Programming Storage Controllers with OX

Ivan Luiz Picoli, Pınar Tözün, Andrzej Wasowski, Philippe Bonnet
IT University of Copenhagen

1 INTRODUCTION

Offloading processing to storage is a means to minimize data movement and efficiently scale processing to match the increasing volume of stored data. In recent years, the rate at which data is transferred from storage has increased exponentially, while the rate at which data is transferred from memory to a host processor (CPU) has only increased linearly. This trend is expected to continue in the coming years. Soon, CPUs will not be able to keep up with the rate at which stored data is transferred. The increasing volumes of store data compound this problem. The solution is to offload processing from host CPU to storage controllers [1].

Twenty years ago, Jim Gray wrote: *Put Everything in Future Disk Controllers (it's not "if", it's "when")* [7]. His argument was that running application code on disk controllers would be (a) possible because disks would be equipped with powerful processors and connected to networks via high-level protocols, and (b) necessary to minimize data movement. He concluded that there would be a need for a programming environment for disk controllers. In this paper, we follow-up on Jim Gray's prediction and reflect on initial lessons learnt programming storage controllers with OX.

In the 90s, pioneering efforts to develop Active Disks were based on magnetic drives. Today, renewed efforts fall in two groups. The first group combines Open-Channel SSDs [2] with a programmable storage controller integrated with a fabrics front-end. The second group integrates a programmable storage controller directly onto a SSD (e.g., ScaleFlux, NGD). For both groups, the programmable storage is either a Linux-based ARM processor or programmable hardware (FPGA).

In both cases, the following questions must be addressed: (1) Who programs the storage controller? (2) What is actually programmed? and (3) How are programs written?

2 PROGRAMMING ENVIRONMENT

We do not intend to go in depth with the definition of a programming environment for storage controllers. We present initial thoughts, which are meant to start a discussion based on the four questions we identified above:

Who programs the storage controller? Traditionally, SSDs storage controllers have been programmed by firmware engineers with background in NAND flash. The complexity of FTLs has led to decoupling front-end (FTL) and back-end (storage media) SSD management, respectively focused on software and hardware engineering. The advent of pblk [2] now enables Linux kernel engineers to program host-based front-end SSD management. Whether storage controller should be programmed by DevOps teams who program and configure their data center, or by FTL specialists is an open issue.

What is programmed? What does offloading processing to a storage controller actually mean? For NGX and ScaleFlux, offloading processing to a storage controller means installing application-specific code on top of a generic storage management layer. This is

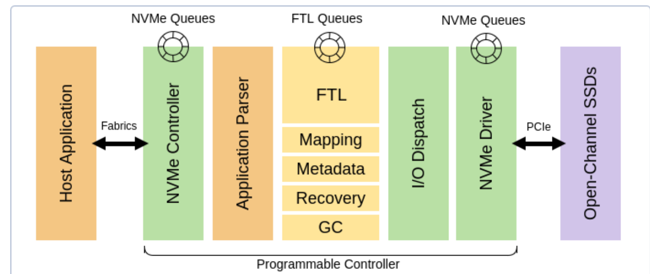

Figure 1: Abstract execution model: mapping from queues of NVMe commands on a logical address space to queues of NVMe commonds on the physical address space.

the model pioneered by Active Disks in the 90s. Storage devices no longer provide a fixed memory abstraction; they offer a communication abstraction based on a form of (asynchronous) RPC.

But why settle for a generic storage management layer? On flash-based devices, generic FTLs cause redundancies and missed optimization opportunities across layers on the data path [2]. Open-Channel SSDs make it possible to specialize FTLs on top of a well-defined abstraction of the physical address space.

Our position is that storage controller programming should be defined as the mapping from commands defined on a logical address space onto storage primitives defined over a physical address space, i.e., modifying a generic FTL rather than simply adding functionality on top of it. Examples of commands defined on a logical address space include `flush_sst` and `key_lookup` for an LSM-tree store [8], or `flush_log` and `read_page` for a log store [5]. Mapping such commands onto a physical address space require modifying the mapping, garbage collection or recovery functionalities of a traditional FTL.

In addition, programming a storage controller also creates opportunities for hardware acceleration.

How are programs written? The key question is whether a programming environment for storage controllers should be based on a general purpose programming language or on a domain-specific language? Put differently, the question is whether the equivalent of P4 can or should be defined for storage controllers[1].

In previous work, we mentionned match-action rules (inspired by OpenFlow) as a means to implement an FTL [3]. Today, we see little advantages in declarative FTL programming. Indeed programming an FTL requires efficient coordination of many dependent tasks affecting a persistent state.

The question is then what kind of high-level abstractions can be defined for FTL programming and whether they warrant the definition of a domain-specific language (as opposed to a collection of libraries or templates). In their seminal work on compositional

[1]This question was first formulated by T.Roscoe (ETHZ).

FTLs, Prof. Sang Lyul Min and his team, propose a log-based framework for representing FTLs [4]. Today, this is great for checking the correctness of an FTL, but it may also emerge as a powerful abstraction for modifying FTLs.

In the rest of this paper, we present the lessons we learnt programming a SoC-Based front-end to Open-Channel SSDs. More specifically, we present the abstract execution model corresponding to the mapping between application-specific commands and the physical address space exposed by Open-Channel SSDs. We adopt an approach where code is compiled and installed on the SoC. We present the OX template that we specialized for different applications.

