ACCELERATING STORAGE APPLICATIONS WITH EMERGING KEY VALUE STORAGE DEVICES

A Dissertation

by

MIAN QIN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	A. L. Narasimha Annapareddy
Co-Chair of Committee,	Paul V. Gratz
Committee Members,	Daniel A. Jimenez
	Stavros Kalafatis
	Krishna R. Narayanan
Head of Department,	Miroslav M. Begovic

August 2021

Major Subject: Computer Engineering

ABSTRACT

With the continuous explosion of unstructured data in the big data era, traditional software and hardware stack are facing unprecedented challenges on how to operate on such data scale. Thus, designing new architectures and efficient systems for data oriented applications has become increasingly critical. Key-value stores (KVS) are important storage infrastructure to handle the fast growing unstructured data and have been widely deployed in a variety of scale-out enterprise applications. How to efficiently manage data redundancy for key-value stores to provide data reliability, support range query to accelerate analytic oriented applications under emerging key-value store system architecture become important research problems.

In this work, we focus on how to design new software hardware architectures for the key-value store applications. In order to address the different issues identified in this dissertation, we propose to employ a logical key management layer for the emerging key-value devices, a thin layer that maps logical keys into physical keys on the devices. We show how such a layer can enable multiple solutions to improve the performance and reliability of key-value device based storage systems. First, we present KVRAID, a high performance, write efficient erasure coding management scheme on emerging key-value SSDs. The core innovation of KVRAID is to propose a logical key management layer that maps logical keys to physical keys to efficiently pack similar size KV objects and dynamically manage the membership of erasure coding groups. KVRAID outperforms software block RAID by 18x on throughput and reduces 15x write amplification while retain less CPU utilization. Second, we present KVRangeDB, an key-value store that supports range queries on a hash-based KVSSD which leverage an ordered log structure tree based key index. In addition, we propose to selective pack smaller and cold application records into a larger physical record on the device through the logical key translation layer. Our experiments shows that KVRangeDB greatly improves performance for metadata service in distributed filesystems. Third, we propose a generic FPGA accelerator for emerging Minimum Storage Regenerating (MSR) codes which maximizes the computation parallelism and minimizes the data movement between off-chip DRAM and the

on-chip SRAM buffers. Our proposed accelerator achieves ∼3x better throughput performance and ∼5x better power efficiency compared to state-of-art multi-core CPU implementation and modern GPU accelerators.

DEDICATION

To my mom and dad, who make this amazing journey possible.

To my beloved wife, your inspiration and encouragement are priceless.

ACKNOWLEDGMENTS

First and foremost I am deeply grateful to my advisors, Dr. A. L. Narasimha Reddy and Dr. Paul V. Gratz for their continuous advice, feedback and enormous support during my Ph.D. study. They guided me into the fascinating world of computer architecture and system research, encouraged me to explore every corner of the research possibilities, and lifted me to pursuit every opportunity to collaborate with brilliant minds in academia and industry. Their immense knowledge and abundant professional experience have been the sturdy backbone of my daily academic research. I would also like to express my gratitude to my advisory committee members: Daniel A. Jiménez, Krishna Narayanan, and Stavros Kalafatis for their insightful feedback and comments on my research, and this final dissertation.

I also thank to my colleague Dr. Fei Wen who I closely collaborate with during my research. I appreciate all the insightful discussion and fun debugging experience. Special thanks to Rekha Pitchumani, Joo Hwan Lee and Yang seok Ki at Memory Solutions Lab in Samsung Semiconductor Inc. The precious collaboration experience with the Samsung researchers helps me gain many industry perspectives in the field. I would also like to thank my advisor Bradley Settlemyer and colleagues Qing Zheng, Jason Lee during my internship in Los Alamos National Laboratory. The research collaboration in LANL and afterwards allows me grasping a wide knowledge of storage research and the cutting edge industry trends. Thanks to all the colleagues in Computer Architecture, Memory Systems and Interconnection Networks (CAMSIN) groups.

Finally, I want to thank my parents and my uncle and other family members, for their selfless, unconditional support during my Ph.D. program. It's my greatest fortune to have you to come through all struggles and difficulties in my life. Thanks to my beloved wife Ya Gao, your support and encouragement are priceless.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor A. L. Narasimha Reddy, Professor Paul V. Gratz, Professor Stavros Kalafatis and Professor Krishna R. Narayanan of the Department of Electrical and Computer Engineering and Professor Daniel A. Jiménez of the Department of Computer Science and Engineering. Yang seok Ki and Rekha Pitchumani of Samsung Semiconductor Inc collaborated in work reported in Chapter II. Bradley Settlemyer, Jason Lee of Los Alamos National Laboratory and Qing Zheng from Carnegie Mellon University collaborated in work reported in Chapter III. Joo Hwan Lee, Rekha Pitchumani and Yang seok Ki of Samsung Semiconductor Inc collaborated in work reported in Chapter IV. All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by National Science Foundation (NSF) through grants I/UCRC-1439722 and FoMR-1823403, and a generous support by Samsung Semiconductor Inc, Los Alamos National Laboratory.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND LITERATURE REVIEW

As we entered the era of big data, computer systems are experiencing a prodigious scale of data volume. IDC projects that data created and replicated by the world will grow to 163 ZB by 2025 [5], and unstructured data such as photos, videos, and audios will dominate the overall data volume [5]. The growth of data volume and the rise of unstructured data brings new challenges to the conventional computer systems.

1.1 Key value stores for unstructured data

Key-value stores (KVS) such as Dynamo [6], BigTable [7], HBase [8], Cassandra [9] are important building blocks for handling the increasingly growing unstructured data. Compared to traditional datastores such as file systems [10, 11, 12] and relational databases [13, 14, 15, 16] which were designed for structured data, key-value stores provide simple hash table like interface which store, retrieve, delete a piece of data using a unique key. The simplicity of key-value interface also empowers better scalability, availability [7, 8, 6, 9] which are critical to today's big data environment.

Today's state-of-the-art software key-value store engines [8, 17, 18] leverage Log Structured Merge-tree (LSM-tree) structure [19] to optimize performance on block devices such as Hard Drive Disks (HDDs) [20, 21, 22] and Solid State Drives (SSDs) [23, 24]. LSM-trees organize small objects into multiple levels of large, sorted tables (SSTable) [17, 18]. All writes will perform out-of-place update-and-writes to the top-level table. Reads will search from the smaller top-level table to the larger bottom level table to find the most updated data. A major advantage of the LSM-tree design is small writes are converted to large sequential I/O which is optimal for HDDs/SSDs performance. However, it requires a background compaction process, which reads the written data from bottom levels back and merges them to reorganize into a new level in multiple iterations as data scale grows. Such compaction process comes at the cost of high CPU utilization and I/O amplification [25, 26, 27].

Both academia and industry seek to overcome the high write amplification and CPU utilization problems for software key-value stores. Wisckey [25] proposes keys and values separation which stores values in a log file out from the LSM tree and only store key and value pointer pairs in a small LSM tree to reduce the write amplification. NVMKV [27] presents a hashing-based key-value store design to reduce write amplification. PebblesDB [26] proposes an enhanced LSM tree structure named Fragmented LSM trees to reduce the compaction complexity. Samsung KVSSD [28] and Pink [29] proposed key-value interfaced hardware that exposes direct key-value interface to reduce the host CPU utilization by offloading the key-value engine to the hardware.

1.2 Emerging key-value devices

Due to the physical characteristic of the storage medium, conventional storage device such as HDDs [20, 21, 22] and SSDs [23, 24] are block based, which means the hosts communicate with the storage device through block address and operate on fixed size, aligned blocks. On top of that, a special software layer referred to as file system is responsible to manage the storage devices and provide interfaces for storage applications. Thus, storage applications such as key-value stores needs to go through multiple software layers and in-directions to the underneath storage devices, adding extra latency and I/O overhead. Figure 1.1 demonstrates the traditional *fat* software stack between key-value stores applications and the storage devices. A key value record needs to go through multiple translations in the entire system stack. First, the key of the record needs to map to file offsets. Second, file offsets will translate to logical block addresses (LBAs) through file system. Finally, logical block addresses (LBAs) needs to map to physical block addresses (PBAs) inside the device [17, 18, 30]. Software layers also require extra indies for each translation, leading to storage and performance overhead.

However, with the emerging key-value devices such as KVSSD [28], the system architecture becomes more streamlined with a *thin* layer of software stack as support library and device that offloads key to block management. Instead of suffering multiple software in-directions from keys to logical block addresses, applications can directly issue put/get requests to the device through a lightweight library and device driver that come with the key-value devices. The new architecture

helps simplify software overhead by reducing host CPU utilization and memory overhead which greatly benefits today's cloud environments [31].

