LSM Management on Computational Storage

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

Philippe Bonnet
IT University of Copenhagen
phbo@itu.dk

Pınar Tözün
IT University of Copenhagen
pito@itu.dk

ABSTRACT

LSM-trees have emerged as the write-optimized index of choice for key-value stores and relational database systems. LSM-trees typically rely on a storage manager on top of a file system for storing data on Solid-State Drives (SSDs). The I/O path thus comprises four layers, each independently managing similar indirection, journaling, and garbage collection mechanisms. Such overhead is increasingly problematic. First, the advent of microsecond-scale SSDs makes it necessary to streamline the I/O software stack. Second, the increasing performance gap between storage and CPU makes it necessary to reduce CPU storage overhead. A solution is to collapse LSM, file system, and SSD management layers into a single software layer embedded on computational storage. Specific commercial solutions are already available. In this short paper, we describe the design space for LSM management on computational storage.

ACM Reference Format:
Ivan Luiz Picoli, Philippe Bonnet, and Pınar Tözün. 2019. LSM Management on Computational Storage. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, the rate at which data can be read from or written to storage and network devices has increased exponentially, while the rate at which data can be read from or written to the memory of 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 data moves through storage and network. Computational storage, or near-data processing, architectures are re-emerging to tackle this issue by offloading low-level storage processing from host CPU to storage controllers [8]. At the same time, the emergence of Solid State Drives (SSDs) with microsecond-scale latency makes it necessary to streamline the storage stack to avoid redundancies and to bypass bottlenecks.

The confluence of these two trends motivates a profound redesign of the storage stack. Two of the key questions in this context are: (a) How to streamline existing layers on the I/O path? and (b) What components of the storage stack can be offloaded from the host CPU to the storage controller?

As initially pioneered by active disks, it makes sense to offload access methods on the storage controller to minimize data movement in database systems. In particular, LSM-trees constitute ideal candidates for collapsing layers on the storage stack in order to streamline the I/O path. Indeed, LSM-trees and Flash Translation Layers both rely on immutable storage structuctures associated with a form of garbage collection. Initial work by Baidu [9] and CNEX Labs [7] focused on supporting an LSM-based key-value store on open-channel SSDs by implementing an LSM-Tree specific FTL on the host. These solutions are defined in the context of traditional host-SSD architectures. On the other hand, recently, Samsung announced KV-SSD, an SSD equipped with an LSM-tree specific FTL [3]. Can these solutions be applied in the context of computational storage? What are the possible architectures for LSM management on computational storage? What is the associated design space? We tackle these questions in this short paper.

2 ARCHITECTURE

Let us briefly present an overview of the architectural impact of computational storage, illustrated in Figure 1. In traditional architectures (Figure 1a), a KV-store implemented on the host relies on the file system and block layer from the operating system as well as a generic FTL embedded on the SSD. Open-channel SSDs (Figure 1b) [2, 4] make it possible to specialize the FTL for LSM Management [7]. While Figure 1b improves on Figure 1a by streamlining the data path, it does so by increasing the load on the host. In order to reduce the load on the host, it is necessary to offload processing to the SSD.

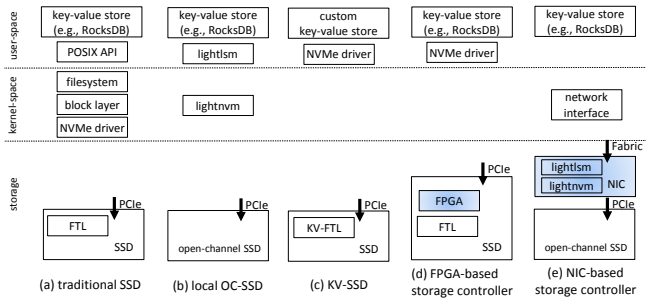

Figure 1: State-of-the-art architectures with (a) traditional SSDs and (b) open-channel SSDs compared to (c) a custom KV-store on a specialized KV-SSD and upcoming architectures based on (d) FPGA-based computational storage as well as (e) NIC-based computational storage.

Samsung recently announced KV-SSD, an SSD equipped with an LSM-specialized FTL (Figure 1c). KV-SSD requires a specialized NVMe driver on the host, which is accessed from a custom KV-store.

An alternative to this specialized solution is to consider general-purpose computational storage. So far, two types of computational storage have emerged. The first type integrates a programmable storage controller directly onto an SSD (Figure 1d). Commercial products are available in this space (e.g., Samsung SmartSSD, ScaleFlux, NGD). The programmable storage is either a Linux-based ARM processor or programmable hardware (FPGA). The second type combines Open-Channel SSDs with a programmable storage controller integrated with a Network Interface Card (Figure 1e). Commercial products are available from Broadcom or Mellanox, while research products include the DFC Platform [1]. While both architectures support functionality offload from the host, the feasibility to collapse layers and streamline the I/O path on the Figure 1d architecture is unclear as some products only expose storage media to the storage controller via a proprietary generic FTL.

