Open-Channel SSD (What is it Good For)

Ivan Luiz Picoli
IT University of Copenhagen
ivpi@itu.dk

Niclas Hedam
IT University of Copenhagen
nhed@itu.dk

Philippe Bonnet
IT University of Copenhagen
phbo@itu.dk

Pınar Tözün
IT University of Copenhagen
pito@itu.dk

ABSTRACT

Open-Channel SSDs are storage devices that let hosts take full control over data placement and I/O scheduling. In recent years, they have gained acceptance in data centers (e.g., Alibaba) and for computational storage (e.g., Pliops). Open-Channel SSDs require a host-based Flash Translation Layer (FTL) that manages the physical address space they expose. Open-source FTLs are now available for Open-Channel SSDs, providing either a generic yet tunable block device interface (e.g., pblk, SPDK, OX-Block), or application-specific FTLs developed for a specific data system (e.g., LightLSM, OX-ELEOS). In this paper, we share our experience developing three of those FTLs in the context of the OX controller. We position Open-Channel SSDs in the SSD landscape and discuss their relevance for data systems. In particular, we argue that Open-Channel SSDs cannot be considered as a uniform class of devices. Our main contribution is a description of the key design decisions we took in OX related to Open-Channel SSDs. We reflect on lessons learned and propose hints for the co-design of data systems and Open-Channel SSDs.

1. INTRODUCTION

Open-Channel SSDs are disks with a minimal firmware layer [7] that exposes the physical storage space and let a host manage data placement and I/O scheduling. Baidu described the use of Open-Channel SSDs in a proprietary environment in 2014 [13]. LightNVM emerged as the Linux Open-Channel subsystem in 2015 [6]. More recently, Intel incorporated Open-Channel SSDs in the SPDK framework [5], Alibaba announced its efforts to incorporate Open-Channel SSDs in their cloud infrastructure [1], and Pliops started advertising their Key-Value system over Open-Channel SSDs [14]. Yet, the number of manufacturers commercializing Open-Channel SSDs remains low (CNEX Labs, Lite-On, Shannon) and there is no Open-Channel SSD standard agreed upon in the storage industry, despite significant efforts.

Are Open-Channel SSDs getting mainstream or are they just a transient step in the evolution of SSDs? Are Open-Channel SSDs relevant for the design of data systems? How? Can they even be considered as a uniform class of devices? These are the questions we tackle in this paper.

We have worked with two generations of Open-Channel SSDs in the context of our work on the OX Controller [11]. OX is a software framework for programming computational storage controllers. We designed and evaluated OX on the DFC card, equipped with an ARMv8 storage controller connected to up to 2 Open-Channel SSDs, and accessible from a host via PCIe or 40GE. In the context of OX, we designed a generic Flash Translation Layer (FTL) that exposes Open-Channel SSDs as block devices, and two application-specific FTLs: OX-ELEOS for log-structured storage in LLAMA [9] and LightLSM for LSM-tree storage management [12]. In this paper, we describe the lessons learned designing these storage engine components. More specifically, our contributions are the following:

1. We describe the current SSD landscape, which has evolved much in recent years, and discuss the relevance of Open-Channel SSDs;
2. We review key design decisions we took in OX with respect to Open-Channel SSDs.
3. We present lessons learned as well as design hints for the co-design of data system with an Open-Channel SSD.

2. OPEN-CHANNEL SSD

In this section, we briefly review the characteristics of SSD storage space, how it is abstracted at the SSD interface, and how it is made available on the host.

2.1 Physical Storage Space

All SSDs are composed of a collection of storage chips wired in parallel to a controller via independent channels. There are no interferences across channels. Several storage chips are connected to the same channel. There might be interferences across chips on the same channel (e.g., a write request on one chip might have to wait until a read request on another chip completes before it can be issued). Operations are sequential within a storage chip. SSD manufacturers define the number of channels in an SSD, and the number of storage chips per channel.

Flash chips are multi-dimensional arrays of flash cells, organized in sectors, pages, blocks and planes. The number of

10th Annual Conference on Innovative Data Systems Research (CIDR '20)
January 12-15, 2020, Amsterdam, Netherlands.

bits stored in a cell is one for SLC chips, 2 for MLC chips, 3 for TLC chips and 4 for QLC chips. Higher density enables lower cost at the price of lower performance and lower resilience to wear. Sectors are the unit of read, typically 4 KB. A flash page is the unit of write to the chip. It is composed of 4 sectors together with out-of-bound space. A block is typically composed of 512 pages. It must be erased, before the pages it contains can be written. Pages must be written sequentially within a block.

Chips can be composed of several planes (2 or 4). Pages at the same address on two different planes must be written (or read) together. These writes are executed in parallel. Planes are thus a way to increase capacity and storage density, without increasing storage latency.

Each cell is partitioned across so-called *paired pages*: 1 page per bit stored on the cell, i.e., 1 with SLC, 2 with MLC, 3 with TLC and 4 with QLC. All paired pages must be written before one of them can be read. As a result, the unit of write increases with storage density, e.g., on a QLC chip with 4 planes, 4 paired pages must be written together on four planes, as a result the unit of write is 16 pages `16*4 sectors = 16*4*4KB = 256 KB`.