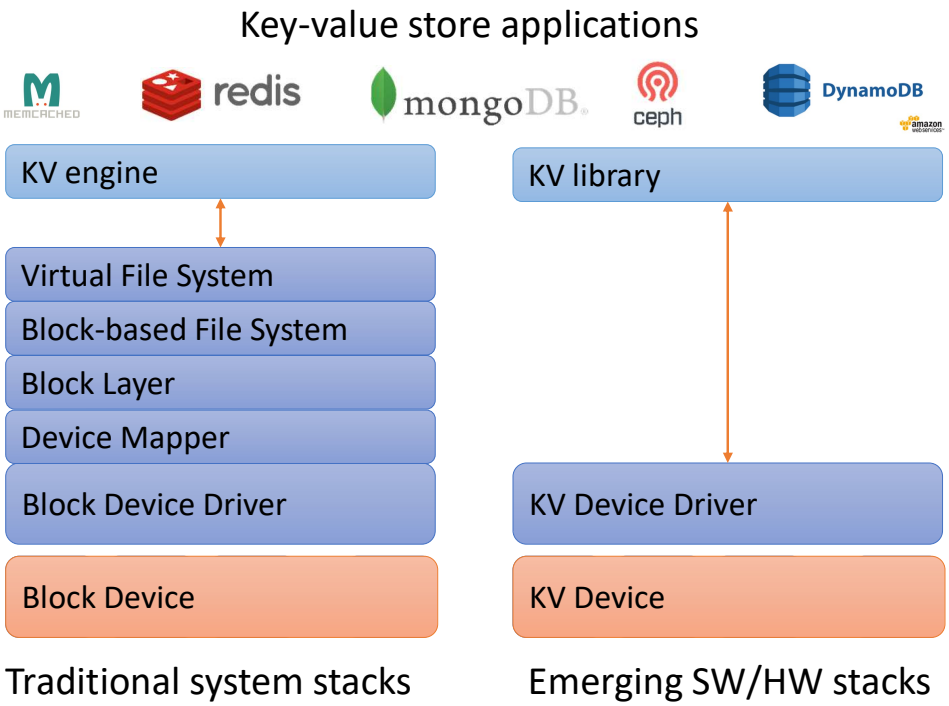


Figure 1.1: Comparison between traditional system stack and new software hardware stack for the emerging key-value devices.

The key-value storage devices present unique characteristics in case of performance. In Figure 1.2, we show the results of experiments we conducted on Samsung KVSSD [28] to evaluate read and write I/O performance characteristics for different value sizes from 128B to 2MB (maximum size supported by the device).

There are two interesting performance characteristics of KVSSD as shown in Figure 1.2.

- The IOPS performance for both put and get remain nearly the same for value sizes (less than 16KB).
- The update (overwrite the existing keys) IOPS performance is noticeably lower ($\sim$25%) than put operations.

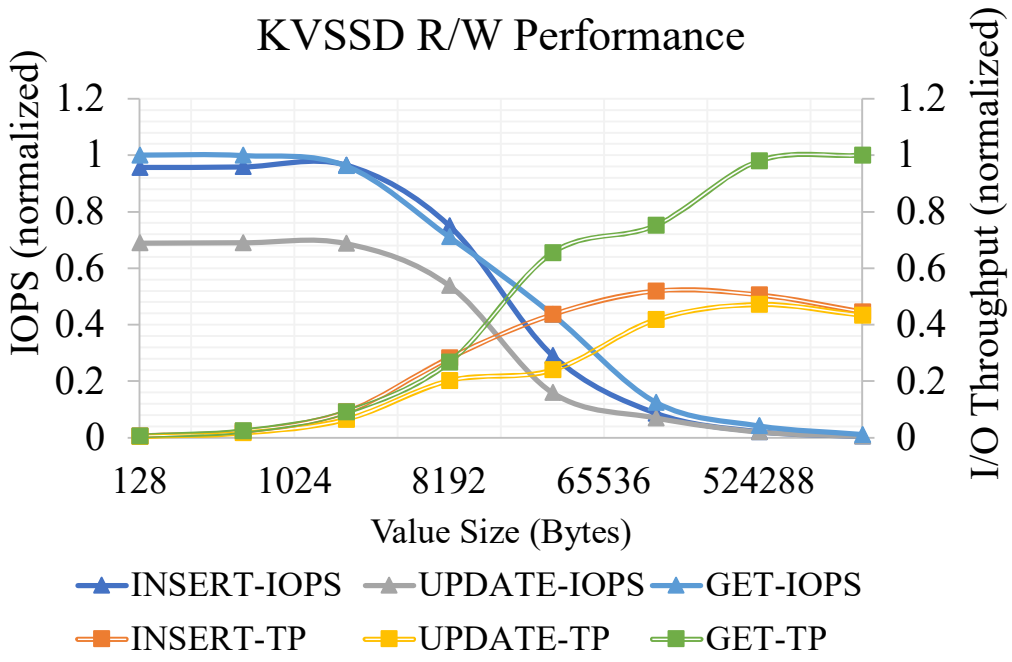


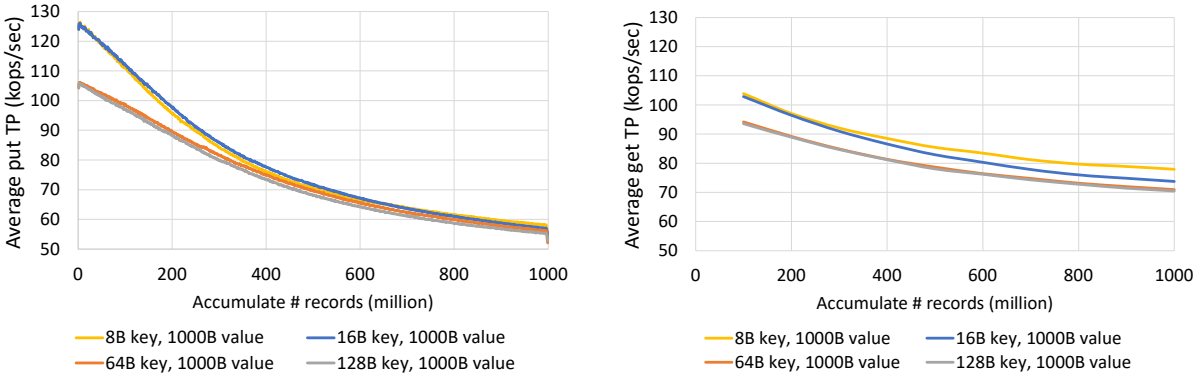
Figure 1.2: KVSSD performance characteristics under different value size "Reprinted from [1]".

The performance for put and get for smaller records (under 16KB) implies that by packing multiple KV objects in a single I/O, we can increase the overall device bandwidth while not sacrificing latency performance.

Figure 1.3 indicates that both put and get performance of the KVSSD drops significantly with larger number of records. Besides, the size of the key also impact the performance, i.e. large key size deteriorates performance. We also conduct experiments on various value size and observe that the performance is dependent on the number of records stored on the device, irrespective of the size of the records. Such observations demonstrate asymmetric performance characteristics of key-value storage devices, which are fundamentally different from conventional block devices. In the following section, we propose logical key remapping for key-value devices to tackle the asymmetric performance characteristic and other issues introduced by the key-value devices.

1.3 Logical key remapping for key-value devices

Unlike conventional block-based storage device such as Hard Drive Disks (HDDs) [20, 21, 22] and Solid State Drives (SSDs) [23, 24] which operate on fixed size blocks and index through



(a) Put throughput

(b) Get throughput

Figure 1.3: Performance scale down with increase of number of records stored in KVSSD.

continuous block address, the emerging key-value storage devices such as KVSSDs index through arbitrary size key for arbitrary size value. The fundamental difference in indexing between block-based storage device and key-value storage devices, i.e. finite block address space versus infinite key space raises new problems for existing storage services and applications.

In this dissertation, propose a logical key management layer that translates logical application keys to physical device keys. The design of physical keys opens up the space for flexible addressing to the key-value storage devices. The logical key remapping also enables packing multiple key-value records into a single physical records to optimize device performance.

In the following sections, we will introduce some unique problems of the key-value storage devices and how we leverage the logical key remmaping technique to gracefully address those problems.

1.4 Dissertation Statement

In this dissertation, we focus on providing better storage solutions for the rise of unstructured data in the "big data" era. This dissertation focuses on improving key value stores based on KV devices. We propose a logical key management layer that enhances and enables many new functions on Key value interface devices.

