LightStore: Software-defined Network-attached Key-value Drives

Chanwoo Chung
Massachusetts Institute of Technology
cwchung@csail.mit.edu

Jinhyung Koo
DGIST
jhk5361@dgist.ac.kr

Junsu Im
DGIST
junsu_im@dgist.ac.kr

Arvind
Massachusetts Institute of Technology
arvind@csail.mit.edu

Sungjin Lee
DGIST
sungjin.lee@dgist.ac.kr

Abstract

We propose LightStore, a key-value flash store, as a substitute for x86-based storage servers. A LightStore node has a low-power embedded-class processor, a few gigabytes of DRAM and a few terabytes of NAND flash, and can be directly connected to a network port in a datacenter. A large-scale distributed storage cluster can be formed simply by adding more LightStore nodes to the network. Applications in a datacenter can take multiple "software-defined" views of LightStore stores via thin LightStore adapter layers, which translate conventional KV, YCSB, block, and file accesses to KV ones for LightStore. LightStore is estimated to be 2.0x power-efficient and 2.3x space-efficient than an x86-based all-flash array system of the same capacity. Experimental results on our LightStore prototype show that 1) the LightStore node performance is comparable to an x86 server with a single SSD; 2) a four-node LightStore cluster exhibits up to 7.4x better ops/J than an x86 server with four SSDs.

CCS Concepts • **Information systems** → **Key-value stores**; *Cloud based storage*; • **Computer systems organization** → *Embedded systems*; • **Hardware** → External storage.

Keywords key-value flash drive; LSM-tree; hardware FTL

ACM Reference Format:
Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. 2019. LightStore: Software-defined Network-attached Key-value Drives. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304022

ASPLOS'19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304022

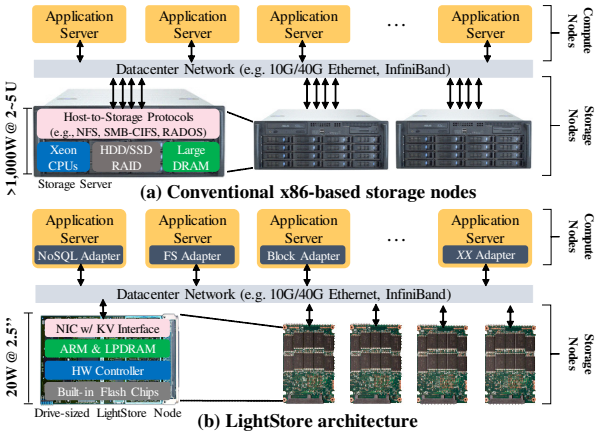

Figure 1. Conventional storage system and LightStore

1 Introduction

A great deal of digital data is generated every day. These data are stored in the cloud and processed by large-scale distributed systems, running a myriad of applications in datacenters. A typical datacenter architecture is shown in Figure 1(a), where storage servers are responsible for managing the data and providing application servers fast access to the data over high-speed networks. Storage servers represent a significant capital and operating cost of a datacenter. This paper is about a new architecture for storage systems, which is more efficient in terms of cost-power-performance.

A typical storage server in the current market comprises tens of Terabyte-HDDs, hundreds of Gigabytes of DRAM and 1 or 2 Xeon processors [44, 45]. Large DRAM is used to compensate for the poor latency and random-access performance of HDDs. It can be also used for application level caching. The server internally employs RAID to aggregate the capacity and throughput of multiple slow HDDs. Powerful CPUs are needed to run software layers including distributed and local file systems and to support storage-access protocols (e.g., NFS [56], SMB-CIFS [42] and RADOS [63]). This architecture has evolved to embrace NAND flash-based solid state drives (SSDs), which consume much less power and offer better I/O performance than HDDs. A representative in the market is all-flash array (AFA) systems [20, 33, 46].

A naive replacement of storage media in this architecture, however, does not allow us to fully exploit the characteristics of NAND flash. First, HDD-based systems require a large amount of DRAM buffer, specially for random accesses, because of a huge mismatch between the network and disk bandwidth. A modern enterprise-class NVMe SSDs deliver up to 1 million IOPS of 4 KB random reads [54, 59], and therefore, the impact of large DRAM cache on improving random I/O is rather marginal. Second, in HDD-based systems, disks are so slow that they have to be grouped together to fully utilize the internal bus or the network bandwidth. Unlike HDD-based systems, in SSD-based systems the network easily becomes a bottleneck [33]. Latest NVMe SSDs offer 10 GB/s of sequential reads [59], surpassing the maximum bandwidth of a 40 Gb Ethernet (40GbE) port and nearly matching the bandwidth of a 100GbE port. Thus, aggregating SSDs to increase the bandwidth does not make sense. Third, storage servers usually have a deep software stack, which may result in degraded I/O performance, whose relative effect is more noticeable on SSDs due to their high bandwidth [9, 34, 39, 66, 67].

The above observations have led us to propose, *LightStore*, a new storage architecture with a KV interface, and whose SSD-sized nodes can be plugged into datacenter network ports directly, as depicted in Figure 1(b) [10]. Such a KV store is deployed using thin software adapters on the client side to translate client requests to KV requests, supporting the interoperability of existing protocols. These adapters not only provide "software-defined" storage views (e.g., block stores) of LightStore nodes, but eliminate the burden to support other protocols on the storage side. Thus, by simplifying the software stack and minimizing its resource requirement, we can implement LightStore using embedded-class processors and a small amount of DRAM in the SSD form factor, and simultaneously, deliver the high throughput and low latency of the flash to the network. The performance of LightStore scales with the number of network ports because each port is connected to a LightStore node. With this design, we expect LightStore to be 2.0x power-efficient and 2.3x space-efficient than commercial AFA systems of the same capacity.

Directly attaching a storage disk to the network, a central idea in LightStore, was proposed as an HDD-based Network-Attached Secure Disk (NASD) [24]. The Kinetic HDD [58] has extended the idea further. The main difference is that the random accesses in flash are 2-3 orders of magnitude faster than HDDs (e.g., 10 *m*s vs 100 μs) and, consequently, much more computing power is required to deliver the flash bandwidth to the network. Another difference is that flash devices require a lot of bookkeeping, which was not required for HDDs. Running such tasks on embedded cores while delivering high throughput to the network is a real challenge.

The most crucial technical issue in realizing the LightStore idea is the design of a lightweight KV store. Although there are existing persistent KV stores based on an LSM-tree [47] (e.g., RocksDB [21]), they are mostly used and optimized for x86 CPUs. For example, until 2017, RocksDB did not support arm64-linux cross-compilation with their build script. As we discuss in Section 4.2, the performance of RocksDB on the ARM platform is totally inadequate to deliver full raw NAND performance to the network ports. Using a hash-based KV design as in Samsung's KV-SSD [55] and KAML [28] could be a reasonable choice because it is simpler than LSM-trees. A hash-based design, however, is not capable of handling scans and range queries efficiently since KV pairs are *not sorted*. Thus, a hash-based KVS is not appropriate to support various applications using adapters. To address these issues, we have developed a new KV store based on the LSM-tree from scratch with optimization suited to embedded cores.

An SSD controller includes a relatively powerful embedded-class processor (e.g., 1 GHz ARM quad-cores) and 1-16 GB of DRAM [26, 52], and these might be fast enough to run our lightweight KV store. In typical SSDs, however, this hardware is dedicated to run sophisticated software known as a flash translation layer (FTL) which manages flash chips and delivers the internal bandwidth of them to the storage interface (e.g., PCIe). The FTL hides the physical characteristics of flash and makes SSDs behave like HDDs. We have exploited the append-only write nature of the LSM-tree and used the interface defined in AMF [39] to make the FTL so simple that it could be implemented completely in hardware. This frees up embedded resources which can be used to implement the LSM-tree algorithm as well as the network stack.

This paper makes the following contributions:

- **LightStore Architecture**: a new low-cost drive-sized key-value flash store that can be connected directly to the network and deliver the full device bandwidth to the network;
- **4-node Prototype Implementation of LightStore** using Xilinx ZCU102 boards [64] and custom flash cards [40];
- **Hardware FTL**: a complete HW implementation in FPGA;
- **Performance Studies** to show that LightStore performance scales linearly with the number of nodes and achieves up to 7.4x better ops/J than x86 servers;
- **Case Studies of Software-defined Stores** to confirm the feasibility of virtualizing various stores – native KV, cloud, block and file stores – using adapters.

In Section 2, we review prior studies. After giving an overall design of LightStore in Section 3, its detailed architecture and adapter implementation are described in Sections 4 and 5. Section 6 estimates the operating cost benefits in a datacenter. Section 7 presents results derived from the LightStore prototype. Section 8 concludes with future directions.

2 Related Work

We introduce prior studies on storage designs that have inspired us to design and implement a LightStore cluster. We have already discussed the prior work on Network-attached Storage [24, 58] and KV-SSDs [28, 55] in Introduction.