2.2 Interface

The Open-Channel Interface specification, defined in the context of LightNVM [3], is an extension of the NVMe protocol. It provides an abstraction of the SSD physical address space. Each SSD is organized as a collection of groups. The Open-Channel SSD controller guarantees that there is no interferences across groups. Each group is composed of parallel units (PUs) (unit of parallelism). Each PU is a collection of chunks. In each chunk, logical blocks (unit of write) are written sequentially. The size of a logical block is defined by the size of a sector on the SSD. The unit of write is one or several logical blocks, again depending on the SSD (e.g., 24 logical blocks on a dual-plane TLC drive, corresponding to 4 (sectors per page) * 3 (paired pages) * 2 (planes)). A chunk must be reset before it is written again.

The interface defines administration commands, access to the device geometry, and vector data commands, supporting scatter-gather reads and writes of logical blocks (to and from the host), chunk reset and copy of logical blocks (within the Open-Channel SSD, without host involvement).

Chunk management (e.g., abstracting planes and paired pages) as well as bad media management and asynchronous error reporting are under the responsibility of the Open-Channel SSD.

2.3 Standardization

The first version of the Open-Channel SSD Interface is defined by M.Bjørling during his PhD and released in April 2016. The Westlake series of Open-Channel SSDs, from CNEX Labs implemented this interface, which is now deprecated. The second version, described above, was released in January 2018. It incorporated the feedback of the storage industry, and is implemented in the Granby series from CNEX Labs. There were efforts, in the context of the Denali consortium [4], led by Microsoft and CNEX Labs to standardize the Open-Channel interface. But these efforts did not succeed, and today, there is no Open-Channel SSD standard.

As the Open-Channel SSD Interface was defined as an extension of NVMe, this would have been the natural body for its standardization. But, NVMe is not following this route. Instead, NVMe is preparing a standard on Zoned Namespaces (ZNS) [2], which incorporates some ideas from Open-Channel SSDs. ZNS exposes a disk as a collection of zones that must be written sequentially and reset before rewriting. It is basically a form of append-only storage abstraction that shields the host from the complexities of the physical address space. The host is responsible for creating zones, writing to and reading from zones, and reclaiming zones (a form of garbage collection). The SSD is responsible for the placement of zones on physical media, I/O scheduling as well as wear levelling. ZNS can be implemented as an application-specific Flash Translation Layer on top of Open-Channel SSDs. ZNS is compliant with Shingled Magnetic Recording (SMR) disks and thus provides support for legacy archival applications, which is a great advantage for QLC SSDs. It is also a way to integrate different types of storage devices, with varying density/performance profiles, under a unified interface. How to best port legacy data systems from a block device abstraction to ZNS is an open issue.

2.4 Host-Based Software Frameworks

By definition, Open-Channel SSDs require the host to manage data placement and I/O scheduling. The host thus needs software support for (i) a *driver* accessing the Open-Channel SSD interface and (ii) an *SSD management layer* in charge of provisioning, data placement and I/O scheduling. It is for this purpose, that LightNVM was introduced as the Linux framework for Open-Channel SSDs [6]. It has evolved to accommodate the evolution of the Open-Channel SSD interface, and to provide support for legacy applications, with the introduction of **`pblk`**, a Flash Translation Layer in kernel space, fully supporting the block device abstraction. **`liblightnvm`** is a user-space library that exposes the Open-Channel interface directly to user-space.

The SSD management layer that is necessary with Open-Channel SSDs is typically a Flash Translation Layer, in charge of mapping logical and physical address spaces. We distinguish between (a) generic FTLs, that expose a block device abstraction and (b) application-specific FTLs, that map a logical address space specifically defined for an application, or a data system, to the physical address space exposed by liblightnvm.

Intel recently incorporated Open-Channel SSDs in the SPDK framework [5]. SPDK now provides a generic FTL in user-space, on top of Open-Channel SSDs. SPDK FTL and pblk are both open-source implementations of generic FTLs. This represents a significant evolution for the storage industry, as FTLs used to be protected as company secrets.

Are Open-Channel SSDs limited to legacy applications supported through pblk or SPDK? Are they a transient evolution stop from block device to ZNS drives? Can the complexity of application-specific SSD management ever be justified? These existential questions must be addressed to evaluate whether and how Open-Channel SSDs are relevant to data systems. So, let us review the SSD landscape.