1.5 Dissertation Organization

In the following chapters, we first introduce the motivation of the research and then elaborate on the detailed design and implementation followed by thorough experiments results and analysis. In Chapter II, we present KVRAID, an erasure coding-based design option to provide fault tolerance with Key-Value SSD arrays. Chapter III introduces KVRangeDB, a fast, resource-efficient range queries capable database for Key-value SSDs. In Chapter IV, we propose a generic FPGA accelerator for the emerging Minimum Storage Regeneration (MSR) codes erasure coding based storage redundancy schemes. Chapter V demonstrates explorations for leveraging logical key remapping to provide multi-tiering on single key-value device for Quality of Service (Qos) purpose and I/O and storage load balancing for Key-Value SSD arrays. Finally, we conclude this dissertation in Chapter VI.

2. EFFICIENT ERASURE CODING MANAGEMENT FOR KVSSDS *

This chapter presents KVRIAD which manages the erasure codes on the KV object level to provide high performance and high reliability to the emerging KVSSD devices. First, we introduce some background knowledge of KVSSD and erasure coding. Then, a detailed design and implementation is described in the following two sections. The evaluation section compares our KVRAID with the state-of-art software based KV stack on block devices.

2.1 Introduction

Persistent Key-Value (KV) stores are an essential component in many large-scale applications [6, 7, 9]. Modern persistent key-value store engines are built on top of block devices (hard drives [32, 17] or flash-based SSDs [18, 33, 34]), resulting in high CPU utilization and I/O write amplification factors (WAF). This high CPU utilization comes from the multiple software stack layers required to translate from key space to device block space including key-value data management, file system, block I/O layer, device driver, etc. [28]. Another problem of modern key-value stores are high I/O write amplification [25, 26]. In order to leverage the performance characteristics of HDDs and SSDs, state-of-art key-value stores use the Log Structure Merge (LSM) Tree [19, 17, 18] as a fundamental data structure to manage key-value objects. Although Log Structure Merge Trees significantly reduce the WAF compared to B Tree [19], it still suffers from relatively high WAF ($\sim$10-40x) due to background compaction [25, 26, 27]. Another source of write amplification comes from write-ahead logging (WAL) to maintain data consistency [17, 28]. Even worse, due to interface gap between the logical block to physical flash page, the device internal Flash Translation Layer (FTL) can also introduce up to $\sim$8x device level write amplification [35].

Recently, both academia [36, 30] and industry [28] have proposed key-value interfaced de-

*Reprinted with permission from "KVRAID: High Performance, Write Efficient, Update Friendly Erasure Coding Scheme for KV-SSDs" by M. Qin, A. L. Narasimha Reddy, P. V. Gratz, R. Pitchumani, Y. S. Ki 2021. Proceedings of the 14th ACM International Systems and Storage Conference, Copyright 2021 by ACM

vices to replace the conventional software-based key-value engines built on block interface devices. Key-value SSDs simplify the software stack for key-value store applications, reducing their overall CPU/memory usage [28]. Further, by consolidating redundant software in-directions, KVSSDs also reduce the overall write amplification [28]. The reduction of CPU usage and WAF can significantly benefit today's cloud environment [31, 37].

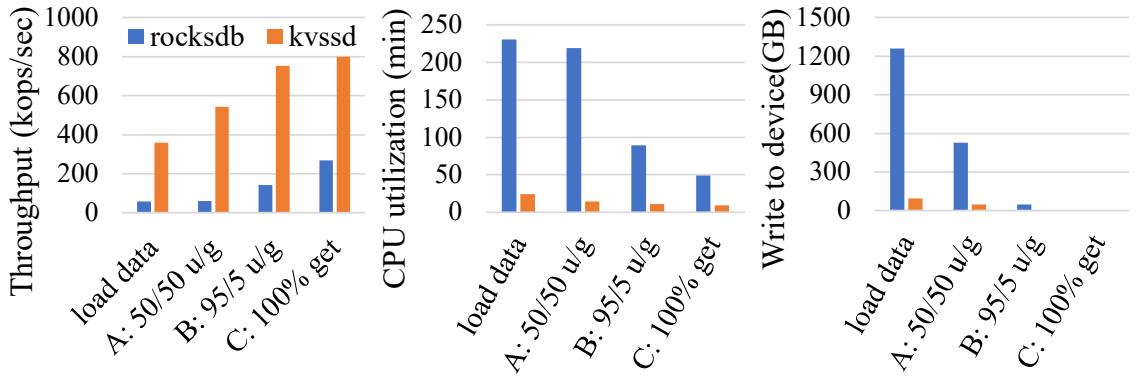


Figure 2.1: Performance comparison between RocksDB and native KVSSD under YCSB workloads.

With SSD arrays, the high CPU utilization and high WAF of software KV stores will further prohibit performance scale up as well as device lifetime. In Figure 2.1 we demonstrate the results of experiments comparing RocksDB and native KVSSD on YCSB workloads in a SSD array environment. RocksDB runs on ext4 file system with 4 block devices (share same hardware with KVSSD but with block device firmware) under Linux RAID0 (striping). Here, we configure RocksDB to use direct I/O to bypass page cache to make a fair comparison with KVSSD which doesn't use any host-side cache. For KVSSD, we hash the request keys to randomly distribute objects to different devices.

In the experiments, we load 50 million records with sizes between 100B and 4KB ($\sim$100GB data in total) and collect throughput, CPU time and the total data writes to the devices. Compared to RocksDB, KVSSDs array outperforms RocksDB on software RAID by $\sim$6x in case of throughput and provides reductions of $\sim$10x in CPU time and $\sim$11x in write amplification. In a practical block

RAID environment with redundancy (such as RAID5 or RAID6), the erasure code calculation and updates will further decrease the overall throughput performance and increase WAF. Detailed experiments setup and more results are discussed in Section 2.5. Such observations motivate us to consider building efficient data redundancy scheme directly on top of KVSSDs.

In this work, we explore an erasure coding-based design option, to provide fault tolerance with Key-Value SSDs, aiming for higher storage efficiency and lower write amplification, to accommodate the write endurance limitations of flash devices. This design option may be viable in different environments (local versus distributed) and offer different trade-offs (smaller storage costs versus higher recovery times). This design provides another option along with currently existing replication.

Parity-based protection groups several blocks, or objects, in a parity group and protects those blocks or objects by storing the parity, or erasure code, of those items in other devices. Erasure coding is straight-forward when the objects or records are all of same size. Several objects or records can be grouped into a parity group that can be protected together using an erasure code. In a Key Value store, however, the records may not be of equal size. How do we protect records of different sizes in a parity-based protection scheme?

Further, application generated key spaces tend to be large (typically 128 bits or larger). Thus, keys tend to be distributed sparsely over the key space. Randomly generated keys may not correlate with each other and there may not be any locality in the generated keys, thus making it difficult to form parity groups based on the application keys. When application keys are employed to form parity groups, because of their properties of sparsity and no locality, it is likely that most parity groups would have one or very few keys. Thus, parity computed on that group will effectively lead to replication in most cases.

In regular storage systems employing block addresses, forming parity groups is fairly straightforward. Blocks are typically distributed in a round-robin fashion across the devices. The parity group of a given block address can be easily calculated. Similarly, given a parity group, it is easy to identify the block addresses belonging to that parity group. These make constructing and main-

taining a parity group simple in a block-based storage system employing erasure coding-based protection. By contrast, these problems need to be addressed to enable effective erasure coding-based data protection in a KV Store:

- How should a parity group be formed given that the keys could be generated randomly by the application?
- How do we identify the membership of a parity group?
- How do we compute the parity group that a given key belongs to?

One of the main costs of replication is the storage space overhead. In addition to the space overhead, when SSDs are employed, replication results in write amplification and hence lower lifetime since these flash devices have limited write endurance. One of the goals of this work is to explore new design points for key-value stores that can reduce the impact of the extra writes needed for data redundancy. While it is easy to see that erasure coding can be more efficient on initial writes when records are stored, various factors such as out-of-place updates and garbage collection needs to be factored into addressing the total impact.

Table 2.1: Object amplification comparison for *insert* under D data and P codes/parities configurations with 16B key and 1KB value. KVRAID-P demonstrates packing two logical objects into a physical object "Reprinted from [1]".

(D,P)	REP	KVMD [38]	SF [39]	KVRAID	KVRAID-P
(4, 2)	3	4.5	1.8	1.5	0.75
(8, 3)	4	5.38	1.78	1.38	0.69

Current proposals for managing redundancy on key-value storage devices focus on write once, read dominate workloads. KVMD [38] proposed different redundancy schemes based on the value size of a KV pair. KVMD employed replication for small value sizes (less than 128B), and for large value sizes (16KB or above), employed splitting of the record across multiple devices with

Table 2.2: Overall I/O amplification comparison for *update* (same configuration as Table 2.1). KVMD and SF require read metadata object and read-modify-write data/code object operations for updates. For KVRAID, small ϵ (near 0) relies on better garbage collection, as discussed in Section 2.4.3 "Reprinted from [1]".