Flash-based Key-value Cache: Serious attention has been devoted to building key-value caches using flash. Representatives include BlueCache [66], FlashStore [18], FatCache [62] and SkimpyStash [19]. The implementations of individual techniques may be different from each other, but the common theme is to use flash as a replacement for DRAM cache where persistency is less of a concern. The challenges in building a KVS as opposed to a KV cache are different and KVSs require more complex data management.

Key-value SSD: Samsung has introduced a KV-SSD [55] which combines a KV frontend with the traditional FTL [1, 25], allowing both KV and block accesses. KAML [28] is another KV-SSD that optimizes the performance by integrating its novel FTL and KV store. Such devices plug into the *local bus* of application servers and are intended to enhance its *KV* access performance only. Furthermore, they do not support operations on ordered keys due to hash-based algorithms.

I/O Stack Optimization for Fast Storage: There have been several studies to refactor existing I/O stacks suitable for fast storage [9, 10, 30, 36, 39, 49]. Software-Defined Flash (SDF) [49] enabled datacenter applications to access individual channels inside an SSD, bypassing deep I/O layers. This makes it possible for applications to better control underlying flash media with less overhead. Application-Managed Flash (AMF) [39] took a similar approach. It provided better interoperability with log-structured applications by exposing a typical block I/O-like interface with append-only. ReFlex [36] optimized both network and storage stacks to offer consistent I/O latency to applications over the network. QuickSAN [9] eliminated software and hardware overheads to access SSDs in a storage area network. Since these above approaches were targeted for server-class systems, they did not take into account issues that arise when drive-sized embedded nodes are connected to applications servers directly. Finally, BlueDBM [29, 30] suggested optimizing the storage stack with flash and FPGA for near-data processing.

Low-power Flash Cluster: Previous works, such as FAWN [3], CORFU [6] and Gordon [8], have explored low-power architectures using a cluster of cheap and low-power embedded systems with flash. FAWN and CORFU focused on how to group wimpy nodes with slow flash (e.g., 40 MB/s CompactFlash) to realize high aggregate throughput at low power. In their system, the network was not a bottleneck while in ours optimizing the access to network is a central concern. Gordon [8] used a low-power flash cluster for near-data processing to improve the performance of MapReduce applications [17]. In Gordon also, the network was not a bottleneck. It used wimpy processors near flash to offload some server work, and is not a storage system.

Other Non-volatile Memory: Many researchers have proposed using other types of non-volatile memory, such as PRAM and MRAM, for persistent storage to boost application performance [48, 61, 65, 69]. However, their limited capacity make it a improper substitute in large-scale storage systems.

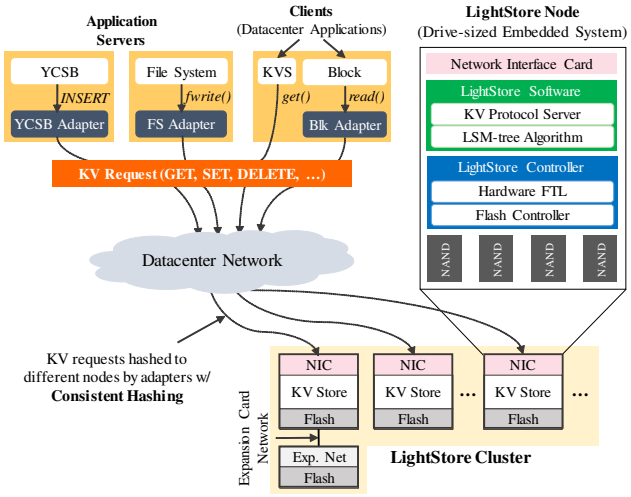

Figure 2. Overall architecture of LightStore

3 LightStore Overview

Figure 2 illustrates the two main components of a LightStore system: 1) a cluster of LightStore nodes to support KV operations and 2) application servers with LightStore adapters to provide compatibility with typical applications. The adapters translate various storage protocols to KV requests, providing conventional storage views to datacenter applications. To emulate a block store, for example, block I/O requests from applications using POSIX APIs (e.g., `fread()`) are eventually translated to KV requests by a block adapter and then sent to LightStore nodes. We have implemented a separate adapter for each of YCSB, block and file applications (Section 5).

No matter what types of stores are emulated, the LightStore adapters should be able to decide a designated LightStore node to which the translated KV request is sent. This is done by borrowing the concept of consistent hashing [32], which is widely used in distributed systems (e.g., Redis [57] and Swift [23]). Using the key of a KV request, the adapters compute a hash key that decides a target LightStore node. If required, it sends replicas to different nodes in the same manner, which would ensure data availability and persistence against node failures.

A LightStore node comprises a *LightStore controller* (Section 4.1) and *LightStore software* (Section 4.2). Similar to a typical SSD controller, the LightStore controller is in charge of managing an array of NAND flash chips organized in channels and ways [1]. It implements a simplified FTL that is designed to perform address remapping, wear-leveling and bad-block management, all exclusively in hardware (HW FTL). This hardware implementation of an FTL frees up ARM cores to run the LightStore software. However, since the LightStore hardware provides an append-only block interface, the software must translate overwrites into appends.

The main responsibilities of LightStore software are to handle KV requests and keep KV pairs persistently. We have chosen a log-structured merge-tree (LSM-tree) [47] to build

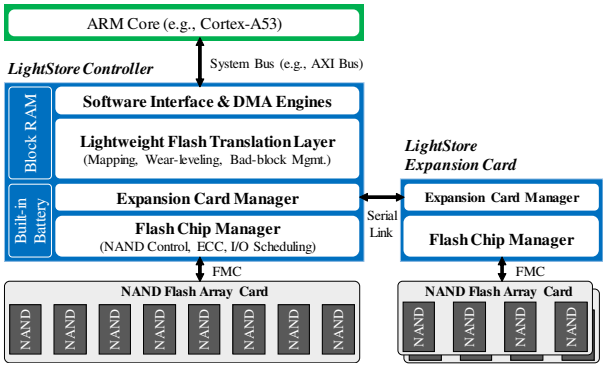

Figure 3. LightStore controller with an expansion card

LightStore software because it writes data in an append-only manner, well-suited for the LightStore controller and flash. In addition, LSM-tree is one of the most popular data structures used to implement a persistent KVS. We have developed the LSM-tree software from scratch, with optimizations aimed at embedded processors (see Section 4.2).

Both the throughput and the capacity of a LightStore cluster increase linearly by attaching more nodes to the network. The LightStore approach, however, does require more network ports and associated facilities (e.g., switches) for proper scaling. This is also true for x86-based systems when it adds more servers to the network. To support low-cost capacity-only scaling, we provide each LightStore node the capability to connect to more NAND chips using inexpensive expansion cards, which is just like adding more drives in a server.

4 Design of LightStore Node

In this section, we explain the detailed design and implementation issues of the LightStore controller and software.

4.1 LightStore Controller

There are four main building blocks in the LightStore controller. Figure 3 shows how these components are connected. Note that off-loading flash management duties into the hardware FTL plays a key role in LightStore in letting the rest of the software run at high speed.

Software Interface: It accepts block I/O commands (`READ`, `WRITE`, and `TRIM`) from the LightStore software, delivers them to the HW FTL, manages DMA engines [35] to transfer data from/to DRAM, and replies back once the operation has completed. The LightStore controller exposes an array of *append-only* 8 KB logical blocks to the software, and each I/O command carries a logical block address (LBA) which is different from the physical address in the flash array. The mapping of a logical block to a physical flash page is the responsibility of the hardware FTL, which is described next.

Hardware FTL: LightStore borrows the idea of a simplified FTL from AMF [39], which restricted the software to write data in an append-only manner. The primary responsibility of the simplified FTL is to map the logical block address

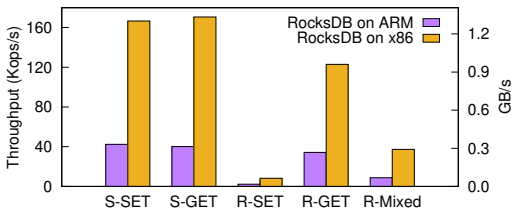

Figure 4. Performance comparison of RocksDB on ARM and x86

(LBA) used by the software to a physical flash address (e.g., channel, block and page) using block-based mapping and page offsets [4]. We map consecutive LBAs to different channels to maximize the internal bandwidth usage. If there is no valid mapping for the logical flash block initially, a free flash block is allocated to the logical block. When choosing a new flash block, hardware ensures that no bad block is allocated. We maintained two data structures for this purpose: a block mapping table and a block status table. The first table stores whether a logical block is mapped, and, if so, its corresponding physical block address. The second table keeps the status and program/erase (P/E) cycles of flash blocks. The two tables require only 1 MB per 1 TB, while a conventional FTL required at least 1 GB DRAM per 1 TB flash [26, 52]. The two tables survive even when power fails (see Section 4.2).

AMF's FTL (AFTL) was implemented as software that directly issued I/O commands with flash addresses and run on x86. We have exploited the simplicity of AFTL to implement it completely in hardware for fast translation speed. Hardware FTL translates LBAs in 4 cycles (if mapped) or 140 cycles (if not yet mapped). At 200 MHz clock frequency, LightStore HW FTL adds at most 700 ns which is less than 1% of flash access latency.