3. THE SSD LANDSCAPE

Open-Channel SSDs are just one of the many evolutions of the SSD landscape in recent years. Other notable evolutions include the evolution of NAND technology (QLC, 3D NAND, Z-NAND), the emergence of persistent memory (3D-Xpoint), the definition of Zoned Namespaces as a

FTL abstraction FTL placement	Block-device	ZNS	App-Specific
Host	**Fusion-IO** (SLC/MLC, kernel space, black box, host) **Pblk** (MLC/TLC, kernel space, white box, host) **SPDK** (MLC/TLC, user space, white box, host)	**LightNVM target for ZNS** (TLC, kernel space, white box, host)	**RocksDB NVM engine** (MLC/TLC, user space, white box, host)
Controller	**Traditional SSDs** (any, embedded, black box, host) **Smart SSD** (QLC, embedded, black box, controller) **OX-Block** (MLC, user space, white box, controller)	**ZNS SSD** (any, embedded, black-box, host) **OX-ZNS** (TLC, user space, white box, controller)	**KV-SSD** (QLC, embedded, black box, host) **Pliops** (TLC, user-space, black box, controller) **OX-Eleos, LightLSM** (MLC, user-space, white box, controller)

Figure 1: SSD models organized by FTL placement and abstraction. Additional dimensions are indicated in parenthesis next to the SSD model name. Lighter colors indicate models that aren't fully available today.

NVMe standard, the availability of computational storage devices (SmartSSD from Samsung, devices from ScaleFlux or NGD Systems) and special-purpose SSDs (KV-SSD from Samsung [10]).

The discussion in this section does not include any consideration on the market mechanisms that impact the success of a solution. We focus on the technical aspects of the SSD landscape. We focus on the SSD hardware and on the system services that make it available to data systems.

3.1 Design Space

We identify the following dimensions as a means to characterize the SSD landscape:

Storage chip: The storage chip is the core component of the SSD hardware. It can be designed for low-latency (e.g., Z-NAND SLC or 3D-Xpoint) or high capacity (QLC). Intermediate NAND technologies, MLC and TLC, make it possible to trade latency for density and thus introduce various options in terms of $/GB and IOPS/GB.

FTL placement: FTL is a necessary component of the system services defined on top of SSD hardware. It can be placed on the storage controller or on the host. FTL placement impacts where resources are being consumed. FTL placement on a storage controller is necessary in the context of computational storage, where the goal is to offload the tasks from the host CPU to the storage device and minimize data movement.

FTL integration: FTL can be part of an SSD firmware, it can be part of the operating system kernel, or it can be defined in user-space. An FTL in firmware gets access to all the resources of the controller efficiently. An FTL in kernel space gets direct access to all OS resources efficiently. An FTL in user-space is flexible; it is not constrained by the licensing model of the Linux kernel.

FTL transparency: An FTL can be a black box for the data systems that relies on its services or a white box.

FTL abstraction: As we have seen in the previous section, an FTL can expose a block device abstraction, a ZNS device or it can be application-specific. While a block device abstraction supports file systems and legacy applications, both ZNS and application-specific FTLs require a redesign of the data systems that access them. With ZNS, this redesign consists of adopting an append-only storage discipline for storage management. With application-specific FTLs, this redesign makes it possible to streamline the data path at the cost of the increased complexity of managing the physical address space of the underlying Open-Channel SSD.

FTL access: FTL can be accessed from the host or from the storage controller (with computational storage).

Figure 1 illustrates how a range of SSDs are positioned with respect to FTL abstraction and FTL placement. Interestingly, traditional SSDs and SmartSSD (the computational storage platform recently announced by Samsung) are in the same quadrant using those two dimensions. ZNS drives were demonstrated by Western Digital and Radian Memory Technologies at the Flash Memory Summit in August 2019 and won a *Best of Show Award* for *Most Innovative Flash Technology* category. It should be straightforward to define a LightNVM target that exposes the ZNS interface through a host-based FTL on top of Open-Channel SSDs, but this has not - to the best of our knowledge - been released or even announced. The RocksDB NVM engine was originally developed by J.Gonzalez at CNEX Labs for MLC drives and has recently been ported to TLC drives by Niclas Hedam in the context of our work on LightLSM (see next section). KV-SSD is a key-value store SSD commercialized by Samsung [10]. It is equipped with a KV FTL implemented on top of a block device abstraction. Pliops supports a KV FTL over a CNEX Labs Open-Channel SSD [14].

3.2 Open-Channel SSDs

Let us review how Open-Channel SSDs relate to the dimensions identified above:

Storage chip: The Open-Channel SSD interface does not make any assumption about the underlying storage chip. However, the characteristics of the storage chip has a deep impact on the design of an FTL over an Open-Channel SSD. This is one of the points we illustrate in the next section. The consequence is that an FTL is not designed for a generic class of Open-Channel SSD, but for a given Open-Channel SSD, with a specified storage chip.

FTL placement: LightLSM was designed for host-based Open-Channel SSD management. But the emergence of computational storage platforms with FPGA or ARM-based CPUs running Linux makes it possible to access Open-Channel SSDs on a storage controller. This way, the FTL can be placed on the host or on the storage controller (as long as there exists a software framework such as OX, see next section).

FTL integration: With Open-Channel SSDs, the FTL can be located in kernel-space (pblk) or in user-space (on top of liblightnvm).

FTL transparency: With Open-Channel SSDs, the FTL can be a white box or a black box (with a proprietary license in user space).

FTL abstraction: Open-Channel SSDs do not dictate the type of abstraction provided by the FTL through which they are accessed. However, Open-Channel SSDs are the only type of SSDs that make it possible to streamline the data path in the context of an application-specific FTL.