(D,P)	REP	KVMD [38]	SF [39]	KVRAID	KVRAID-P
(4, 2)	3	5	5	1.5+ϵ	0.75+ϵ
(8, 3)	4	7	7	1.38+ϵ	0.69+ϵ

associated erasure codes [40, 41, 42] to improve storage efficiency. The data and code blocks of the erasure codes can share the same user key and spread across different devices. However, For value sizes in the middle (128B to 16KB), both mirroring and splitting introduce unacceptable overhead, such as storage inefficiency (mirroring), read/write performance degradation (splitting). KVMD [38] proposed stateless packing mechanism to group multiple objects into an erasure codes/parity group (or stripe). However, to maintain the membership of each user object to the stripe, KVMD needs to create a small metadata object for each user object to keep track of the keys for other user objects in the same stripe. In order to ensure fault tolerance of metadata, KVMD further replicates each metadata object which introduces not only byte level write amplification but also object level write amplification. Table2.1 shows KVMD requires $\sim$30-50% more objects than replication (REP) for insert.

To address the metadata amplification issue for the KVMD packing mechanism, StripeFinder [39] proposed an efficient metadata membership tracking method to reduce the metadata size as well as the number of metadata objects required. The main idea is to use a circular chain to keep track of the stripe membership, and group multiple metadata objects into a single object through hashing on the user key to reduce the number of metadata objects as shown in 2.1.

Both KVMD and StripeFinder introduce complexity for updating an object in a stripe. KVMD employs **in-place** update mechanism, which means it needs to read back the code blocks in the stripe and update multiple code blocks as well as the user object, which introduces significant read/write I/O amplification. The update throughput of KVMD is an order of magnitude slower

compared to put and get. To make matters worse, if the update object's value size changes, it may cause an unbalanced stripe that leads to extra performance and storage overhead. StripeFinder doesn't discuss the update scenario. However, in theory, it encounters a similar issue as KVMD. Even if KVMD and StripeFinder employ an **out-of-place** update mechanism by creating a new stripe for the updated object along with the new incoming objects, there will be significant read-/write amplification for updating the associated metadata objects to update the stripe membership information.

In this work, we focus on records of small to medium size(128B to 4KB) which can greatly benefit from grouping multiple objects into one erasure code group. We explore an alternative way to keep track of the membership information for each stripe by translating the user/logical keys to physical/device keys. The physical keys are designed as 64-bit monotonically increasing numbers which enable easy identification of a stripe. Given a stripe, it is easy to compute the physical keys on rebuild. We leverage well-established in-storage data structure LSM-Tree [19, 17] to map the logical keys to physical keys. The LSM-Tree introduces acceptable write overhead and amplification since it allows converting a logical key to physical key (as a KV pair) to a large I/O (large value size in KV devices), which also significantly reduces the number of objects required for metadata compared to KVMD and StripeFinder. With compression which is well established in existing LSM-Tree implementation such as LevelDB [17], RocksDB [18], the metadata write overhead can be further reduced.

The introduction of logical keys to physical keys mapping enables packing multiple objects into a single physical object within an erasure code group. Recent studies on KV-SSD show noticeable read/write performance scale-down ($\sim$15-20%) as the number of objects managed in the KV-SSD grows. The reduction of number of objects on the devices through our design can help maintain performance as the number of user objects scales. Table 2.1 shows that KVRAID-P (packing two logical objects to a physical object) can reduce object amplification by more than 2x for insert compared to StripeFinder (SF) and by more than 4x for update compared to KVMD and StripeFinder.

This work makes the following contributions:

- Presents a design of a KV store employing parity or erasure coding for data protection or fault tolerance.
- Propose several novel ideas for organizing such a system, among them, slab allocation within the KV store, logical to physical key mapping and maintaining multiple states of data within the system.
- Evaluates the proposed system on real KVSSD devices to show that the proposed solution can reduce storage and CPU load overheads for data protection, while improving lifetime of the system through reduction of write operations.

The remainder of the section is organized as follows. Section 2.2 describes the background in software based key-value stores, KVSSD technology and erasure coding. Section 2.3 gives an overview of the KVRAID design and Section 2.4 describes the implementation of KVRAID in detail. Section 2.5 evaluates KVRAID. Section 2.6 discusses the related work. Section 2.7 summarize the chapter.

2.2 Background

KVRAID's focus is building an erasure code management mechanism for KVSSDs. In this section, we first provide a brief overview of software key-value store engines and the KVSSD devices. Then we introduce the background of erasure codes.

2.2.1 Software key-value store engines

Modern Key-value stores applications [6, 43, 7] rely on software key-value store engines to translate the key-value interface to a block interface for HDDs/SSDs. State-of-the-art software key-value store engines [8, 17, 18] leverage an LSM-tree structure [19] to optimize performance on block devices.

LSM-trees organize small objects into multiple levels of large, sorted tables (SSTable). All writes will perform out-of-place update-and-writes to the top-level table. Reads will search from

the smaller top-level table to the larger bottom level table to find the most updated data. A major advantage of the LSM-tree design is small writes are converted to large sequential I/O which is optimal for HDDs/SSDs performance. However, this comes at the cost of high CPU utilization and I/O amplification [25, 26, 27]. Considering that, if replication is performed on top of typical software key-value store engines, they will suffer significantly more write amplification.

2.2.2 Key-value SSDs

The idea of key-value interface device has been proposed on both academia [36, 30] and industry [28]. Currently, Samsung provide KVSSD product with a hash table implementation [28] that targets on fast store/get performance and low write amplification. In Figure 1.2 we show the results of experiments we conducted on KVSSD to evaluate performance characteristics for different value sizes from 128B to 2MB (maximum size supported by the device). For all the experiments, we use 64 threads to issue the requests to the device. We sustain each experiment long enough to rule out the device internal caching effects. The experimental setup details are described in Section 2.5.

There are two interesting performance characteristics of KVSSD as shown in the figure.

1. The IOPS performance for both insert and get remain nearly the same for value sizes between 128B to 16KB. (Slightly lower from 2KB to 16KB)

2. The update (overwrite the existing keys) IOPS performance is noticeably lower ($\sim$25%) than insert operations.

The steady IOPS performance for insert and get for smaller records (under 16KB) implies that by packing multiple KV objects in a single I/O, we can increase the overall device bandwidth while not sacrificing latency performance. Further, we can leverage the performance benefit for insert operation (in comparison of update) by our logical to physical translation technique (by always inserting new foreign keys for update operations). Section 2.3 will demonstrate more details of how we leverage these KVSSD performance characteristics in our KVRAID design.

2.2.3 Erasure codes for storage systems

Compared to a replication (or mirroring) scheme which provides data redundancy for storage systems to tolerate failures, erasure codes [40] achieve similar levels of fault-tolerance with less storage overhead [44]. As shown in Figure 2.2, erasure codes stripe a large data block into k even-sized data chunks and encode m same size parity chunks. These data and parity chunks are then spread to $n = m + k$ independent storage nodes. In the remainder of this section, we refer to this kind of code as EC[n, k]. When up to m nodes fail, the failed data chunks can be reconstructed from the surviving chunks.

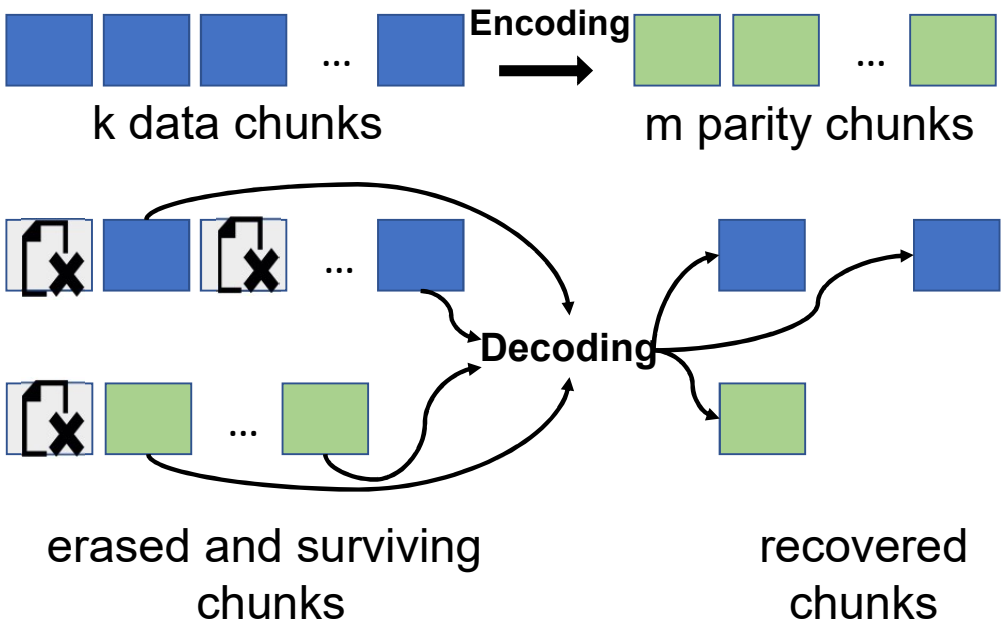


Figure 2.2: Erasure codes example for storage systems, original data is striped to k data chunks and m parity chunks. When less or equal than m nodes failed, the erased chunks content can be decoded by the surviving data/parity chunks.