Expansion Card Manager: It plays an important role in the capacity scaling of the LightStore node. A master LightStore node and its expansion cards are connected to form an expansion card network which is based on a simple protocol with 0.48μs per-hop latency [30, 31]. I/O commands are routed to an appropriate flash chip manager via the network. The network controller exposes a unified address space to the software, making an illusion that all the NAND chips in the master and expansion cards form a single flash array.

Flash Chip Manager: It directly interacts with NAND chips – it forwards commands translated by the HW FTL to the chips, maintains multiple I/O queues, and performs scheduling to exploit parallelism of multiple chips. It also performs error correction using ECC bits. Thus, the controller provides us with a robust and error-free access to flash.

4.2 LightStore Software

Before designing LightStore software, we ran RocksDB [21] on the LightStore hardware to see if an existing persistent KVS implementation was feasible. Figure 4 compares the performance of RocksDB running on ARM and x86 CPUs. We used five synthetic workloads (see Table 3, Section 7) that

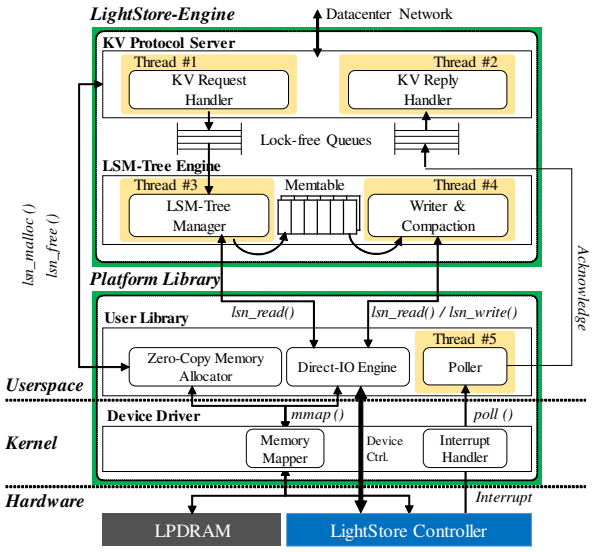

Figure 5. LightStore software architecture and thread assignments

are used widely to assess KV stores. Except for the CPU and DRAM (see Table 2), the experimental setups were identical; we used the same software layers and the same SSD, Samsung 960 PRO. One can immediately see that there is a noticeable drop (3.6x-4.2x) in the performance on ARM compared to x86. Using profiling tools like `perf` [16], we identified three main bottlenecks in running RocksDB on ARM:

1. *Excessive Memory-copy Overhead:* When we break down CPU cycles, `memcpy()` calls account for up to 30% of the total CPU cycles. This is due to lower bandwidth of DRAM and memory bus as well as the lack of cache-coherent DMA support in some embedded architectures.
2. *High Context Switch Overhead:* RocksDB typically spawns more than 20 threads for simultaneously processing user requests, flush and compaction. Since a typical embedded CPU has 2 to 4 cores, the cost of context switch and lock management dwarfs the benefits of multi-threading.
3. *Deep and Sophisticated Software Stack:* RocksDB runs atop kernel layers, such as a page cache, a file system and a block I/O layer. To expose KV interfaces, it also requires running a 3rd-party KV server. Running RocksDB with those layers is a serious burden on the embedded cores.

Keeping these problems in mind, we have designed the LightStore software, as illustrated in Figure 5, to include the following optimizations: 1) a zero-copy memory allocator to avoid data copy between layers; 2) lock-free queues to lessen context switch overheads; and 3) bypass intermediate kernel layers including a file system and even a page cache to directly access the hardware FTL. Additionally, we have further optimized the LSM-tree itself by 4) decoupling keys from KV pairs to build a small "keytable" that allows metadata-only compaction to speed up writes [41]; 5) use bloom filters [60] and fully cache the keytable in DRAM to accelerate reads.

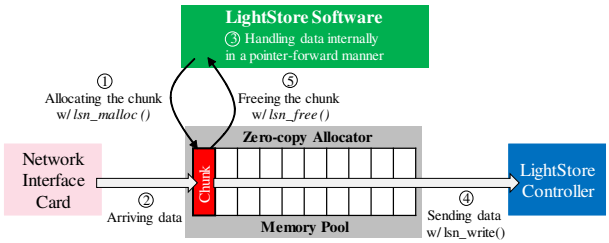

Figure 6. Handling of data with the zero-copy memory allocator

The platform library provides LightStore-Engine with a set of APIs to directly access the flash storage and the network with zero page copies and with minimal OS kernel involvement. The LightStore-Engine is responsible for handling KV requests over the network (KV protocol server), and its lightweight LSM-tree engine is in charge of retrieving and storing key-value pairs from/to the flash. The LightStore software spawns a single process with only five threads. This number is chosen to balance the workload among software modules as well as to minimize the context-switch cost. Figure 5 shows how threads are assigned. To glue one thread to another, lock-free queues are used, instead of standard queues with locks. In addition to these threads, the kernel spawns many other threads for its internal tasks (e.g., TCP/IP and interrupt handling). Using lock-free queues is effective in mitigating the penalty of frequent context switches.

4.2.1 Platform Library

The zero-copy memory allocator maintains a pool of 8 KB memory chunks that enables us to carry data from the network device to the LightStore controller (and vice versa). The pool is created and DMA-mapped by the LightStore kernel driver and the hardware device can directly access memory chunks. The kernel driver exports them through the `mmap()` system call so that the LightStore software running in the user space can access memory chunks. As a result, the LightStore-Engine and the device directly share and manage the same memory pool. For its easy use, the zero-copy allocator encapsulates the DMA-mapped pool and exposes two memory manipulation functions, `lsn_malloc()` and `lsn_free()`. As illustrated in Figure 6, when receiving data from the network, LightStore first reserves a free chunk using `lsn_malloc()` (①). As in the zero-copy socket [14], data streams arriving from the network are directly stored in the chunk (②). The same DMA-mapped chunk can be sent to the flash directly (④). As will be discussed in 4.2.2, the LightStore-Engine deals with memory chunks in a pointer-forwarding manner (③). Thus, from the network to the flash, no copy of a page is made. `lsn_free()` is invoked to release the chunk if no longer used (⑤).

LightStore-Engine accesses the flash directly through a direct-IO engine. The LightStore controller is managed using memory-mapped registers and the register addresses are exposed to the user space by the kernel driver. This makes it

possible for the direct-IO engine to implement device control commands in the user-space. The direct-IO engine exposes simple commands: the two important ones are `lsn_read()` and `lsn_write()`, which perform a read and write operation on a specified LBA, respectively. Interrupts from the hardware are delivered to the poller thread through `poll()`. In this way, only minimal kernel layers are involved.

4.2.2 LightStore-Engine

This section details how the KVS protocol server and the LSM-tree engine work, along with a data persistence issue.

KVS Protocol Server: The KVS protocol server is based on TCP/IP and designed to saturate the effective throughput of a 10GbE interface used in our experiments. As a KV protocol, we use a derivative of RESP (REdis Serialization Protocol) for its wide usage [57]. Currently, we have implemented only selected operations needed to show the feasibility of emulating various data stores on LightStore, which include `SET`, `MSET`, `GET`, `MGET` and `DELETE`. Some operations like range queries and iterations are not implemented yet but are straightforward to implement since KV pairs are already sorted in the LSM-tree. Upon receipt of a KV request header, the KVS request handler allocates memory chunks. The allocated chunks are used to keep incoming values (`SET`, `MSET`) or outgoing values (`GET`, `MGET`). It then forwards keys and chuck pointers to the LSM-tree engine. The KV-reply handler gets a response from the LSM-tree engine once the command is processed and replies back to the client. Finally, the memory chunks are released.

LSM-tree Engine: The LSM-tree engine does not rely on a file system or a block I/O layer. It does not exploit even the kernel's page cache. Since hot data is mostly cached in application servers or caching layers (e.g., memcached), benefits from caching with small node DRAM are marginal. Instead, using the memory for other purposes like metadata caching is more beneficial. Regardless of original value sizes, the current LSM-tree engine deals with all the values in the 8 KB unit. This may cause internal fragmentation, but is not a serious issue in our use cases; LightStore is intended to replace data stores where the object size is large [7].

