LightNVM: The Linux Open-Channel SSD Subsystem

Matias Bjørling∗† Javier González† Philippe Bonnet∗
∗*IT University of Copenhagen* †*CNEX Labs, Inc.*

Extended Abstract

Solid-State Drives (SSDs) are projected to become the dominant form of secondary storage in the coming years. Despite their success due to superior performance, SSDs suffer well-documented shortcomings: log-on-log, large tail-latencies, unpredictable I/O latency, and resource under-utilization. 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.

A new class of SSDs, branded as *Open-Channel SSDs*, is emerging on the market in order to address these shortcomings. 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. Also, Fusion-IO and Violin Memory 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.

In our FAST'17 paper[1], 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.

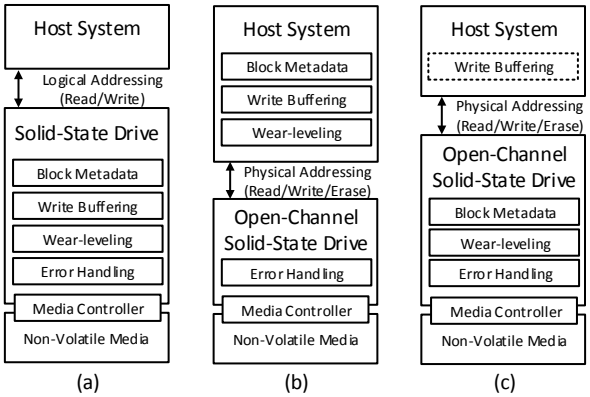

Figure 1: Core SSD management modules on (a) a traditional Block I/O SSD, (b) the class of open-channel SSD considered in this paper, and (c) future open-channel SSDs.

1. Open-Channel SSDs

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 media management are explicit, requiring the host to manage media in details and (c) future open-channel SSDs that splits the responsibility such that the host only manages log-structured writes, media management using device feedback, and wear-leveling. The definition of the PPA I/O interface and the architecture of LightNVM encompass all types of open-channel SSDs. Since the publication of the paper, the future open-channel SSD interface has matured and is now being standardized.

2. Physical Address Addressing

The Physical Page Address (PPA) command set defines a set of commands to expose the geometry of the SSD, and I/O commands to access it efficiently. The SSD is organized as a decomposition hierarchy that reflects the SSD geometry and I/O boundaries.

The decomposition is divided into a hierarchy of groups (e.g., controller channels), parallel units (e.g., independent I/O units within the SSD), For each parallel unit, a set of chunks is defined. Each chunk has a range of logical blocks where writes must be log-structured and written sequential. The

set of groups and parallel units is the total parallelism of the device, and each parallel unit is independent of other units. When the parallelism is exposed explicitly, the host can build a consistent drive model and provide I/O predictability to its applications.

The interface exposes a geometry command, of which the drive geometry is identified. This includes number of groups, parallel units, chunks, and also media relevant timings for how long it takes to read/write/reset to/from a chunk. The read granularity is a logical block (e.g., 4KB), while writes can be a multiple of multiple logical blocks (e.g., 16K or 32K). When a chunk has been written, it must be reset such that it can be written again.

The geometry enables the host to issue I/Os to the appropriate parallel units, and manage the SSD. The parallel units are laid our sequentially in the logical block address address space, and thus if data is stored across multiple parallel units. A read of the data, will leave the host no other choice that to issue multiple read commands, which both increases latency, and CPU utilization on the host. To mitigate this, the interface also exposes a set of vector I/O commands. Which, in addition to data and metadata buffers, replaces the logical block address with a list of logical block addresses. The drive will read the list and fetch the logical blocks in parallel taking advantage of both hardware acceleration, as well as better utilization of the media.

3. LightNVM

The Linux kernel open-channel SSD subsystem (LightNVM) is organized in three layers (see Figure 2), each providing a level of abstraction for open-channel SSDs:

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 flash translation layers and user-space applications can understand the device geometry before use. It also exposes the vector interface through the NVMe device driver's private I/O interface, enabling vector I/Os to be efficiently issued through it.

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 or an application-specific interface provided by a custom target.

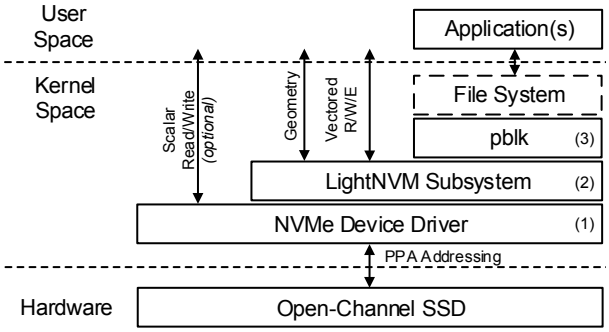

Figure 2: LightNVM Subsystem Architecture

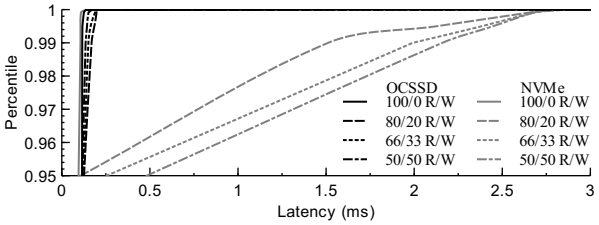

Figure 3: Latency comparison of OCSSD and NVMe SSD showing how writes impact read latencies. Note that the y-axis does not start at 0, but at 0.95.

pblk implements a fully associative, host-based FTL that exposes a traditional block I/O interface. Its main responsibilities are to (i) deal with geometry boundaries, (ii) map logical addresses onto physical addresses and guarantee the integrity—and eventual recovery in the face of crashes—of the associated mapping table, (iii) handle errors, and (iv) implement garbage collection.

4. Predictable Latency

We illustrate the potential benefits of application-specific FTLs in the following experiment. We use a 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 3. 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.

References

[1] BJØRLING, M., GONZALEZ, J., AND BONNET, P. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 359–374.