The encoding/decoding procedures are linear arithmetic operations in finite fields. For example, see the widely used Reed-Solomon codes (RS code) [45]. The encoding procedure can be represented as equation 2.1. The decoding procedure for m-node failure ($m \leq n - k$ which is the maximum number of nodes failure that the codes can tolerate) can be achieved by solving the

linear equation 2.1 as shown in 2.2. Different erasure code schemes follow similar principles as RS codes, but may apply different coefficients combinations.

When we make an update on the original data, the update can affect one or more data chunks. From equation 2.1 we know that any update on the data chunk will affect all the parity chunks. Since the encode equation 2.1 is a linear equation, we can simply update the delta value of the data chunks to the parity chunks. For storage devices, this means we need at least two reads (data chunk and parity chunk) and two writes to finish the data update. In real flash storage devices, partial writes are infeasible, thus we need to pay a whole I/O for the delta update to both data chunks and parity chunks, which will cause significant write and read amplification.

$$\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & c_1^1 & \cdots & c_1^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & c_{m-1}^1 & \cdots & c_{m-1}^{k-1} \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} \tag{2.1}$$

$$\begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_m \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & c_1^1 & \cdots & c_1^{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & c_{m-1}^1 & \cdots & c_{m-1}^{m-1} \end{bmatrix}^{-1} * \begin{bmatrix} P_1 - D_{m+1} - \cdots - D_k \\ P_2 - c_1^m D_{m+1} - \cdots - c_1^{k-1} D_k \\ \vdots \\ P_m - c_{m-1}^m D_{m+1} - \cdots - c_{m-1}^{k-1} D_k \end{bmatrix} \tag{2.2}$$

2.2.4 Managing multiple objects for erasure coding over KV devices

In this section, we briefly introduce how the state-of-art works KVMD [38] and StripeFinder [39] manage multiple objects in an erasure code group (also referred as a stripe). KVMD and StripeFinder both group multiple user objects into a single stripe and write user data objects and code objects (calculated from the user data objects through erasure coding) across different devices. Figure 2.3 illustrates how KVMD and StripeFinder use metadata objects to keep track of the membership information within a stripe.

In KVMD, every user object is associated with a metadata object (replicated on multiple de-

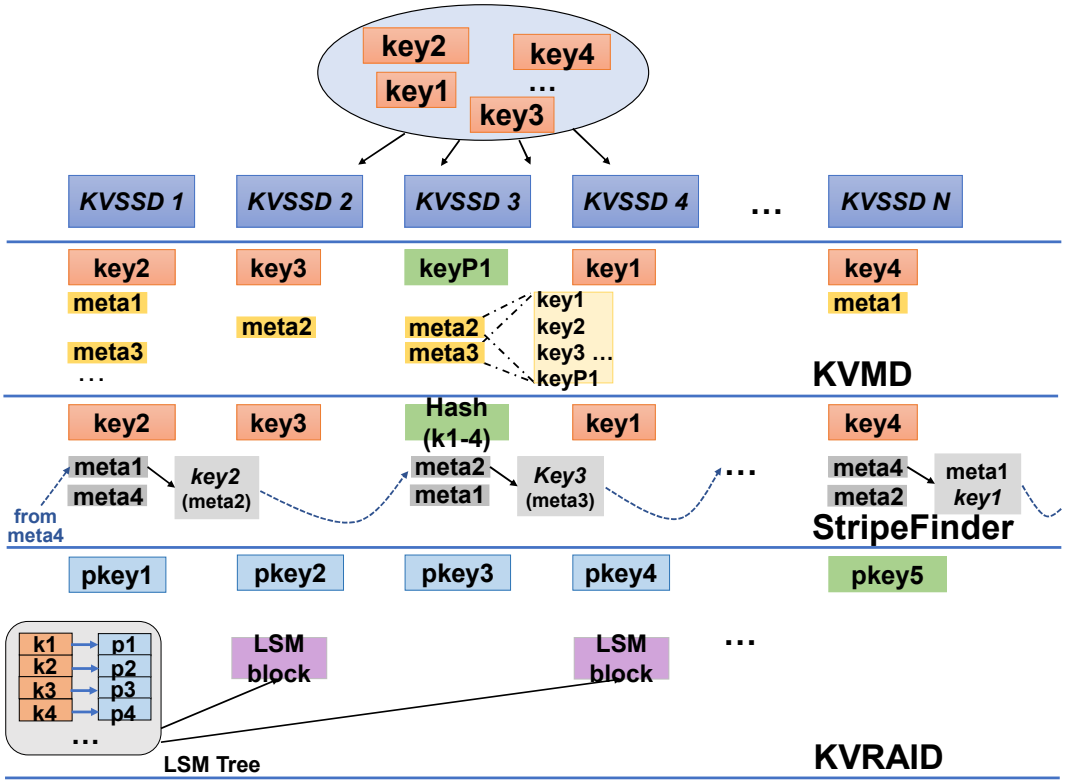


Figure 2.3: Comparison of KVMD StripeFinder and KVRAID on how to manage multiple objects in a single erasure coding stripe "Reprinted from [1]".

vices for fault tolerance). The metadata object key is assigned based on the user key. (For example if the user key is key1, the associated metadata object key is key1:1). The value of the metadata object stores all the user keys and code keys within the same stripe. When rebuild happens, every surviving user key can retrieve all the data/code objects in the stripe through the information in the metadata object.

StripeFinder takes a step further to reduce the metadata overhead of KVMD. Instead of storing all the data/code objects keys within a stripe in the metadata object, StripeFinder creates a finder object (metadata object) that only stores the adjacent data object keys within the stripe as shown in Figure 2.3 and forms a ring chaining the data object keys within a stripe. During update and rebuild, each user object can walk the ring to retrieve all the data objects keys within a stripe. The code object keys are generated through all data object keys through a hash function. Code key can be easily retrieved after retrieving all the data object keys.. To further optimize, StripeFinder groups multiple finder objects together through hashing (User keys hashed to the same bucket use the same finder object) to reduce the object amplification [39].

As shown in Tables 2.1 and 2.2, while KVMD and StripeFinder improve storage efficiencies compared to mirroring, their performance, especially update performance needs further improvement. Unlike KVMD and StripeFinder, our KVRAID design chooses a different route by translating user/logical keys to physical keys and uses physical keys to keep track of erasure code group information. In the following sections, we will elaborate how the key translation idea helps reduce the object amplification and improve the update performance compared to the state-of-art works.

2.3 Design Overview

KVRAID is a data redundancy scheme for key-value interface SSDs (KVSSDs). It exposes a simple key-value interface with high reliability to the user and hides the redundancy management complexity underneath. There are fundamental differences between block I/O semantics and key-value semantics when applying erasure codes. Compared to block I/O and block devices, which operates on continuous block addresses (BAs) and fixed size blocks, key-value devices operates on arbitrary key and value size. Thus, making it difficult to directly mange block based erasure

codes on key-value devices. Such problem has also been discussed in prior works [38, 39]. Our KVRAID design focuses on small to medium size objects (128B to 4KB). We propose to translate logical keys to physical keys and efficiently manage erasure code group membership information through physical keys. Such a design enables the possibility of packing multiple logical objects into a single physical object to maintain high write throughput and reduce the number of objects managed by the device.

2.3.1 Semantics

KVSSDs provide a simple key-value interface (store, retrieve, delete) to simplify the software stack for key-value store applications. After building our data redundancy management scheme, KVRAID retains this simple key-value interface for users while optionally providing an iterator interface with user-defined order for range queries without direct device support. We define the following semantics for our KVRAID:

1. ***put (k, v):*** Insert new key-value pairs.
2. ***update (k, v):*** Update existing key-value pairs.
3. ***get (k):*** Retrieve value from key.
4. ***delete (k):*** Delete key-value pairs.
5. ***scan (k1, k2):*** Range scan key-value pairs through iterator (optional).

2.3.2 Parity group formation with packing

As discussed in the background section, erasure codes stripe original data to k data chunks and apply finite field operations on the k data chunks to generate m parity chunks. Finally, data/parity chunks will be spread to $n = m + k$ independent devices.