While Open-Channel SSDs appear in all the quadrants in Figure 1, the optimizations they enable (streamlining of the data path, control over interferences, adaptation to storage chip characteristics) is best leveraged in the context of application-specific FTLs, placed on the storage controller. This way the cost of the increased complexity of dealing with the SSD physical address space can be offset with the benefits of offloading the host CPU and minimizing data movement.

4. OX DESIGN

4.1 OX Framework

OX is a framework for programming storage devices equipped with computational capabilities., i.e., a programmable storage controller. OX was implemented and evaluated with the DFC card, i.e., an ARMv8 storage controller on top of various storage media, including Open-Channel SSDs. OX is composed of three layers. The bottom layer focuses on media management, it is responsible for abstracting various forms of underlying storage media under a common representation of the physical address space. The middle layer is responsible for the translation of logical addresses to physical addresses. It is an FTL layer. The upper layer is the host interface that supports the commands and notifications that the host applications uses to store and retrieve data with OX.

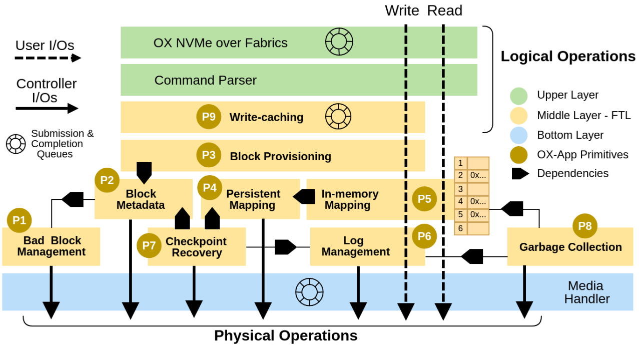

Figure 2: Architecture of a modular FTL within the OX Controller

With OX, we target data systems over computational storage. Our hypothesis is that the storage engine of such a data system must incorporate an FTL. We thus designed a modular FTL that can be reused and customized across data systems. Figure 2 shows the core components of the modular OX FTL architecture. Controller I/Os represented by solid lines, are executed synchronously and generated by several components: (i) garbage collection may read and write to media, (ii) recovery log may be persisted according to atomic requirements, (iii) mapping and block metadata may be persisted during checkpoint process, (iv) mapping information may be read and persisted by caching mechanisms, and (v) bad block information may be updated at any time. User I/Os, represented by dashed lines are executed asynchronously. Interested readers are referred to [11] for a complete presentation of OX.

4.2 OX-Based FTLs

We designed and implemented one generic (OX-Block) and two application-specific FTLs (OX-ELEOS and LightLSM). The goal of OX-ELEOS is to reduce the load on the host CPU in a data system based on the LLAMA storage engine [9]. The goal of LightLSM is to minimize data movement and avoid congestions due to compaction in RocksDB [12]. In the reminder of this section, we review key design decisions we took with respect to Open-Channel SSDs when designing those FTLs.

OX-Block exposes Open-Channel SSDs as block devices. We assume 4 KB as the minimum read granularity in the underlying Open-Channel SSD and OX-Block exposes a logical address space composed of 4KB blocks. Thus, OX-Block maintains a 4KB-granularity page-level mapping table.

OX-ELEOS exposes Open-Channel SSDs as log-structured storage, with writes at the granularity of Log-Structured Storage (LSS) I/O buffers, typically 8MB, and reads at the granularity of a single page. With fixed sized pages of 4KB, mapping in OX-ELEOS is similar to mapping in OX-Block. However, with variable-sized pages of an arbitrary number of bytes, mapping becomes more challenging. Describing our solution is beyond the scope of this paper. Our point here is that application-specific FTLs might require mapping at a granularity which is smaller than the unit of read on an Open-Channel SSD.

LightLSM exposes Open-Channel SSDs as a RocksDB environment supporting SSTable flush and block reads. In RocksDB, a block is the unit of transfer for reads and writes. The size of an SSTable is a multiple of the RocksDB block size. On a dual-plane TLC drive, the size of a RocksDB block must be a multiple of 96KB, i.e., the minimum unit of write on a single chunk. By forcing the units of read and write to be similar, RocksDB thus causes the unit of read to be many times larger than possible with the underlying Open-Channel SSD. Note again that the ratio between unit of read (typically 4KB) and the unit of write varies depending on the drive characteristics.

4.3 OX Mechanisms

The OX FTL layer relies on the media manager and its abstraction of Open-Channel SSDs. In this section, we review the core OX mechanisms and how they interact with Open-Channel SSDs.

Transactional Support Each operation defined at the FTL API is a transaction [8]: the FTL must ensure atomicity and durability. The problem is that the vectorized operations supported by Open-Channel SSDs are not atomic. The only atomic operation within an SSD is read/write of a flash page. In all our designs, we use write-ahead logging (WAL) and checkpoints to ensure atomicity and durability of FTL writes. The description of our WAL, checkpointing and recovery designs are beyond the scope of this paper. However, we illustrate the impact of checkpointing on recovery time with OX-Block. We simulate a fatal failure by killing OX during an experiment. All metadata in volatile memory is lost, some updates since last checkpoint might not be persisted, and the Open-Channel SSD is left inconsistent. Such a failure forces OX to rely on recovery to reconstruct metadata and mapping information and bring the Open-Channel SSD back to a consistent state. Before the failure, OX-Block manages random writes of up to 1 MB in size; each of these writes is a transaction. To cause the failure, we kill OX with *sudo kill -9 <process>*. We vary the point in time at which the failure occurs. We consider six different points in time *T1-6*. After the failure, we restart OX so that recovery takes place. Figure 3 shows the experiment for disabled checkpoint as well as checkpoints every C_i 10, and C_i 30 seconds.

