl: Database systems on ftl-less flash storage. *Proceedings of the VLDB Endowment 6*, 12 (2013), 1278–1281.

[25] HOLT, K. Information technology—scsi block commands—3 (sbc-3). Tech. rep., T10/03-224, Working Draft, 2003.

[26] HUANG, J., BADAM, A., CAULFIELD, L., NATH, S., SENGUPTA, S., SHARMA, B., AND QURESHI, M. K. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST)* (2017), USENIX.

[27] JOSEPHSON, W. K., BONGO, L. A., LI, K., AND FLYNN, D. DFS: A File System for Virtualized Flash Storage. *ACM Transactions on Storage 6*, 3 (Sept. 2010), 1–25.

[28] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The Multi-streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2014).

[29] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: transactional FTL for SQLite databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), pp. 97–108.

[30] KIM, J., LEE, D., AND NOH, S. H. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 183–189.

[31] KING, M., DELLETT, L., AND SULLIVAN, K. Annual High Performance Computing Trends Survey. Tech. rep., DDN, 2016.

[32] KOPYTOV, A. Sysbench: a system performance benchmark. *URL: http://sysbench. sourceforge. net* (2004).

[33] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[34] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 339–353.

[35] LIU, M., JUN, S.-W., LEE, S., HICKS, J., ET AL. minFlash: A minimalistic clustered flash array. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2016), IEEE, pp. 1255–1260.

[36] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 133–148.

[37] LU, Y., SHU, J., AND ZHENG, W. Extending the Lifetime of Flash-based Storage Through Reducing Write Amplification from File Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (2013), USENIX Association, pp. 257–270.

[38] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. *6th USENIX Workshop on Hot Topics in Storage and File Systems* (2014).

[39] MATSUNOBU, Y. Rocksdb storage engine for mysql. In *FOSDEM 2016* (2016).

[40] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS Performance Evaluation Review* (2015), vol. 43, ACM, pp. 177–190.

[41] MICHELONI, R., MARELLI, A., AND ESHGHI, K. *Inside solid state drives (SSDs)*, vol. 37. Springer Science & Business Media, 2012.

[42] NARAYANAN, I., WANG, D., JEON, M., SHARMA, B., CAULFIELD, L., SIVASUBRAMANIAM, A., CUTLER, B., LIU, J., KHESSIB, B., AND VAID, K. SSD Failures in Datacenters: What, When and Why? In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (2016), ACM, pp. 407–408.

[43] NVMHCI WORK GROUP. NVM Express 1.2.1, 2016.

[44] O'DWYER, A. *Handbook of PI and PID controller tuning rules*, vol. 57. World Scientific, 2009.

[45] OUYANG, J., LIN, S., JIANG, S., AND HOU, Z. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[46] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. Beyond block I/O: Rethinking traditional storage primitives. In *High Performance Computer Architecture (HPCA)* (2011), IEEE, pp. 301–311.

[47] PAN, Y., DONG, G., AND ZHANG, T. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *FAST* (2011), vol. 11, pp. 18–18.

[48] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In *11th USENIX Conference on File and Storage Technologies (FAST)* (2013), USENIX Association, pp. 298–306.

[49] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST)* (2016), pp. 67–80.

[50] SONG, Y. H., JUNG, S., LEE, S.-W., AND KIM, J.-S. Cosmos OpenSSD: A PCIe-based Open Source SSD Platform OpenSSD Introduction. *Flash Memory Summit* (2014), 1–30.

[51] SUN, H., GRAYSON, P., AND WOOD, B. Quantifying reliability of solid-state storage from multiple aspects. *7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os* (2011).

[52] SWANSON, S., AND CAULFIELD, A. M. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *IEEE Computer 46*, 8 (2013).

[53] SWEENEY, P. *Error control coding*. Prentice Hall UK, 1991.

[54] VIOLIN MEMORY. All Flash Array Architecture, 2012.

[55] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014), 1–14.

[56] WU, G., AND HE, X. Reducing ssd read latency via nand flash program and erase suspension. In *11th USENIX Conference on File and Storage Technologies (FAST)* (2012), vol. 12, pp. 10–10.

[57] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)* (2014).