In the rest of this paper, we assume the Figure 1e architecture to discuss the design space.

3 DESIGN SPACE

In this section, we highlight the opportunities and trade-offs associated with LSM management on computational storage with a specialized FTL. While previous work has focused on leveraging LSM-tree compaction as a form of garbage collection, here we focus on the challenges of interface, mapping, and journaling.

An LSM-tree is composed of levels. The top level is an in-memory buffer called *mem-table*, while the rest of the levels are kept as files on secondary storage called *Sorted Sequence Table (SST)*. We refer readers to Dayan et al. [6] for a thorough description of LSM-tree internals.

3.1 Interface

An initial design decision is whether LSM management should be fully offloaded to computational storage, or whether one should split the functionalities. An obvious split is to offload SST management on computational storage and leave mem-table management on the host. The rationale is two-fold. First, offloading SST management on computational storage makes it possible to handle compaction within storage, thus avoiding unnecessary data movement to the host, and enabling compaction to be the sole form of garbage collection in an LSM-specific FTL. Second, keeping mem-tables on the host leverages locality to minimize data movement.

An LSM-based storage manager manages auxiliary data structures in addition to mem-tables and SST. For example, RocksDB has WAL for recovering mem-tables and manifest files to keep track of SSTs with their corresponding key range. Ideally, those data structures are maintained in persistent memory with an in-memory data structure for the manifest contents for faster access. Alternatively, computational storage should support storing/retrieving these data structures. On completion of a command flushing an SST to storage, the WAL entry for the given SST can be discarded. The ideal management of these data structures over computational storage has to be studied.

3.2 Mapping

Any FTL maps a logical name space onto the physical address space. FTLs specialized for LSM management must map SSTs onto the physical address space. In open-channel SSDs, the physical address space is represented as a hierarchical space of groups, parallel units (PUs), chunks, and logical blocks. The minimal unit of read and write is a logical block. This is a clean abstraction of the physical space used by any FTL. We consider that the maximum size of an SST is fixed in such a way that it corresponds to the product $number\_of\_parallel\_units * chunk\_size$[1]. On an Open-Channel SSD with 32 PUs and 1MB chunks, we get a maximum SST size of 32MB, with 4KB as the unit of reads.

Such an SST size gives us a range of mapping options. The fist option is a horizontal mapping, where the contents of each SST is striped across all PUs. This option maximizes write throughput, but it may cause interference that will negatively impact read latency, as any read will interfere with any write of an SST. The second option is a vertical mapping, where the contents of each SST is striped across the PUs of a given group. This option trades write throughput for improved concurrency as a maximum of $number\_of\_groups$ reads and SST writes can be executed concurrently without interference. A third option is a hybrid mapping, where the contents of each SST is striped across PUs in a subset of all groups. Such a mapping might offer a compromise between write throughput and read latency. In vertical and hybrid mapping, the groups on which SSTs are striped are picked in a round-robin fashion.

A thorough investigation of the trade-off between write throughput and read latency for LSM-tree management on computational storage is needed.

3.3 Journaling

The state of the software embedded on computational storage must be re-constructed whenever it starts up or recovery is needed. It is natural to consider the modifications of this state as transactions for which the durability property is guaranteed. The transactional nature of the FTL for traditional SSDs has been observed in prior work [5].

A first option to ensure durability is write-ahead logging. Specific portions of the storage space (e.g., specific chunks) must be reserved for this purpose. A second option is to use the out-of-bound area on flash storage to store metadata (e.g., reverse mapping in a traditional FTL). A third option is to store snapshots at fixed locations in the address space.

The key tradeoff here is between the speed of recovery and space amplification, which has to be studied to design efficient recovery procedures for LSM-trees on computational storage.

4 CONCLUSION

We have given an overview of the fundamental trade-offs for the design of LSM management solutions on computational storage. We have identified the design of interfaces, mapping, and journaling as open issues. Exploring this design space on various forms of computational storage is an interesting topic for future work.

REFERENCES

[1] DFC Platform. https://github.com/DFC-OpenSource.
[2] Open-Channel SSD. https://openchannelssd.readthedocs.io/en/latest/.
[3] Samsung KVSSD. https://github.com/OpenMPDK/KVSSD.

[1] The size of each SST might vary, but it will not exceed the given limit.

[4] M. Bjørling, J. González, and P. Bonnet. LightNVM: The Linux Open-channel SSD Subsystem. In *FAST*, pages 359–373, 2017.
[5] 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 Transactions on Storage (TOS)*, 14(4), 2018.
[6] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*, pages 79–94, 2017.
[7] J. González, M. Bjørling, S. Lee, C. Dong, and Y. Ronnie Huang. Application-Driven Flash Translation Layers on Open-Channel SSDs. In *NVMW*, 2016.
[8] J. Gray. Disk Architecture Directions: Put Everything In the Disk Controller. In *NASD*, 1998.
[9] 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.