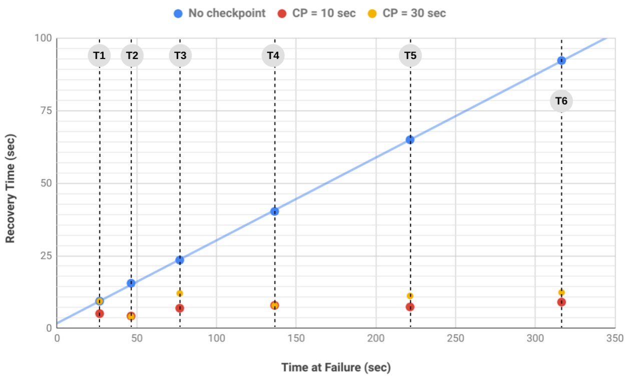

Figure 3: Impact of checkpoint intervals on recovery time.

The blue line represents recovery time when checkpointing is disabled. Dots show the timing of a failure (i.e., *T1-T6*). Without checkpointing, recovery time increases linearly with the amount of log records written to disk; it is close to 100 seconds at *T6*. When checkpoints are introduced, recovery time decreases dramatically for longer experiments. We also see that recovery time oscillates up and down and remains constant. This is important for performance predictability. It was expected as the checkpoint process truncates the log at regular intervals. The difference between checkpoint frequencies of C_i 10 and C_i 30 sec is not significant.

Garbage Collection For garbage collection, OX-Block marks a group for collection. Then, background threads recycle victim chunks within that group. This guarantees locality of interferences from garbage collection. Put differently, a significant percentage of application reads and writes are not affected by garbage collection interferences. On an SSD with 16 channels, this percentage is 93,7%. On an SSD with 8 channels, this percentage is 87,5%. In LightLSM, garbage collection is a side-effect of compaction. Interferences between compaction and flush operations depend on how SSTables are mapped onto the Open-Channel SSD (see Figure 4). We set the size of SSTables to 768 MB. This corresponds to the number of parallel units (32) multiplied by the size of a chunk (24 MB). The rationale for this data placement position is that we do not want to consider several SSTables per chunk. As SSTables are the unit of space reclamation in RocksDB, our mapping guarantees that garbage collection does not result in read and write operations of invalid pages within chunks. Each SSTable deletion only causes chunk erases. With a horizontal mapping, each SSTable is mapped onto all available PUs. As a result, any compaction operation interferes with SSTable flush or block read. With a vertical mapping, each SSTable is mapped onto a single group. This way, compaction might occur without interferences from SSTable flushes.

Write Pointer The Open-Channel interface requires that logical blocks are written sequentially within a chunk. As a result, LightLSM maintains a write pointer per chunk. We consider a single dispatch thread for submitting I/Os to the Open-Channel SSD so that there are no concurrent accesses to the write pointers.

Figure 5 shows RocksDB throughput in operations per second when running three db_bench workloads[1]: fill-sequential, read-sequential and read-random. The results are obtained with RocksDBv5.4.6 without any compression or caching enabled to put more stress on SSD accesses. We use 16 byte keys and 1KB values. We run fill-sequential first. In all scenarios, there are 3 levels of SSTables on disk (L0, L1, L2) at the end of this run. Read-sequential and read-random run over the database populated by fill-sequential. We also vary the number of threads submitting put/get requests in db_bench. Each db_bench thread submits the same workload. For fill-sequential, each thread writes 3GB sequentially.

The graph shows that write throughput is much higher than read throughput. This is expected as the Open-Channel SSD implements a write-back policy where writes complete as soon as they hit the storage controller cache. On the other hand, the media must be accessed for a read I/O to complete. We obtain highest write performance with 2 db_bench threads with horizontal placement. In fact, with horizontal placement write performance is already very good when we consider a single db_bench thread. Indeed, the SSTable is striped across all PUs. However, performance degrades by 60% when considering 4 or 8 db_bench threads. In contrast, the performance of fill-sequential for vertical placement scales gracefully with the number of threads. So, with one thread we observe 4x more throughput with horizontal placement compared to vertical, while with eight threads vertical yiels 2x more throughput than horizontal. This is unexpected.

The throughput of read-sequential is much higher than the throughput of read-random. This is also expected, as each random read might traverse several SSTables, depending on the performance of bloom filters. Data placement has a marginal impact on read performance. Horizontal placement consistently dominates vertical placement.