The handling of `SET` and `MSET` from the KV protocol server is processed as follows: the values of the requests are stored in a small buffer in DRAM, which is often called a *level-0* (L_0) *memtable* [47]. The keys of the requests are separately maintained in a L_0 *keytable* according to their order. To avoid data copy between the KV server and LSM-tree engine, the L_0 memtable is built like a pointer table that augments multiple 8 KB chunks from the KV server as sorted in the keytable. When L_0 becomes full, L_0 memtable is evicted to a *persistent* L_1 memtable in the flash. Since a large number of buffered chunks are written to the flash sequentially (L_0 is sorted), we not only satisfy the append-only write restriction, but fully utilize the parallelism of multiple flash chips. After flushing the chunks, the keytable is updated to point to values

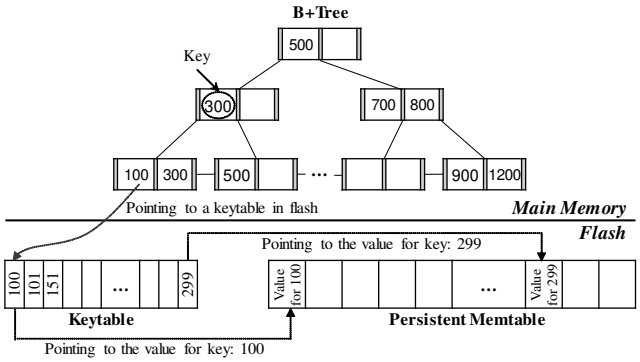

Figure 7. Simplified illustration of three data structures, a B+tree, a keytable and a persistent memtable in LightStore-Engine

in the persistent memtable and is also written to the flash (L_1 keytable). This decoupling of keys and values is useful to mitigate compaction overheads later. To keep track of keytables in the flash, the LSM-tree engine maintains a B+ tree in DRAM. Figure 7 illustrates the three components (a B+ tree, a keytable and a persistent memtable), showing how values for given keys (e.g., 100 and 299) are indexed.

The persistent memtables are organized hierarchically in multiple levels (e.g., L_1, L_2, ..., L_n) in increasing sizes [47]. As in L_0, once L_{n-1} becomes full, key-value pairs in this level must be flushed to L_n. This process is called compaction [47, 51] because it requires us to read all the key-value pairs from the old L_{n-1} and L_n, sort them and write back the sorted pairs to the new L_n. The compaction process involves many read and write operations. To lower this overhead, we adopt the idea of metadata sorting [41]. Instead of physically rearranging values in the flash, the LSM-tree engine only sorts keytables in L_{n-1} and L_n and writes the result to the flash. The pointers to values in keytables do not need to be updated because the physical locations of values are not changed. This greatly reduces the I/Os cost of compaction. Note that invalidated and deleted key-value pairs are removed from the flash as part of the compaction process. It is thus unnecessary to run a separate FTL garbage collection thread to reclaim free space.

To retrieve a value requested by `GET` or `MGET`, we need to look up keytables in all the levels (from L_0 to L_n) since the desired key can be located in any layer. This inevitably requires multiple keytable lookups, resulting in the degradation of `GET` latency [47, 60]. We use two methods to address this: 1. use a bloom filter to avoid looking up levels not having the desired value [15]; and 2. caching keytables in the main memory. Our LSM-tree engine does not rely on the OS page cache and minimally uses the main memory for other kernel modules. Moreover, the LightStore software requires only 82 MB of DRAM for a B+ tree, lock-free queues and L_0 memtables, leaving much room for caching keytables. Furthermore, LightStore does not maintain a huge mapping table for address remapping because an FTL is unnecessary. The

size of DRAM required to keep the entire keytables is about 1 GB per 1 TB flash, and modern SSDs already employ larger than 1 GB DRAM per TB of flash [26, 52]. Consequently, the flash is read only to retrieve values not metadata.

Data Persistence/Consistency: Maintaining persistence and consistency of data in case of power failure is important in any data storage systems. Many systems use a write-ahead log (WAL) for data consistency; however, this incurs considerable overhead owing to double writes. Some x86-based storage appliances use expensive external batteries to save the host state [43]. Enterprise SSDs are equipped with built-in capacitors by default, thereby providing enough time to flush out ephemeral data structures as well as pending data to non-volatile media. By leveraging this, LightStore achieves the high level of persistence/consistency without issuing any additional I/Os as well as requiring external batteries.

5 LightStore Adapter

In order to prove the concept of a software-defined store based on a KV interface, we implement two adapters for popular data stores – a block store and a cloud store with a YCSB interface. Our implementation is preliminary in that they are built upon basic KV operations, i.e., `GET`, `MGET`, `SET`, `MSET` and `DELETE`, and we have not fully explored the use of other operations (e.g., range queries and iterations). However, our implementation is complete enough to show the feasibility of virtualizing various stores via the KV interface. Note that LightStore implements a KV store *natively*.

Block Adapter: We have designed a *block store* using a BUSE module [2], a user-space block I/O layer. The working mechanism of BUSE is similar to FUSE [50]. User applications send I/O requests by calling POSIX APIs (e.g., `read()`), which are then translated into `bio` requests by the kernel block I/O layer. The `bio` requests are forwarded to the block-store adapter in BUSE. The adapter creates a key for a KV request by using an LBA carried by `bio` as an input for a hash function. Currently, it simply concatenates LBA with a unique ID assigned to a client server. For example, LBA is `0x89` and the server ID is `0x1A0000`, a new key becomes `0x1A000089`. Then, the adapter gets a destination LightStore node using the consistent hashing algorithm with two input values, the server ID and the key. Depending on the type of a request, it builds a proper KV request (e.g., `GET`, `SET`), which is then submitted to the LightStore node. The data transfer unit is 8 KB in the block store. The block store can be used as a backend of various stores. One example is a *file store*; a file system is mounted on a block store and provides a file abstraction to applications. In Section 7, we will show how a LightStore file store operates, along with its performance.

YCSB Adapter: YCSB is a popular benchmark for evaluating cloud stores [13]. To emulate a *cloud store* over LightStore that exposes only the native KV interface, it is necessary to implement an adapter supporting YCSB commands, such as `READ`, `SCAN`, `INSERT`, `UPDATE` and `DELETE` [13]. Since range queries are not supported by LightStore yet, `SCAN` is not implemented. Each YCSB command directly corresponds to a specific KV operation (e.g., `INSERT` → `SET`). The only exception is that, for a single key, YCSB supports multiple fields, along with corresponding values (e.g., `INSERT Key : Field1=Value1 [Field2=Value2 ...]`). Multiple fields can be supported with `MSET` and `MGET` which are able to transfer multiple KV pairs at once. More specifically, the YCSB adapter is implemented as a form of a user-level library compiled with the YCSB client. Once an exported function is invoked by the client, say `INSERT 100 : foo=10 bar=20`, it creates subkeys by concatenating the main key (`100`) and fields (`foo` and `bar`) using a hash function. For instance, `subkey1 = hash(100) concat hash(foo)`. It then builds a new `MSET` request that contains (`subkey1`, `10`) and (`subkey2`, `20`). The `MSET` request is sent to a LightStore node using the consistent hashing module that is used for the block store.

Table 1. Datacenter cost comparison

	XtremIO X2 [20]	LightStore	LightStore with expansion
Throughput	144 Gbps	140 Gbps	140 Gbps
Network port	8 x 10GbE 4 x 16Gb FC[1]	14 x 10GbE	14 x 10GbE
Capacity	144 TB	28 TB	144 TB
Components	Xeon servers 72 x 2-TB SSDs	14 x 2-TB LSN[2]	14 x 2-TB LSN[2] 58 x 2-TB LEC[3]
Power	1700 W (storage) 60 W (switch)	280 W (storage) 60 W (switch)	860 W (storage) 60 W (switch)
Rack Space	5 U (storage) 0.25 U (switch)	0.4 U (storage) 0.25 U (switch)	2 U (storage) 0.25 U (switch)

[1] FibreChannel [2] LightStore Node [3] LightStore Expansion Card

6 Expected Operating Cost

To estimate the expected operating cost benefits of LightStore in a datacenter, we need to make a few assumptions regarding a *product* LightStore node. A properly designed SSD consumes less than 10 W [37, 53]. As LightStore aims to use the existing resources in SSDs, we assume the product LightStore node is an SSD-sized drive with 2 TB of flash and consumes up to 20 W, conservatively, considering the addition of a 10G NIC (e.g., 10 W [27]) and removal of host controllers (e.g., SATA, PCIe). We also assume the LightStore expansion card to be of the same capacity and form-factor as a LightStore node, consuming 10 W since it lacks a NIC.

We compare LightStore with an AFA system, Dell-EMC's XtremIO X2 [20], in terms of throughput, capacity, power and rack space requirement. XtremIO provides eight 10G Ethernet ports and four 16G FibreChannel ports, which aggregate up to 144 Gbps throughput. Since LightStore aims to deliver the internal flash throughput to the network, it is natural to build a LightStore cluster that has the same network throughput to compete against. The network is the eventual bottleneck in both systems as will be confirmed later. A LightStore cluster of the same throughput comprises 14 LightStore nodes, providing 140 Gbps data transfers. We

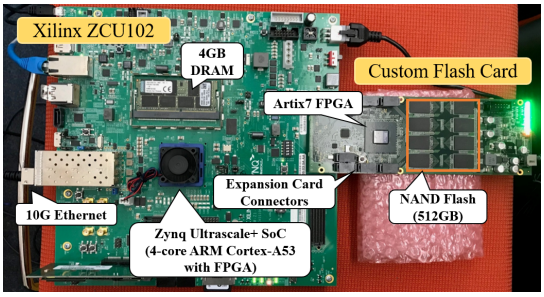

Figure 8. A LightStore node prototype with ZCU102

Table 2. KVS experimental setup

	x86	LightStore
CPU	Xeon E5-2640 (20x, 2.4 GHz)	ARM Cortex-A53 (4x, 1.2 GHz)
DRAM	32 GB	4 GB
SSD/Flash Throughput (R/W)[a] Latency (R/W)[a]	Samsung 960 PRO 512 GB 3.21 GB/s / 1.38 GB/s 80 µs / 120 µs	Custom 512 GB Flash 1.2 GB/s / 430 MB/s 120 µs / 480 µs
KVS	RocksDB v5.8 [21]	LSM-tree Engine
Client Ifc	ARDB [68]	KVS Protocol Server
Network Throughput[a]	10 Gbit Ethernet 1.20 GB/s	10 Gbit Ethernet 620 MB/s
OS	Ubuntu 16.04 (64-bit)	

[a] Throughput and latency were measured using real devices.

estimate the space requirement of a LightStore cluster from the fact that XtremIO requires 2 U for an SSD enclosure hosting up to 72 SSDs; 72 LightStore nodes require 2 U rack space. To match the capacity of the AFA system while keeping the throughput the same, we build another LightStore cluster using expansion cards. These cards increase the total capacity of the system but does not increase the network throughput of the system.