Generally, there are two approaches to apply erasure code on key-value stores:

Splitting Split the single key-value object into k data chunks and generate m parity chunks and store each chunk in $n = k + m$ devices with the same key. The benefit of this approach is that we

don't need to manage any metadata for the parity group. We can use the user key directly as the physical key for each device to store the data/parity chunks.

The main problem of the splitting approach is I/O complexity. It will separate a single I/O into multiple smaller I/Os on both *put/get* scenarios. For smaller objects, this will hurt the overall performance significantly. So splitting is better applied on large objects (several hundreds of kilobytes or more).

Packing Pack multiple objects into a single parity group. Apply erasure code on packed objects. The benefit of packing is that it can reduce the I/O complexity when compared to splitting. However, packing introduces an indirection between the original key of the object (logical key) and the packed key (physical key). This requires additional metadata and corresponding management. Packing variable size objects also brings other problems. First, segmentation. When applying erasure parity on packed objects, a single object may be cut off into multiple code chunks. Second, packing may incur re-write overhead when an object is updated with a different size, requiring a read back and repacking of the whole parity group.

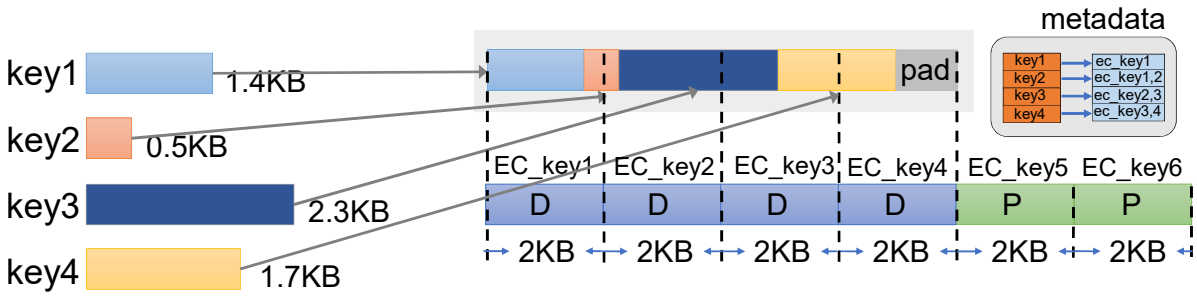


Figure 2.4: Packing approach for parity group formation.

Figure 2.4 shows an example of packing four different sized objects into an 8KB parity group, while applying $EC[6, 4]$ on it. The erasure code chunk size is 2KB. As Figure 2.4 shows, we must keep a small amount of metadata to map user object keys to actual data/parity chunks. For future *get/update* operations, we need to index the physical key from the user key first before accessing

the actual data from the device. For the segmentation issue, taking an example of getting object key2, we need to go through two I/Os (the first and second data chunk). Further, if key2 needs to be updated to a larger object (more than 0.5KB), the whole 8KB parity group needs to be read, re-pack and re-write (plus parity update). This will cause significant read amplification and more importantly write amplification which will impact the device lifetime.

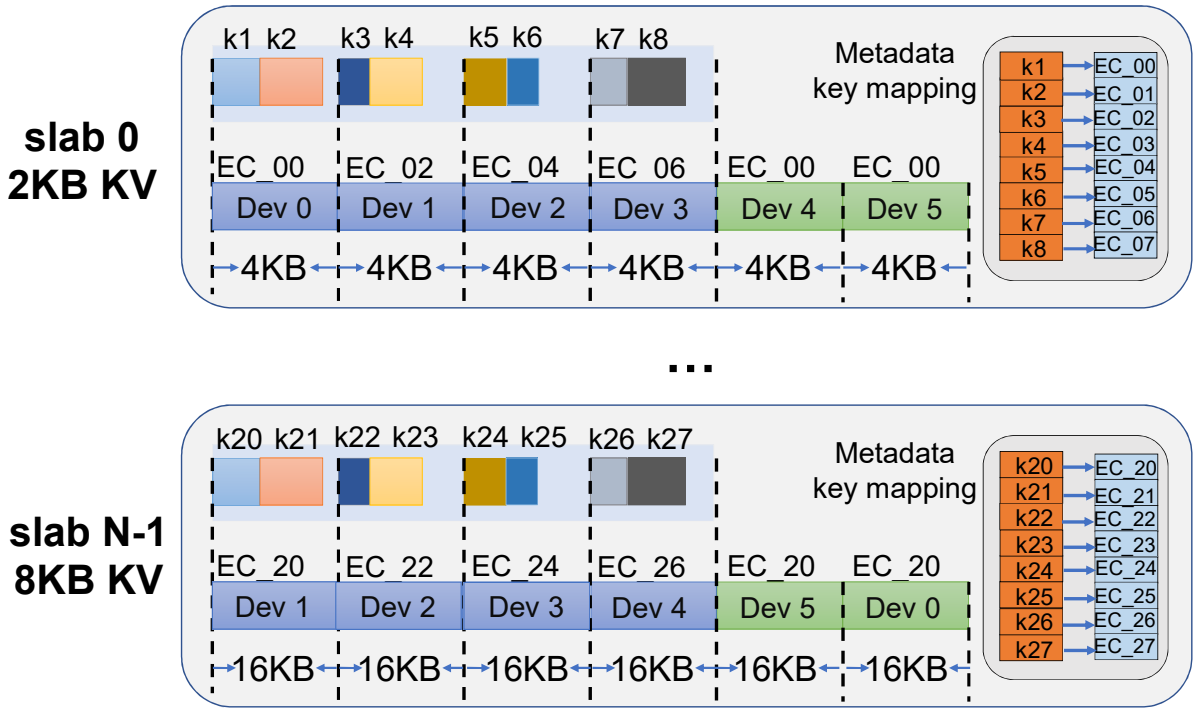


Figure 2.5: Packing KV objects with variable size *slabs* for parity group formation "Reprinted from [1]".

Our design aims to work across a wide range of object sizes. Our approach is inspired by a typical O/S memory allocation policy, the “slab allocator”, together with an enhanced packing approach to achieve storage efficiency while also minimizing I/O complexity. Figure 2.5 demonstrates how our design works. To handle the variable size objects in key-value applications, we pre-define multiple fixed size *slabs* according to the object size distribution of the applications. We assume we know the maximum size of the object as the max slab size. Then, we group the objects

in two dimensions. First, based on the observation that KVSSD has similar IOPS performance across value size from 128B to 16KB as shown in Section 2.2.2, we accumulate and pack multiple similar size objects to form a data chunk for erasure codes. Second, we group multiple of those data chunks to form a parity group and apply erasure coding on these chunks. In order to reduce the segmentation of objects across devices, we align each data chunk to an erasure code chunk size while tightly packing objects within a data chunk (since we need to read/write them in one shot anyway). In summary, our approach has the following advantages:

- **Device Bandwidth Utilization Efficiency** Since we pack multiple objects in to a single data chunk, this has the effect of increasing overall I/O size to the device, which can increase the device bandwidth utilization due to our observation in Figure 1.2.

- **Storage Efficiency** Due to the multiple slab sizes design, we can reduce the storage overhead for the fixed size parity chunks. Note: we still write the variable sized data chunks to device rather than padding them to align with the parity chunk size. The zero padding can be applied in memory during decoding computation.

- **Less Object and I/O Amplification** By packing multiple logical objects into one physical object (data objects in the parity group), we effectively reduce the number of objects managed in each KV device compared to KVMD and StripeFinder.

2.3.3 Batch writes

Now we consider how new parity groups are formed for the EC[6, 4] configuration when there is a new *put* request. We first assign a unique key to this new request, this physical key is a combination of the slab id and unique sequence number. The sequence number will determine the parity group data (device key and device id for the erasure code chunk) for this request. Then we pack zeros for remaining data chunk and other data chunks and calculate the erasure codes in memory. Finally, we write the data chunks (we don't need to write the padding zeros to device) and the erasure code chunks to separate devices. When another request arrives in the same slab, we need to take the following steps, read the old erasure code chunks, pack the new object with

the previous object, re-calculate the new erasure codes, write the new data chunk and update the erasure code chunks in the devices.

In our design, we use batch writes or *big writes* to address this I/O overhead. We will accumulate the data in memory until sufficient number of objects arrive to form a full parity group before we issue the real I/Os for data/code chunks and commit all the requests. In this case, forming each parity group costs 6 writes in total. This *batch writes* technique is widely used in the storage systems [9, 7]. In order to limit the impact on I/O latency, we bound the time for forming a full parity group, at which time available objects are written as a partial parity group to the devices. Application writes are not returned as completed until all the writes have been sent to the devices, thus providing reliability for completed writes.

2.3.4 Lazy deletion

Consider an update operation. This record is already part of an existing parity group. In order to update this record, we have to update both the data chunk it belongs to and the two code chunks (parities) as well in case of EC[6, 4]. This again converts one write into multiple read and write I/Os (data chunk and corresponding code chunks).