Figure 6 shows throughput as a function of time for various numbers of db_bench threads for fill-sequential. The graph shows that throughput remains high with horizontal placement when using 1 or 2 db_bench threads. While the time it takes to complete the benchmark increases significantly when using 4 or 8 threads. Overall, RocksDB throughput fluctuates frequently. Our hyptothesis is that we observe throttling due to RocksDB rate limiter. Tuning RocksDB's rate limiter with LightLSM is a topic for future work. With vertical placement, the graph shows a peak of throughput for a single thread even though the average

[1] db_bench is RockDB's main benchmarking tool. See https://github.com/facebook/rocksdb/wiki/Benchmarking-tools

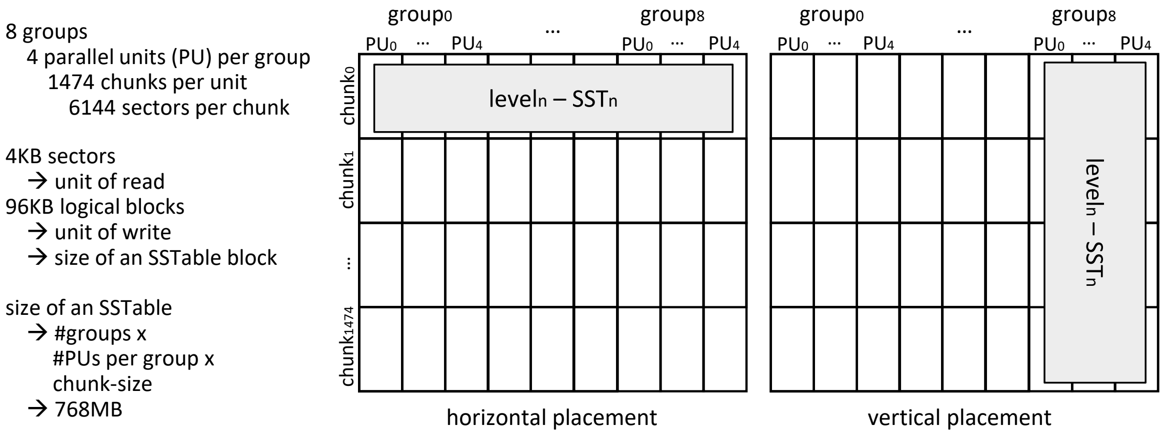

Figure 4: Data placement (horizontal vs. vertical) on Open-Channel SSD.

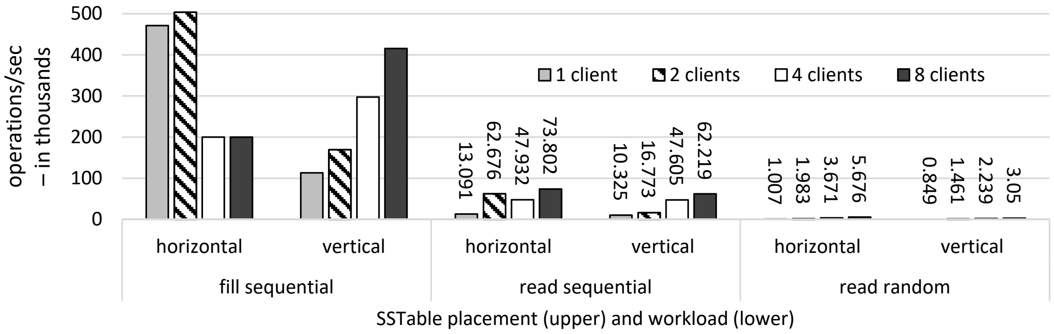

Figure 5: RocksDB average throughput (operations/sec) with db_bench while running fill-sequential, read-sequential and read-random for horizontal and vertical SSTable placement. Data size is 3GB, keys are 16B, and values are 1KB.

throughput is the lowest for this case. As the number of threads increases, the throughput peaks are lower and overall more stable. Since the average throughput for vertical placement is higher with larger number of clients, the time it takes to complete the fill-sequential workload is shorter for the cases with larger number of clients.

Read and Write Paths All FTL APIs contain a form of read and write commands. The read and write paths through the FTL are different. Writes must involve caching (if write-back is supported), provisioning of the physical space, insertion in the mapping table and log management. However, reads only require a lookup in the mapping table. This is important to guarantee that the FTL introduces a minimal overhead with respect to the underlying Open-Channel SSD. Data copies are necessary on the write path, as writes are buffered in order to support write-back semantics and to deal with the constraints imposed on flash (e.g., large unit of write), or queued when waiting for access to a storage chip. Figure 7 show CPU utilization on the DFC platform when writing to OX-ELEOS from a varying number of threads on the host application. The storage controller is saturated with 2 host threads, because it cannot keep up with the data copies within OX: from the network stack to the FTL, and from the FTL to the Open-Channel SSD.

4.4 Lessons Learnt

Let us reiterate and emphasize the main lessons learned: As was already clear from the SSD landscape analysis in Section 3, application-specific FTLs are most relevant when placed on the storage controller in the context of computational storage. As a result, the metrics used to evaluate an application-specific FTL should be (i) how do they help reducing host CPU load? (ii) how do they impact the movement of data to and from the host, and (iii) can they guarantee predictable performance and minimal overhead given the characteristics of the underlying Open-Channel SSD?