Table 1 summarizes the power/space requirement and storage capacity of the three storage systems. To make the cost model more realistic, we have added 60 W power and 0.25 U rack space to account for the network switch with 12-14 ports of 10 GbE or FibreChannel in each system [11]. The LightStore cluster with no expansion card has only 20 % of the capacity but 5.2x power-efficient and 8.1x space-efficient compared to XtremIO of the same throughput. The LightStore cluster with expansion cards is 2.0x power-efficient and 2.3x space-efficient while having the same capacity and throughput as XtremIO. If we include the cooling fee, the benefit of LightStore is even better as x86-based servers require more cooling.

7 Experimental Results

Comparing the performance of LightStore with conventional x86-based storage systems is complicated by the fact that LightStore required a custom flash card to implement our light-weight hardware FTL. Our flash card [40] has considerably lower throughput (up to 3.2x) and higher latency (up to 1.5x for reads and 4x for writes) than modern SSDs [53]. In addition, ARM cores are much slower than x86 cores, and network bandwidths of the two systems are also considerably different. Thus, we have devised a variety of experiments to understand if the performance is being limited by the raw flash performance, the network or the processor itself.

We first present the prototype implementation of LightStore and experimental setups, followed by an evaluation of LightStore as a native key-value store. We show the impact of the HW FTL and its computation requirement along with the evaluation. Finally, we present the performance of various LightStore-emulated data stores to show feasibility of using adapters for storage virtualization.

7.1 Prototype and Experimental Setup

We have built a prototype of the LightStore node using a Xilinx Zynq Ultrascale+ ZCU102 evaluation board [64], which has a SoC that integrates an ARM Cortex-A53 with four cores (1.2 GHz) and an FPGA, 4 GB DDR4 DRAM, and various external connectors (Figure 8). We ran the LightStore software on Ubuntu 16.04 with a Linux 4.9.0 kernel. Even though our board had a 10GbE port, its maximum throughput was only 620 MB/s with no workload on CPUs. This was due to the device driver for our development board which had been released only recently.

We implemented the lightweight HW FTL on a custom flash card [40]. The card has 512 GB NAND flash chips and can be connected to the ZCU102 board. Under the Hardware FTL, the card offered 1.2 GB/s and 430 MB/s sequential read and write performance, respectively, and 150K IOPS and 55K IOPS for random 8 KB page reads and writes, respectively. The flash capacity can be expanded by connecting extra cards using a simple serial-link protocol over SATA connectors.

The peak power of our prototype was measured to be about 25 W. When packaged as a product, unnecessary peripherals would be removed, and we expect the power to be reduced to less than 20 W as discussed in Section 6.

To create a LightStore cluster, we connected four homogeneous LightStore nodes to a 10GbE switch. Our client machines accessing LightStore had Intel's i5 7600K CPU with 4 cores running at 3.8 GHz, 8 GB DRAM and two 10GbE ports. They were connected to the same switch.

For comparing the performance of LightStore with the KVS running on x86, we used RocksDB [21] and its plugin module, ARDB [68], that exposes RocksDB KV operations to the network. Table 2 summarizes the experimental setup. From the specification [53], the 960 PRO SSD was rated to offer 3.5 GB/s and 2.1 GB/s for *sequential* reads and writes, respectively, and 330K and 140K IOPS for 4 KB *random* reads and writes, respectively. However, our measurement revealed that the sequential read and write bandwidths were 3.21 GB/s and 1.38 GB/s, respectively, on the filesystem. We tuned RocksDB according to the guidelines [22] to achieve the highest performance on an SSD. We also disabled an write-ahead log option because it was not used for LightStore.

Table 3. Workload description

KVS Workload	Description
Sequential Write (S-SET)	Set KV pairs using keys from 0 sequentially
Sequential Read (S-GET)	Get all KV pairs using all keys sequentially
Random Write (R-SET)	Set 50M KV pairs randomly (possible overwrites)
Random Read (R-GET)	Get 50M KV pairs randomly
Random R/W (R-Mixed)	Get or set (9:1) 50M KV pairs randomly

Table 3 lists five workloads for our experiments. All SET-related tests were performed on an empty database while GET-related ones were done on a prefilled one. To better understand performance profiles, we included both sequential and random I/O workloads, along with 9:1 mixed I/O. The results are represented in operations-per-sec (ops/s) and, if applicable, MB/s. The value size was 8 KB since LightStore was intended to be used for data stores, where large objects are common.

7.2 LightStore as a Key-value Store

7.2.1 Throughput

Figure 9 compares the performance of LightStore on ARM and RocksDB on x86. The light-shaded bars represent the local performance of a single LightStore node (LightStore-Local) and RocksDB on x86 with one SSD (x86-RocksDB). For the local setting, the KVS benchmark ran on the local KV stores without network connection, directly issuing KV requests to the LSM-tree engine (see Figure 5) or to RocksDB. The solid bars represent the throughputs of KVSs serving clients over the network. We evaluated a network-connected LightStore node (LightStore-Net) and RocksDB with the ARDB frontend on an x86 with an SSD (x86-ARDB). Each system was connected to the application server using an Ethernet switch.

In both settings, the performance of RocksDB is significantly superior than LightStore for the sequential tests (S-GET and S-SET) and for random reads (R-GET) but inferior for random writes (R-SET) and mixed loads (R-Mixed). We will analyze the two systems, both in the local and networked setting, to show that LightStore is likely to outperform RocksDB if similar flash and network were provided.

Local Performance: Both sequential writes and reads (S-SET and S-GET) saturated the raw flash bandwidth on LightStore-Local, offering 421 MB/s and 1.2 GB/s throughput, respectively. For sequential writes (S-SET) x86-RocksDB showed 1.27 GB/s which was much higher than the local performance of LightStore. Some of this performance can be attributed to 960 PRO SSD, whose flash chips have three times higher bandwidth than the ones in LightStore, but some of it has to do with a write buffer in the 960 PRO. Our experiments could not measure these two effects separately.

Another interesting fact to notice is that the sequential read (S-GET) throughput of 1.3 GB/s on RocksDB is much lower than the raw SSD read speed of 3.21 GB/s. This is due to that fact that RocksDB often fetched metadata from the SSD to find a location of a key in the LSM-tree. This traversal

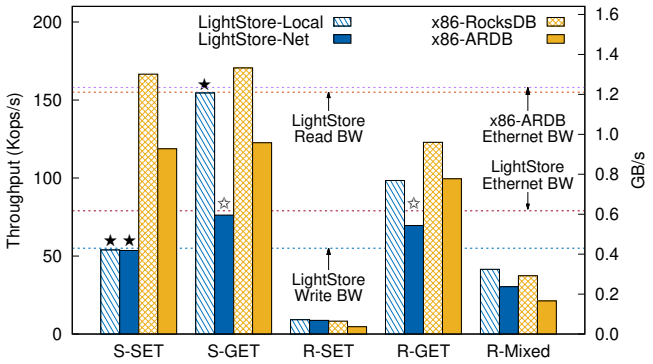

Figure 9. KVS performance: Star symbols indicate that LightStore (almost) saturates effective flash (★) or Ethernet (☆) bandwidth

on the tree caused a huge bandwidth loss. LightStore was able to cache the metadata (or keytables) entirely in DRAM, and thus, was not adversely affected by metadata reads.

There was a noticeable drop in the performance for random writes (R-SET) in both the systems but LightStore slightly outperformed RocksDB despite its slower flash. This was because LightStore, unlike RocksDB, only sorted metadata and avoided much of the repeated compaction overhead.

For random reads (R-GET), we expected that LightStore would show similar performance to sequential reads (S-GET) because all the keytables were cached in DRAM. However, we observed a nontrivial perform drop because of CPU overhead in performing the binary-search for a key in cached keytables. This problem can be alleviated with optimization like cache-line prefetching. The slower performance of RocksDB for random reads (R-GET) can again be attributed to frequent metadata accesses. Finally, for R-Mixed which is the mixture of random reads and writes, LightStore exhibited better throughput than RocksDB because of LightStore's superior random write performance with metadata-only compaction.