In order to avoid these problems, we always update objects *out of place*. We treat an update as a deletion of the old object and an insert (put) of the new object (into a new parity group). To facilitate this approach, an object can have three states on the device: *Valid-alive*, *Invalid-alive* and *empty*. A valid-alive object is a valid record. An Invalid-alive object is only maintained on the device to maintain the parity group consistent and for recovery reasons. An empty record (within the empty parity group) will be deleted from device periodically. With such out-of-place updates, we can combine put and update operations together into a batch writes operation with appropriate metadata updates. The invalid-alive objects are garbage collected later in order to create empty groups that can be deleted from device. This lazy deletion approach allows parity groups to accumulate more deletions over time to make garbage collection more efficient. Figure 2.6 shows an example of how our lazy deletion approach works. The accompanying write reduction benefit will be demonstrated in Section 2.4.3 quantitatively.

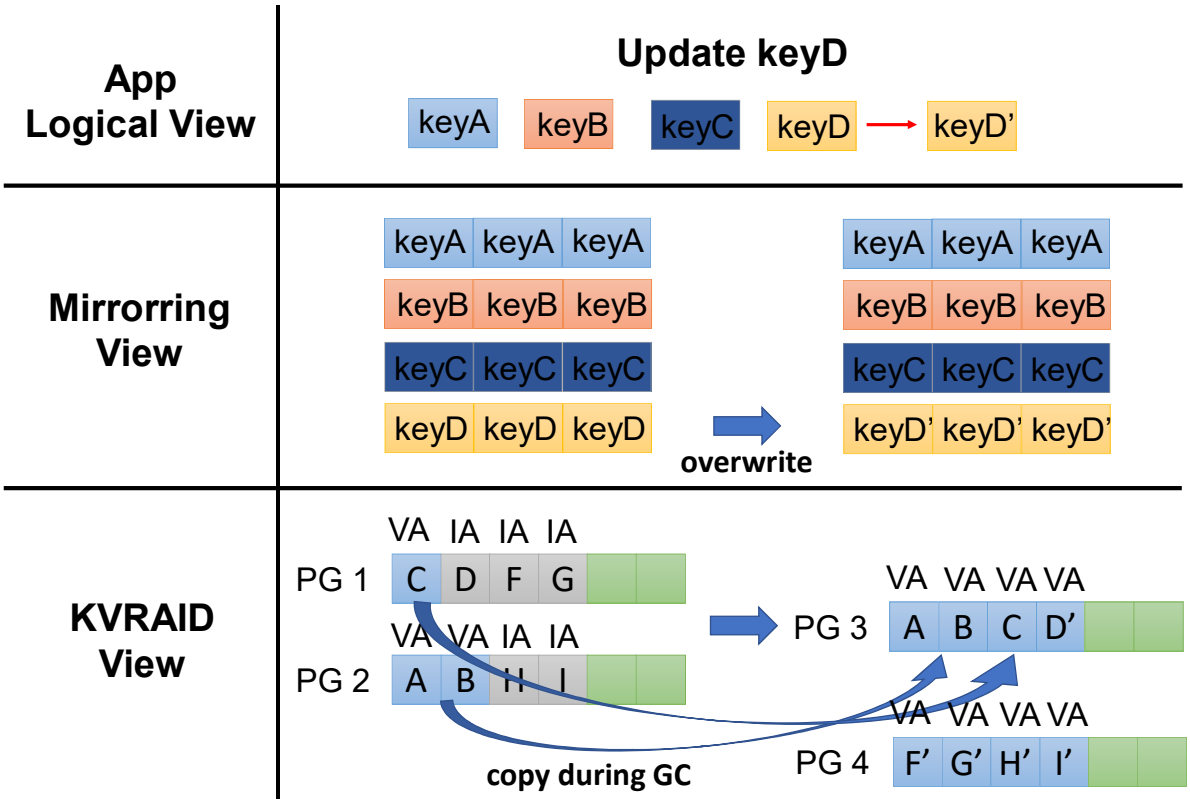


Figure 2.6: Lazy deletion approach demonstration "Reprinted from [1]".

Here we demonstrate an example of how our lazy deletion approach works and show the accompanying write reduction benefit. Figure 2.6 shows object updates from application's view (from keyD to keyD′). The Replication approach just overwrites the triple copies of object keyD to keyD′. The average write cost for *updates* is 3. In our KVRAID approach, the updated objects will be written to new parity groups PG 3&4 in a batch. The garbage collector in background will eventually clean the previous partially full parity groups PG 1&2 and copy the *Valid-alive* (VA) objects to the new parity group. Although the extra copy of the *Valid-alive* objects during *updates* will cost extra write overhead (2.4 writes per updates as shown in Figure 2.6). However, the amortized write cost for *put* and *update* operations is still considerably less than triple replication while still maintaining significant storage efficiency.

2.4 Implementation

KVRAID was implemented from scratch in C/C++ as a user space library with simple key-value interface as shown in Section 2.3.1. It supports different redundancy levels (we evaluate EC[6, 4]) and different erasure coding schemes (we use Reed–Solomon code for evaluation). Section 2.3 focused on the design principles of how we apply erasure coding to variable size key-value objects (parity group formation) and how to optimize to reduce write amplification (batch writes and lazy deletion). In this section, we focus on the implementation issues and deliver a more complete picture of our KVRAID design.

2.4.1 Metadata Management

The key idea of KVRAID is dynamically managing a translation table of user-side logical keys to device-side physical keys. By classifying the objects by value length, KVRAID groups the similar size objects into a fixed-size slab and forms the parity group according to write request order, then applying erasure code on it. Figure 2.7 demonstrates the core structure for metadata management. When an application write request (*put* or *update*) comes, KVRAID determines which slab the object will fall in by object size. That slab will assign a unique physical key for accumulated request objects and apply packing and erasure coding on them. The logical (user) key and physical (device) key mapping will be filled into the mapping table. Finally, the packed data chunks and code (parity) chunks from the erasure coding for each parity group will be written to separate devices as KV objects.

2.4.1.1 Slab

The slab contains a *request queue* which accumulates the incoming write requests and dequeues requests in a batch to form a parity group. We use a global monotonically increasing unique group id number for each parity group after packing and erasure coding of the packed objects. The code and parity objects (physical objects being written to device) will use the unique group id number and slab id to construct the physical key. Figure 2.8 shows an example of how physical key and value are constructed before writing to the devices. The 56bit sequence number is calculated as

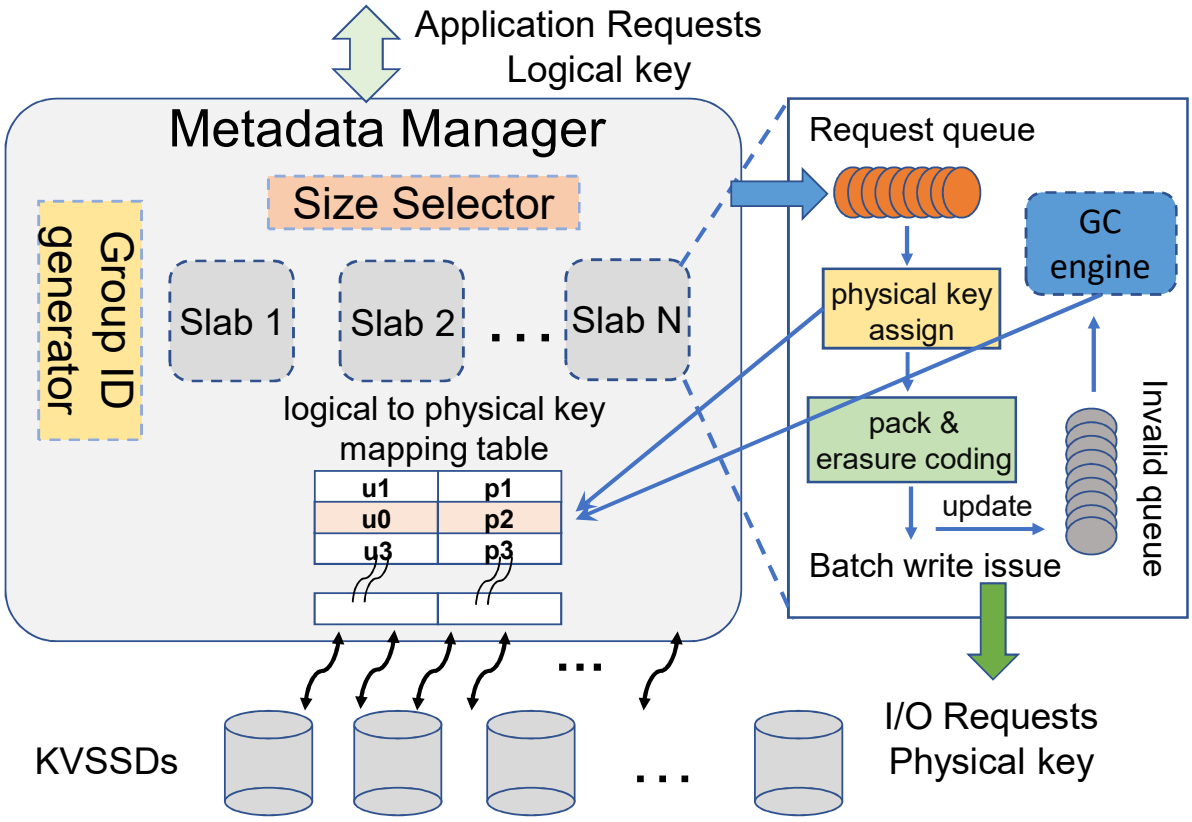


Figure 2.7: KVRAID metadata management data structure layout "Reprinted from [1]".