The characteristics of an Open-Channel SSD have a significant impact on the FTL design. These characteristics include the nature of the storage chip that impacts the unit of write and the number of channels. The number of chips per channel impact the nature of parallelism within the SSD, which impacts mapping and GC design decisions. *When designing an FTL, Open-Channel SSDs cannot be considered as a uniform class of devices.* An FTL is designed for a given type of Open-Channel SSD. As a consequence, *we should not consider the design of a data system on Open-Channel SSDs but the co-design of data system, storage controller and Open-Channel SSD.*

The management of paired pages and planes is under the responsibility of the Open-Channel SSD. With SLC, such a management is trivial. With QLC, or even TLC, this management is complex, error-prone and might introduce weird constraints on the sequences of reads and writes that

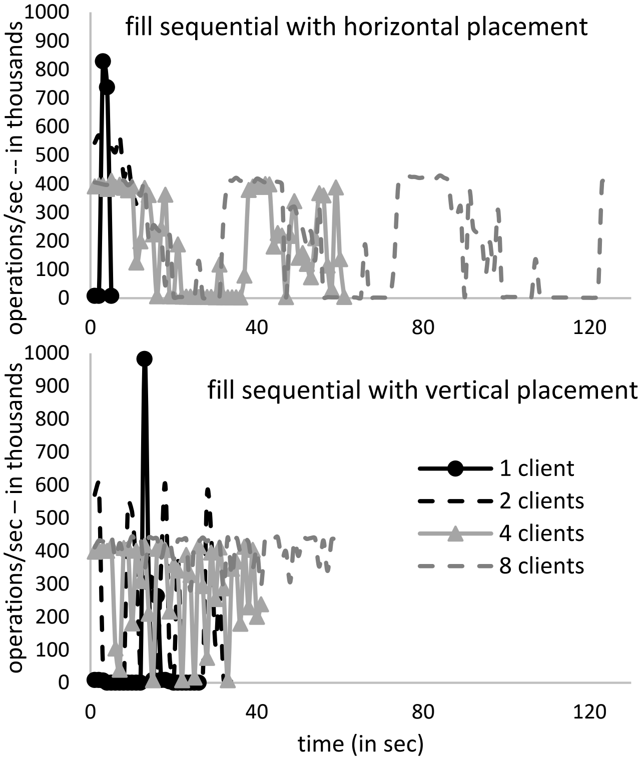

Figure 6: RocksDB throughput (operations/sec) as a function of time with db_bench fill-sequential for horizontal and vertical placement.

can or cannot be issued by the FTL. In fact, it is questionable whether paired pages and planes on QLC (or TLC) can effectively be managed without actually controlling data placement and I/O scheduling. Note that this is what ZNS actually enforces. Also, Open-Channel SSDs are best suited to support the streamlining of the data path that is required for ultra-low latency SSDs. As a result, *we consider that the Open-Channel interface is a good match for the upcoming generation of ultra-low latency SSDs (e.g., equipped with Z-NAND), while ZNS is a good match for higher capacity SSDs (e.g., equipped with TLC or QLC NAND).*

Designing an application-specific FTL is a complex endeavor. Reusing a modular FTL framework helps as it provides default solutions and helpful references in the design space, but it does not take away the complexity of dealing with small and variable-size data, large transactions or high-throughput and low latency data systems. Whether this extra complexity is justified by significant gains in terms of CPU offload, data movement avoidance and performance predictability is still an open issue.

In the context of computational storage, the FTL embedded on the storage controller should introduce a minimum overhead. As a result, data copies should be (a) avoided, and (b) as efficient as possible if absolutely necessary. Avoiding data copies requires support from the operating system (e.g., AF_XDP zero-copy sockets) or hardware acceleration (e.g., hardware ROCE). The efficiency of data copies depend on the RAM modules accessed by the storage controller.

5. CO-DESIGN HINTS

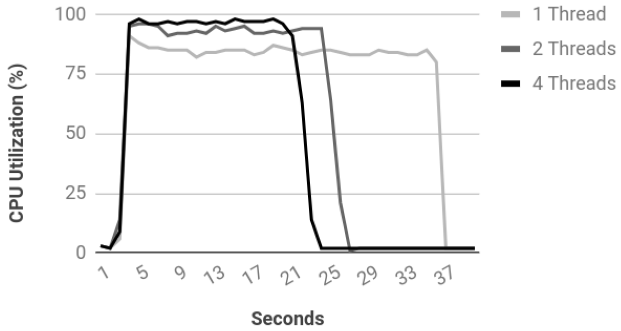

Figure 7: Impact of Data Copies on storage controller utilization

From those lessons learned, we derive a few design hints for the design of data systems equipped with application-specific FTLs over Open-Channel SSDs.

Beware of the interface fallacy. When considering an application-specific FTL, designing the storage interface is essential even if it might seem trivial. For instance, RocksDB assumes that units of reads and writes are similar. Should LightLSM support or challenge this assumption? More generally, what is the most appropriate storage interface for supporting a key-value store? NVMe is standardizing a KV interface, inspired by KV-SSD. How does it compare to LightLSM that supports flush and probe. This is an open issue.