We estimate that given three-times faster flash, i.e., the same SSD as the x86-based setup, LightStore will easily match or outperform RocksDB.

Networked Performance: Figure 9 shows that the attaching to the network reduced the throughput in both systems. In LightStore-Net, S-GET was limited to 620 MB/s, the maximum prototytpe Ethernet bandwidth. The throughput of R-GET was slightly slower than the maximum due to the overheads from keytable searching.

Similar to the results with the local KV tests, x86-ARDB performed better than LightStore-Net on S-GET, S-SET and R-GET, thanks to its higher network and SSD throughputs. For R-SET and R-Mixed, LightStore-Net exhibited better performance than x86-ARDB and the difference was even bigger with the network. This was because the combination of RocksDB and ARDB aggravated the compaction overhead. Similar phenomenon was found in the sequential workloads (S-SET and S-GET). After ARDB was added, their performance dropped much below the network throughput.

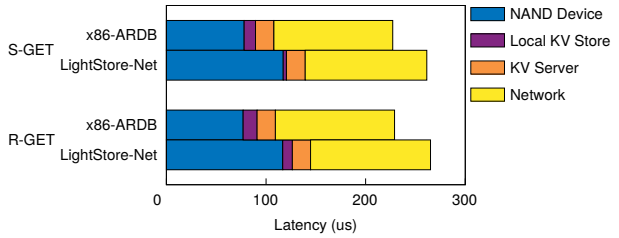

Figure 10. Latency comparison

This supports our claim that the deep software stack negatively affected the overall performance. Unlike x86-ARDB, LightStore-Net exhibited steady performance in spite of the addition of the KV protocol server unless the network was not a bottleneck.

7.2.2 Latency

We measured the average latency of LightStore-Net and x86-ARDB to see if embedded cores acted as a bottleneck in delivering short latency of flash to a client. End-to-end latency from a client to a KV store was measured by sending 10 million, 8 KB-sized, sequential/random read requests, one at a time.

Figure 10 shows our experimental results, where we have broken down the latency into four parts: 'NAND Device' is the time between the submission of an I/O request and the receipt of acknowledgment from the NAND device; 'KV Server' is the time spent by the key-value servers (ARDB or LightStore's KV Protocol Server); 'Local KV store' is the time spent by the local KV engines (RocksDB or LightStore's LSM-tree engine); 'Network' is the time taken by the network.

Even though both the KV server and local KV store of LightStore-Net ran on embedded cores, they exhibited the same or shorter latency compared to those in x86-ARDB. This was due to their lightweight designs. As expected, the main difference was the longer NAND latency of the LightStore's flash card. Under random reads (`R-GET`), the time spent by the local KV stores increased noticeably. RocksDB and the LSM-tree engine showed 1.3x and 3x longer latency than those in `S-GET`, respectively. This was due to the fact that RocksDB required extra flash reads to fetch metadata, and the LSM-tree engine required more CPU cycles for a key search. Those overheads were not significant with single-thread tests, but became huge when many I/O requests were processed concurrently, as we already have seen in Figure 9.

7.2.3 Scalability and Power-aware Performance

We evaluated the multi-node performance by clustering LightStore nodes. We measured the aggregated throughput of 1M `R-GET` operations while increasing the number of LightStore nodes by up to four. Similarly, we measured the throughput of `R-GET` on x86-ARDB while attaching up to four SSDs to the RAID. To understand the impact of better network speed, we also increased the number of network

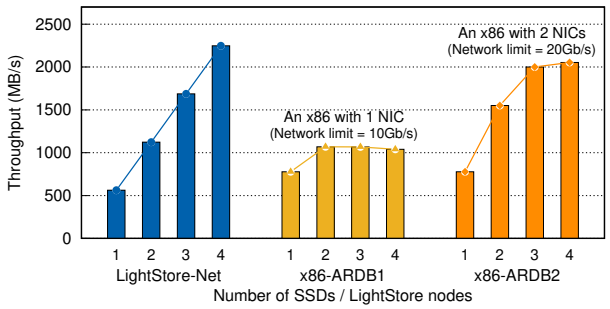

Figure 11. Scalability evaluations with varying numbers of SSDs and NICs in x86-ARDB and LightStore-Net

ports. Since our server had two 10GbE ports, two different settings, x86-ARDB1 with one network port and x86-ARDB2 with two ports, were used.

Figure 11 summarizes the scalability of each setting. The throughput of LightStore improved almost linearly with respect to the number of nodes in a LightStore cluster as expected. This suggests LightStore scaled well by just adding more nodes to the network. On the other hand, the throughput of x86-based KVS was mostly capped by the network interface. For example, the network was a bottleneck on x86-ARDB1 with 2 SSDs and on x86-ARDB2 with 3 SSDs.

Some might argue that it is possible to scale up the performance of x86-ARDB by installing more servers. This is definitely true, but it comes at the expense of purchasing more servers, renting more floor space and consuming more power. Some might want to argue that, by adding more NICs to the server, we can address the scalability problem. This is also possible, but the number of NICs that can be installed on the same server is also limited by the CPU performance, the system bus bandwidth and the available NIC ports. Note that commercial AFA systems usually employ up to 8-12 network ports per server hosting more than 70 drives (see Table 1) [20, 46].

The main benefit of LightStore comes from reduced power consumption because of a removal of an x86 server. We use power-aware performance metric – operation-per-joule (ops/J) to quantify this advantage. As an illustration, we interpret the performance numbers in Figure 9 in terms of ops/J. We compare the power and performance of x86 with 4 SSDs with a four-node LightStore cluster. While running benchmarks, the peak power of a LightStore node and x86-ARDB with 4 SSDs was measured as 25 W and 400 W, respectively. Thus, a four-node LightStore cluster will consume 100 W

Table 4. Power-aware performance (ops/J)

	`S-SET`	`S-GET`	`R-SET`	`R-GET`	`Mixed`
x86-ARDB	296.9	306.4	11.8	248.9	53.0
x86+4SSDs[a]	1187.6	1225.8	47.19	995.45	212
LightStore-Net	2143.2	3049.6	350	2782	1212.8
Gain	1.8x	2.5x	7.4x	2.8x	5.7x

[a] Performance numbers were estimated assuming a perfect scalability (4x) from x86-ARDB with single SSD.

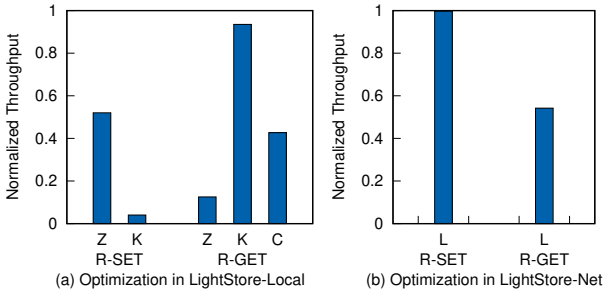

Figure 12. Effects of optimization techniques

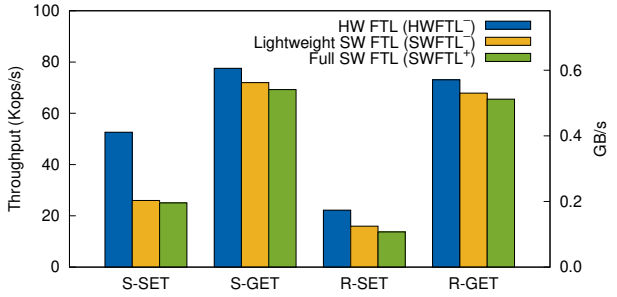

Figure 13. LightStore performance with HW and SW FTLs

and show 4x performance. To estimate the performance of x86-ARDB with 4 SSDs, we assume *perfect scalability* and multiply each performance number for x86-ARDB in Figure 9 by four. Doing this arithmetic yields the numbers shown in Table 4. The LightStore cluster achieves 1.8x-2.5x better sequential and 2.8x-7.4x better random ops/J, respectively. As mentioned in Sections 7.1, the power is measured rather conservatively and *product* LightStore node would consume sub-20 W, making it more attractive.

7.2.4 Effect of Software Optimizations

We have employed four software optimization techniques in LightStore: 1) zero-copy memory (Z); 2) keytable-only compaction (K); 3) keytable caching (C); and 4) lock-free queues (L). To understand their effects, we measured the performance of R-SET and R-GET by *turning off* each optimization one by one as shown in Figure 12. All the results are normalized to the case where all the optimizations are turned on (the absolute numbers were noted in Figure 9).

As shown in Figure 12(a), both R-SET and R-GET suffered from extra memory copies – their throughputs degraded by 2x and 8x. Particularly, R-GET suffered more than R-SET. Compared to a page read, a page write takes much longer, which hides long memory copy latency. Moreover, while R-SET requests could be buffered in the memtable and queues, R-GET should be served immediately.

The keytable-only compaction was effective for R-SET. Compared to physical value sorting for compaction, keytable-only compaction showed a speedup of 23.5x.