3 ABSTRACT EXECUTION MODEL

We propose an abstract execution model for mapping commands defined on an application-specific address space onto storage commands on a physical address space. Figure 1 represents this abstract execution model that consists of 3 layers sandwiched between an NVMe controller exposed to the host, via fabrics, and an NVMe driver connected to one or several Open-Channel SSDs. We are currently evaluating how NVMe over fabrics compares to application-level RPC, e.g., gRPC, as host-interface. Our assumption is that NVMe provides higher performance which is key on the dataplane.

The 3 layers of the abstract execution model are (i) a parser adapted to the application specific commands and address space, (ii) an FTL responsible for the logical-physical mapping and the associated tasks of meta-data management, garbage collection and recovery, and (iii) a standard module for dispatching the commands generated by the FTL to the NVMe driver.

Because of space limitation, we limit ourselves to remarks about the role of the queues in this model. The storage controller is asynchronous in nature. It relies on a decoupling of command submission and notification of completion. Queues are the natural way to structure flows of submission/completions both at the interface between the storage controller and its environment, and within the FTL at the heart of the mapping between logical and physical representations.

4 FTL PROGRAMMING WITH OX

Pblk represents the first robust open-source implementation of an FTL. It is part of the Linux kernel [6]. Therefore, it naturally supports file systems. It must remain generic and robust. It is not meant to be modified to support application-specific address spaces. As a result, we designed and implemented a version of pblk in user-space. This FTL is denoted OX[2]. It adopts the modular structure of pblk. OX is designed as an FTL template, that can be readily modified to program storage controllers.

Figure 2 shows the OX architecture (including dependencies across modules) as well as the data path for both user and controller generated data movement. We represent API calls as generic reads or writes, even specialized calls are defined in practice. The OX FTL is composed of nine modules. We specialized it for two different applications: a log store and an LSM-Tree store. The log store defines each log flush as a transaction boundary. This impacts the write caching and checkpoint/recovery modules. The LSM-tree store modifies garbage collection so that it implements LSM-Tree compaction, as well as block provisioning where we experiment with various data placement policies. In both cases, we modified command parser and mapping table, while the bad block management module remained unchanged.

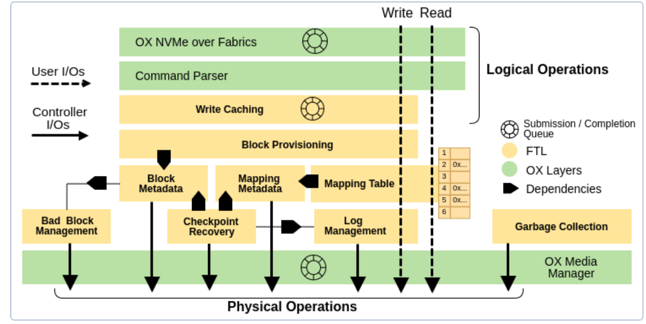

Figure 2: ox controller architecture

A thorough analysis of OX specialisation is a topic for future work. It is already obvious though that there is a need for tools to help check FTL correctness and debug concurrency bugs. In addition, there might be opportunities for developing languages that guide OX specialisation, e.g., the decomposition and mapping of values from the logical to the physical address space.

5 CONCLUSION

In this paper, we pointed to a range of issues related to programming storage controller. We introduce OX as an FTL template for programming SoC-based storage controller on top of Open-Channel SSDs. At its core, our approached is based on defining application-specific FTLs on storage controllers as a means to offload processing from the host. This way, we leverage near-data processing as a means to collapse layers within the I/O stack, in contrast to existing approaches that ship functionalities within the legacy layered architecture. Evaluating whether an approach dominates the other is a topic for future work. More generally, the key question is whether programming storage controller will remain a task for specialized experts, or whether it will be accessible to a larger class of DevOps programmers thanks to high-level abstractions and languages.

REFERENCES

[1] Balasubramonian, R., Chang, J., Manning, T., Moreno, J. H., Murphy, R., Nair, R., and Swanson, S. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro 34*, 4 (July 2014), 36–42.
[2] 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.
[3] Bjørling, M., Wei, M., Madsen, J., Gonzalez, J., Swanson, S., and Bonnet, P. AppNVM: A software-defined, application-driven SSD. In *Non-Volatile Memories Workshop* (San Diego, CA, United States, 2015).
[4] Choi, J.-Y., Nam, E. H., Seong, Y. J., Yoon, J. H., Lee, S., Kim, H. S., Park, J., Woo, Y.-J., Lee, S., and Min, S. L. HIL: A Framework for Compositional FTL Development and Provably-Correct Crash Recovery. *ACM Transactions on Storage (TOS) 14*, 4 (2018).
[5] Do, J., Lomet, D., and Picoli, I. L. High Write IOPS via Log Structuring in an SSD Controller. Tech. rep., Submitted to Publication, 2018.
[6] Gonzalez, J. pblk—the OCSSD FTL. In *Linux FAST'18* (Oakland, CA, 2018), USENIX Association.
[7] Gray, J. Put Everything in the Disk Controller. *Talk at NASD workshop* (1998).
[8] Picoli, I. L., Bonnet, P., and Tözün, P. LightLSM: LSM-Specific SSD Management. Tech. rep., Submitted to Publication, 2018.

[2]https://github.com/DFC-OpenSource/ox-ctrl/wiki