Beware of the atomicity fallacy. Given a storage device, it is easy to make incorrect assumptions about the granularity of atomic writes. On block devices, the unit of write is a block. Writing a single block will succeed or fail. However, writing multiple blocks (e.g., when persisting huge memory pages) might result in partial failures that must be handled by the data system (e.g., InnoDB double buffering). This problem is even more acute for Open-Channel SSDs, where the unit of write changes from one model to the other. When defining an application-specific storage interface, it is possible to provide transactional guarantees, either explicitly through commands defining transactional boundaries, or implicitly (e.g., atomic SSTable flush in LightLSM). A storage interface providing transactional guarantees is a sharp departure from the traditional POSIX I/O abstraction. This has significant implications for data system design. For example, RocksDB is built around the assumption that the storage manager does not compensate for the limitations of the POSIX I/O API. It is the MANIFEST that is used as a transactional log for recovery purpose. With LightLSM, RocksDB does not need MANIFEST.

Embrace hardware dependencies. Hardware characteristics have a far reaching impact on FTL design. We should avoid the illusion that an FTL is designed for Open-Channel SSDs, as if there was a uniform class of devices. Adopting a co-design approach makes it possible to either (a) pick an Open-Channel SSD and design for it (without claiming generality) or (b) pick an Open-Channel SSD that matches the requirement of a given application-specific FTL.

Require a performance contract, not a warranty. In a co-design approach, involving software, firmware and hardware teams, possibly from different companies, it is of the essence to agree on performance contracts (in addition to clear interfaces) across components. When designing an application-specific FTL, it is essential to either (a) precisely characterize the performance of the chosen underlying Open-Channel SSD or (b) evaluate which Open-Channel SSD actually complies with the performance requirements. Without a performance contract, assumptions are made on both sides of the co-design process. The risk that these assumptions are incorrect or inconsistent adds to the complexity of designing for an Open-Channel SSD. Historically, storage devices have come with warranties defined by the manufacturer, typically in terms of life time. Such warranties are offered under the assumption that data is stored on a single storage device. Such an assumption is rarely valid in a storage hierarchy that combines persistent memory and SSDs, or for data systems that manage replication. For such data systems, it is much preferable that a storage device fails early rather than spending energy and time compensating for bit errors. Performance contracts taking wear into account would address this issue. Defining such performance contracts is a topic for future work.

6. CONCLUSION

In this paper, we surveyed Open-Channel SSDs and identified their relevance for data systems. Based on the lessons we learned, we consider that Open-Channel SSDs are not a uniform class of devices. The Open-Channel interface is most relevant for ultra-low latency SSDs accessed by an application-specific FTL placed on the storage controller of a computational storage device. Such solutions require a co-design process that is complex to manage. We identified open issues such as the evaluation of computational storage solutions for key-value stores, the design of application-specific FTLs for a range of data systems, and the porting of legacy data systems on ZNS drives.

7. REFERENCES

[1] Alibaba Launches Dual-mode SSD to Optimize Hyper-scale Infrastructure Performance. https://datacenternews.asia/story/alibaba-launches-dual-mode-ssd-optimize-hyper-scale-infrastructure.

[2] NVMe Zoned Namespaces. https://zonedstorage.io/introduction/zns/.

[3] Open-Channel Solid State Drives Specification Revision 2.0. http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf.

[4] Project Denali to define flexible SSDs for cloud-scale applications. https://azure.microsoft.com/en-us/blog/project-denali-to-define-flexible-ssds-for-cloud-scale-applications.

[5] SPDK FTL on Open-Channel SSD. https://spdk.io/doc/ftl.html.

[6] M. Bjørling, J. González, and P. Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *FAST*, pages 359–373, 2017.

[7] P. Bonnet and L. Bouganim. Flash Device Support for Database Management. In *CIDR*, 2011.

[8] J.-Y. Choi, E. H. Nam, Y. J. Seong, J. H. Yoon, S. Lee, H. S. Kim, J. Park, Y.-J. Woo, S. Lee, and S. L. Min. Hil: A framework for compositional ftl development and provably-correct crash recovery. *ACM Trans. Storage*, 14(4):36:1–36:29, Dec. 2018.

[9] J. Do, D. B. Lomet, and I. L. Picoli. Improving CPU I/O Performance via SSD Controller FTL Support for Batched Writes. In *DaMoN*, pages 2:1–2:8, 2019.

[10] Y. Kang, R. Pitchumani, P. Mishra, Y. Kee, F. Londono, S. Oh, J. Lee, and D. D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *SYSTOR*, pages 144–154, 2019.

[11] I. L. Picoli. *OX: Deconstructing the FTL for Computational Storage*. PhD Thesis. IT University of Copenhagen, 2019.

[12] I. L. Picoli, P. Bonnet, and P. Tözün. LSM Management on Computational Storage. In *DaMoN*, pages 17:1–17:3, 2019.

[13] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *EuroSys*, pages 16:1–16:14, 2014.

[14] I. B. Zion. Key-value FTL over open channel SSD. In *SYSTOR*, page 192, 2019.