The impact of the keytable caching was significant for R-GET in Figure 12(a). In our tests, the configuration C employed a bloom filter by default. It was effective to improve read latency, but since it required reading, on an average, two keytables to retrieve a desired value [15], the degradation of read latency was unavoidable. With the keytable caching that fully loaded keytables in DRAM, we were able to remove all the keytable reads, achieving 2x better throughput.

The effectiveness of lock-free queues was most evident when LightStore was attached to the network, because lock-free queues were used only to glue the KVS protocol server to the LightStore engine. As depicted in Figure 12(b), lock-free queues improved the performance of R-GET by 1.9x. On the other hand, there were no benefits on R-SET since high compaction overheads neutralized the effect.

7.2.5 Effect of HW FTL

To understand the impact of the HW FTL on performance, we evaluated the performance penalty of running software FTLs in a LightStore prototype node. To measure the penalty, we built two LightStore variants with 1) a full-fledged SW page-level FTL [25] that packs raw flash chips as a block device (SWFTL$^+$) and 2) a lightweight SW FTL that implements the same AFTL algorithm [39] used in the LightStore HW FTL (SWFTL$^-$). In both variants, the SW FTLs ran along with the LightStore software stack on the same LightStore prototype board. For FTL tasks, two threads were added for address remapping and garbage collection, respectively. Also, the software variants kept their mapping tables in the system DRAM along with all the data in LightStore. For 512 GB flash, the table sizes for SWFTL$^+$ and SWFTL$^-$ were about 512 MB and 0.5 MB, respectively.

We used a smaller dataset (15 GB) to perform sequential/random reads and writes on the original LightStore using the lightweight HW FTL (HWFTL$^-$) and two variants, SWFTL$^+$ and SWFTL$^-$. Figure 13 summarizes the performance of the three systems. For read tests, SWFTL$^-$ and SWFTL$^+$ showed 7 % and 10 % performance penalty compared to HWFTL$^-$, respectively, regardless of whether the reads were sequential or random. For write tests, SWFTL$^-$ and SWFTL$^+$ suffered from 50 % and 52 % performance loss for S-SET and 28 % and 38 % slowdown for R-SET, respectively. In all cases, SWFTL$^+$ experienced more penalty than SWFTL$^-$, which is expected due to the complexity difference. Since the lightweight FTL used a simple block-mapping algorithm, the number of mappings to perform during KV writes was much less compared to the full FTL. Moreover, SWFTL$^-$, just like HWFTL, does not suffer from copying valid pages during garbage collection, which is huge for SWFTL$^+$ [39].

As discussed in Section 7.2.1, KV read operations in LightStore are network-bound rather than compute-bound. Moreover, the SW FTL performs only a few table lookups for flash read requests, and the writer/compaction thread (Thread #4 in Figure 5) is disabled all the time. These explain the small penalties seen in the read tests when the SW FTLs were used.

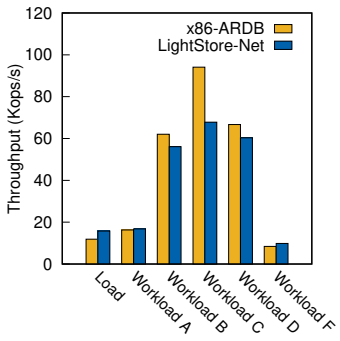

Figure 14. YCSB performance

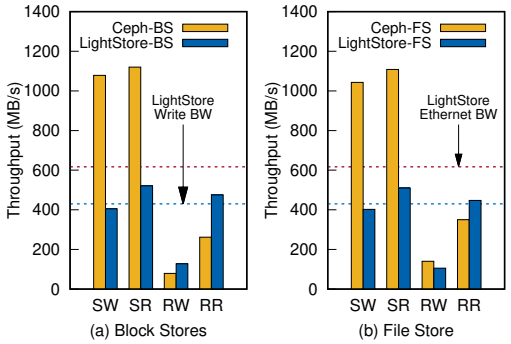

Figure 15. Block and file stores performance

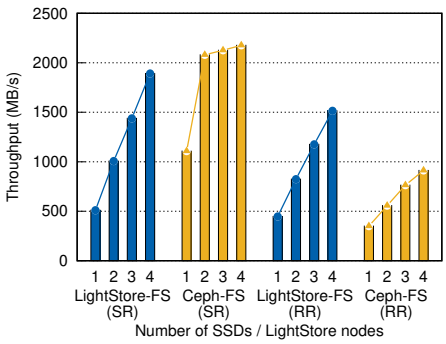

Figure 16. File store scalability

On the other hand, the penalties were much bigger for write tests. Handling KV write operations require more computing power due to writer/compaction threads under heavy workload. Flash page writes also trigger the FTL to perform more tasks, i.e., address remapping and garbage collection. Both compaction and FTL tasks made the penalty of running SW FTLs much more noticeable on write intensive tests. Since the compaction overhead was much greater under the random workload (R-SET), the relative effect of running SW FTLs was bigger with the sequential workload (S-SET).

Despite its superb performance, the adoption of the HW FTL may increase engineering efforts in designing and implementing flash controllers. Conversely, an FPGA-less LightStore prototype implementing the software FTLs would allocate a non-trivial portion of its processing power for the FTL under heavy writes. To overcome the 50 % penalty seen in SWFTL$^-$, the FPGA-less prototype needs another set of processor and DRAM dedicated to run the FTL and communicate with the other processor running LightStore software stack. These extra resources evidently require more chip area and power and may have worse latency compared to the LightStore HW FTL. The above observation implies that, when deploying LightStore nodes, there exists a design trade-off between cost and design changes. The HW FTL requires us to modify an existing hardware controller but achieves high performance at a lower cost. SW FTL enables us to keep using an existing controller design. However, more CPU/DRAM resources should be added to offer similar performance as HW FTL, which results in the increase of the per-node hardware cost as well as the maintenance cost.

7.3 Evaluation under Datacenter Applications

7.3.1 Cloud Application

To give better insight into the performance of LightStore when used as a KVS in real-world cloud services, we ran YCSB [13] on LightStore-Net and x86-ARDB. For the YCSB client to access LightStore, the YCSB adapter was run on the client side. YCSB consists of six workloads (A-F), which are the combination of reads, writes and read-modify-writes, except for Workload E (range-query) [12]. Since the range-query is not yet available in LightStore, we ran YCSB with Workloads A-D and F along with Load which prefills a database for workloads.

Figure 14 summarizes the results. LightStore-Net performed slightly better with the write-intensive workloads (Load, Workloads A/F) as compaction was less expensive than x86-ARDB. On the other hand, since the SSD used in x86-ARDB had better read throughput (3.21 GB/s) compared to our custom flash card (1.2 GB/s), x86-ARDB started to perform better than LightStore-Net as workloads had more reads (Workloads B-D). The performance difference was maximum for Workload C which has 100% reads. LightStore with a faster flash card would have certainly performed better.

7.3.2 Block and File Applications

LightStore's block adapter enables us to emulate a network-attached block device (LightStore-BS) over LightStore nodes. Thanks to the generality of a block I/O interface, a file system can be mounted on a block store, which creates another type of store, a file store (LightStore-FS). We compared them with a block store (Ceph-BS) and a file store (Ceph-FS) provided by Ceph, a popular distributed storage system [63]. We ran Ceph on the same x86 machine and SSD used for the previous experiments (Table 2). We first evaluate a single-node/SSD performance and then discuss scalability with multiple nodes/SSDs.

LightStore-BS: To measure the performance of block stores, we used a popular I/O benchmark tool, fio [5], on LightStore-BS and Ceph-BS. We configured fio to issue four different workloads of 5M block I/Os: sequential write (SW), sequential read (SR), random write (RW) and random read (RR). For sequential I/Os, a request size was set to 1 MB, and for random I/Os, it was 8 KB. Figure 15(a) shows the results.

Owing to limited flash/network bandwidths, LightStore-BS showed slower performance than Ceph-BS for sequential I/Os. But, LightStore-BS fully saturated the write throughput of the flash for SW. Even if it was not significant, there was a small gap between the maximum network throughput and the achieved one in SR. It was caused by BUSE that required

frequent kernel-user mode changes and context switch having a high impact on reads. This problem should disappear once the block adapter is implemented in the kernel. For random I/Os, LightStore-BS outperformed Ceph-BS on both reads and writes. Ceph was optimized for large objects (≥ 1 MB), and thus it experienced a huge overhead for highly random and small I/O workloads. A similar observation was also reported in [38].

LightStore-FS: To understand the performance of file stores, we mounted the ext4 file system on LightStore-BS. We ran `fio` to generate large file (40 GB) writes and reads (SW and SR) and random file (8 KB, 5M requests) writes and reads (RW and RR). Figure 15(b) summarizes the performance of file stores. Although the results were mostly similar to block stores, Ceph-FS performed slightly better than LightStore-FS on random writes. This was because, while Ceph-FS directly talked with Ceph storage nodes, LightStore-FS ran atop LightStore-BS as an additional layer. We plan to implement a file-store adapter converting file operations to KV queries. This will provide better performance by enabling file-system applications to access LightStore without the intermediate layer.

Scalability: We have installed up to four LightStore nodes or SSDs to evaluate the scalability of LightStore and Ceph. Due to the limited number of PCIe ports, two 10GbE NICs were installed on x86. Figure 16 summarizes the read performance (SR and RR) of LightStore-FS and Ceph-FS. LightStore performance scaled as the number of the nodes increased. In Ceph-FS, the network became bottleneck with 2 SSDs under sequential reads. Again, this confirms the bandwidth mismatch problem in x86-based systems. Under random reads, LightStore-FS performed better than Ceph-FS in all cases due to slow random I/O performance of Ceph.

8 Conclusion

We have presented LightStore, an ARM-based key-value store, as a replacement for x86-based storage servers. Our work was motivated by the observations that the existing storage architecture did not scale well, and applications failed to exploit full performance of high-speed SSDs over the network. By rethinking the design of the storage node and by restricting it to behave as a KV store, we designed a lean drive-sized high-speed LightStore node which plugs directly into a network port. We estimate LightStore to be 2.0x power-efficient and 2.3x space-efficient compared to x86-based AFA systems of the same capacity. Our results showed that a LightStore cluster scaled linearly as more nodes were added to the network. A four-node cluster showed a comparable throughput to the AFA with four SSDs and achieved up to 7.4x better ops/J.

We plan to extend our LightStore in several directions. First, LightStore does not support range queries and small-sized requests. Since the LSM-tree is used as a storage engine, these features can be added easily. Second, we only included a few simple cases for adaptors, i.e., YCSB and block adapters. Our next plan is to support more advanced protocols like file requests and SQL queries. Finally, for proper integration of LightStore in a datacenter, technical issues (e.g., data consistency and failure tolerance) must be addressed. We plan to begin this work by looking at prior studies like FAWN.

Acknowledgments

This work was partially supported by the Samsung Semiconductor GRO grant and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2018R1A5A1060031, NRF-2017R1E1A1A01077410). We thank Vijay Balakrishnan, a former Samsung employee, for his insights and comments during our discussions. We also thank the MIT CSAIL computation structures group members for fruitful discussions and numerous helps on the HW-SW interfaces and flash cards. Bluespec and Xilinx supported licenses for their tools. Chanwoo Chung is a recipient of the Samsung scholarship.
(Corresponding author: Sungjin Lee)

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 57–70.

[2] Zachary Amsden. 2009. Allow Userspace Block Device Implementation. LWN.net. https://lwn.net/Articles/343514/.

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 1–14.

[4] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2015. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books.

[5] Jens Axboe. 2018. Flexible I/O Tester. https://github.com/axboe/fio.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1–14.

[7] Michaela Blott, Ling Liu, Kimon Karras, and Kees Vissers. 2015. Scaling Out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[8] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. 2009. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 217–228.

[9] Adrian M. Caulfield and Steven Swanson. 2013. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 464–474.

[10] Chanwoo Chung, Jinhyung Koo, Arvind, and Sungjin Lee. 2017. Lightweight KV-based Distributed Store for Datacenters. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'17)*. USENIX Association, Berkeley, CA, USA, 11–11.

[11] Cisco. 2018. Cisco 550X Series Stackable Managed Switches Data Sheet. https://www.cisco.com/c/en/us/products/collateral/switches/

550x-series-stackable-managed-switches/datasheet-c78-735874.html.

[12] Brian F. Cooper. 2010. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 143–154.

[14] Jonathan Corbet. 2017. Zero-copy Networking. LWN.net. https://lwn.net/Articles/726917/.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 79–94.

[16] Arnaldo Carvalho De Melo. 2010. The New Linux 'perf' Tools. In *Slides from Linux Kongress*, Vol. 18.

[17] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-value Store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425.

[19] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 25–36.

[20] Dell. 2018. Dell EMC XtremIO X2 Specifications. https://www.emc.com/collateral/specification-sheet/h16094-xtremio-x2-specification-sheet-ss.pdf.

[21] Facebook. 2018. RocksDB. http://rocksdb.org/.

[22] Facebook. 2018. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[23] OpenStack Foundation. 2018. Swift. https://wiki.openstack.org/wiki/swift.

[24] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. 1997. File Server Scaling with Network-attached Secure Disks. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 272–284.

[25] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 229–240.

[26] Hitachi. 2017. Hitachi Accelerated Flash 2.0. https://www.hitachivantara.com/en-us/pdf/white-paper/hitachi-white-paper-accelerated-flash-storage.pdf.

[27] Intel. 2018. 10GBASE-T for Broad 10 Gigabit Adoption in the Data Center. https://www.intel.com/content/dam/support/us/en/documents/network/sb/intel_ethernet_10gbaset.pdf.

[28] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 373–384.

[29] Sang-Woo Jun, Chanwoo Chung, et al. 2015. Large-scale high-dimensional nearest neighbor search using flash memory with in-store processing. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*. IEEE, 1–8.

[30] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 1–13.

[31] Sang-Woo Jun, Ming Liu, Shuotao Xu, et al. 2015. A transport-layer network for distributed fpga platforms. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*. IEEE, 1–4.

[32] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the ACM Symposium on Theory of Computing (SOTC)*. 654–663.

[33] Byungseok Kim, Jaeho Kim, and Sam H. Noh. 2017. Managing Array of SSDs When the Storage Device Is No Longer the Performance Bottleneck. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[34] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[35] Myron King, Jamey Hicks, and John Ankcorn. 2015. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 13–22.

[36] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 345–359.

[37] Andrey Kudryavtsev. 2016. Managing Power Consumption of Intel Data Center SSDs. https://itpeernetwork.intel.com/managing-power-consumption-of-intel-data-center-ssds/.

[38] Eunji Lee, Youil Han, Suli Yang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. How to Teach an Old File System Dog New Object Store Tricks. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[39] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-Managed Flash. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 339–353.

[40] Ming Liu, Sang-Woo Jun, Sungjin Lee, Jamey Hicks, and Arvind. 2016. minFlash: A Minimalistic Clustered Flash Array. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. 1255–1260.

[41] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 133–148.

[42] Microsoft. 2013. *Common Internet File System*. Microsoft TechNet Library.

[43] NetApp. 2014. NetApp FAS25xx System: Replacing an NVRAM or NVMEM Battery. https://library.netapp.com/ecm/ecm_download_file/ECMP1188031.

[44] NetApp. 2017. NetApp FAS9000 Modular Hybrid Flash System. https://www.avnet.com/wps/wcm/connect/onesite/a8e61d63-b112-4187-bc2d-7bee49541e49/netapp-FAS9000-datasheet.pdf?MOD=AJPERES&CVID=m1mboQX&CVID=m1mboQX.

[45] NetApp. 2018. NetApp DS4486 Specification. https://www.netapp.com/us/products/storage-systems/disk-shelves-and-storage-media/index.aspx.

[46] NetApp. 2018. NetApp SolideFire All-Flash Array. https://www.netapp.com/us/products/storage-systems/all-flash-array/solidfire-web-scale.aspx.

[47] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[48] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A High Performance File System for Non-volatile Main Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Article 12, 12:1–12:16 pages.

[49] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating*

Systems (ASPLOS). 471–484.

[50] Nikolaus Rath. 2018. FUSE (Filesystem in User Space). https://github.com/libfuse/libfuse.

[51] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[52] Samsung. 2016. Samsung Intoduces World's Largest Capacity (15.36TB) SSD for Enterprise Storage Systems (PM1633a). https://goo.gl/65evye.

[53] Samsung. 2016. Samsung NVMe SSD 960 PRO. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/.

[54] Samsung. 2017. Introducing the Samsung PM1725a NVMe SSD. http://www.samsung.com/semiconductor/insights/tech-leadership/brochure-samsung-pm1725a-nvme-ssd/.

[55] Samsung. 2017. Samsung Key Value SSD Enables High Performance Scaling. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Key_Value_Technology_Brief_v7.pdf.

[56] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. 1985. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Conference (USENIX)*. 119–130.

[57] Salvatore Sanfilippo. 2018. Redis. https://redis.io/documentation.

[58] Seagate. 2015. Kinetic HDD. https://www.seagate.com/support/enterprise-servers-storage/nearline-storage/kinetic-hdd/.

[59] Seagate. 2016. Nytro XP7200 Add-In Card. https://www.seagate.com/www-content/product-content/ssd-fam/nvme-ssd/nytro-xp7200/_shared/docs/nytro-xp7200-add-in-card-ds1905-1-1607us.pdf.

[60] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 217–228.

[61] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 91–104.

[62] Twitter. 2017. Fatcache. https://github.com/twitter/fatcache.

[63] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. 307–320.

[64] Xilinx. 2018. Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[65] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 323–338.

[66] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proceedings of the VLDB Endowment* 10, 4 (2016), 301–312.

[67] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

[68] yinqiwen. 2018. ARDB: A Redis Protocol Compatible NoSQL. https://github.com/yinqiwen/ardb.

[69] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 3–18.