$\mathsf{group}_{id} \times k + \mathsf{offset} \times p$, where k is the number of data objects in an erasure code group, p is the number of records packed into a single erasure code object and offset is the relative position of the erasure code objects.

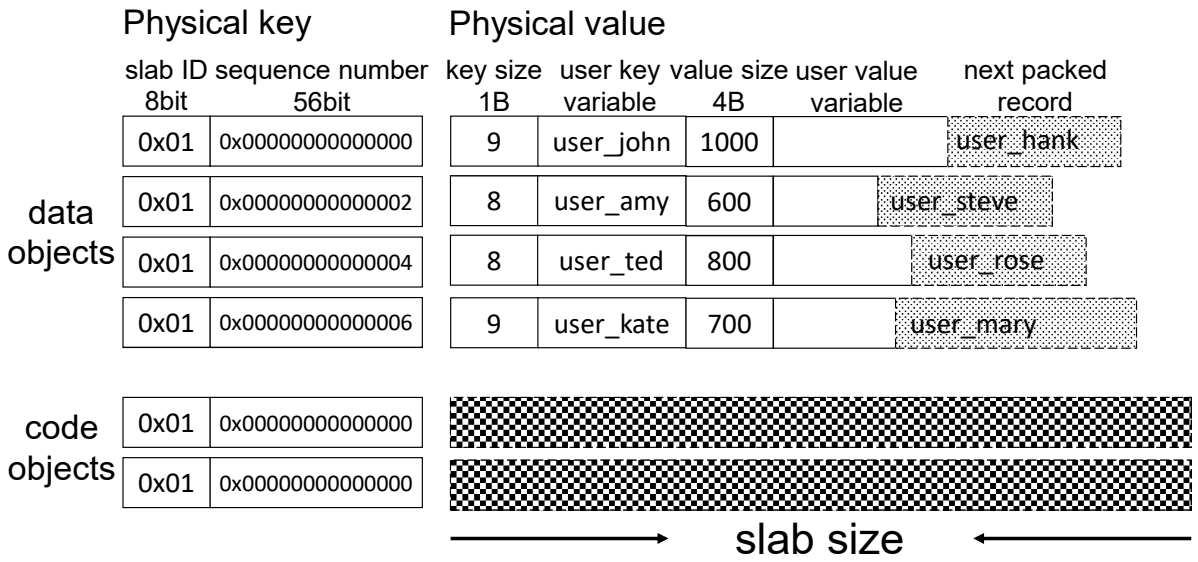


Figure 2.8: Physical key/value format "Reprinted from [1]".

The monotonically increasing physical keys can also help determine the device membership for each physical object (physical objects are assigned to devices with round-robin algorithm like RAID6 [46].) Besides, monotonically increasing keys also provide a versioning effect which will help us for crash recovery (see Section 2.4.4). For the physical value, we don't need to actually write the padding zeros used in erasure coding to device (referred as virtual zeros padding in KVMD [38]) since we are aware of the slab size (erasure code object size) and the user value size (embedded in the physical value). We also embed the user key into the physical value for future garbage collection and crash recovery (see Section 2.4.3, 2.4.4).

For slab size choice, ideally we can use prior knowledge on the dataset value size distribution to find the optimal slab sizes to minimize the erasure coding computation and storage overhead as mentioned in 2.3.2. Our initial evaluations have shown that the results are not highly sensitive to the slab sizes.

2.4.1.2 Mapping Table

The mapping table maps each user logical key to a physical key to get the erasure code data object and the internal packed position (offset) of the record. As shown in Figure 2.8, the physical key of user_john and user_hank will be 0x0100000000000000 and 0x0100000000000001 respectively. By accessing the mapping table, we can extract the slab id, physical key of the erasure code data object and the packed position with in the data object to retrieve the user record.

In our design, we examine two possible approaches for persisting metadata. First, we propose to employ persistent memory (like Intel Optane [47]) to store the metadata. In this case, we can leverage in-memory data structure like hash-table and balanced trees to store the metadata to achieve minimal metadata overhead for insert, update and read.

Second, when NVM is unavailable, we propose to use the in-storage data structure LSM-Tree [48] to store the metadata externally on KVSSDs with redundancy. The LSM-Tree structure converts small writes (logical key to physical key pair) to larger writes (typically 16KB block size for LSM-Tree implementation). Such a design significantly reduces the metadata write overhead and number of metadata objects required in the device compared to KVMD and StripFinder which require metadata objects proportion to the total number of logical objects managed in the system. For example, for an EC[4,2] configuration with 100 million 16B-key objects, KVMD requires 300 million metadata objects and StripeFinder(C=10) requires 30 million metadata objects, while KVRAID only requires 0.22 million metadata objects(16KB block size for LSM-Tree without compression).

In our implementation we port LevelDB [17] to KVSSD storage. The levelDB on-disk structures (such as manifest, sstables blocks, etc.) are stored as key value pairs in KVSSD. To provide redundancy to the mapping table, we split the large SSTable blocks (64KB) to multiple objects with erasure coding and store each object to a separate KVSSD. Other levelDB related files (stored as KV pairs) are replicated. We also store a magical KV record (with replicas) to indicate the existence of in-storage mapping table.

2.4.1.3 Invalid queue

To facilitate the lazy deletion, each slab also maintains an *invalid queue* from the *update* and *delete* requests for further garbage collection. To enable efficient garbage collection, the invalid queue is implemented as a hash-table which key is the parity group id in the slab and the value is the group offsets for all invalid-alive records in that group. Thus, the garbage collection thread can linearly scan this structure and identify the near empty group to perform further reclamation. In our KVRAID implementation, we did not persist this *invalid queue* structure. On KVRAID closing, our GC engine will enforce cleaning all the partially full parity groups and moving the valid objects in them to new parity group to compact space utilization (with the assumption that the KVRAID is not frequently opened and closed). Also, we can gracefully recover from crash during this final "compaction" by leveraging the monotonically increasing sequence number in the physical keys. (see Section 2.4.4).

2.4.2 Key-value operations

In this section, we will show how the ***put, update, get, delete, scan*** semantics are implemented.

2.4.2.1 put

We leverage *batch writes* to address the I/O amplification issue for *small writes*. In our implementation, we accumulate *put* requests in the same slab until the parity group is full or a timer expires. Then we will store the data and the parity objects to the devices at the same time. This will avoid an extra *get* and *update* for the parity objects. If we do not have a full parity group before the timer expires, we will write the partial parity group to the devices. Finally, we will insert a new entry the logical to physical keys mapping table appropriately.

2.4.2.2 update

As we know in the *small writes* implementation, we need to update both data and parity objects (if changing slab, we also need to update the parity in the old parity group). With the *batch writes* and *lazy deletion*, the *update* is analogous to the *put* operation described above, except that we need

to tomb mark the updated key (pushing the stale physical key to the invalid queue) and update the stale entry in logical to physical keys mapping table. Since KVSSD devices perform much better for inserting a new key compared to overwrite an existing key 2.2.2, this *out of place* update can help improve the overall performance.

2.4.2.3 delete

Similar as discussed above for the *update*, if we *delete* in a *small writes* fashion, we need to delete the object on the device and update the parity object within the same parity group. With *lazy deletion*, we tomb mark the deleted object and do garbage collection in the background.

2.4.2.4 get

For the non-mutated *get* operation, there are two cases. If no failure happens, it will just lookup the metadata to get the physical key, which is used to get the object from the device. Alternately, when failure occurs, it goes into the recovery procedure as shown in Section 2.4.4. Since our parity group membership information contains all physical keys for the data and parity objects in the parity group, we simply get the other surviving objects in the parity group and decode to recover the request object.

2.4.2.5 scan

One key advantage of our KVRAID design is that it can provide range queries in user-defined order without direct device support. The *scan* operation relies on the implementation of our mapping table described in Section 2.4.1.2. The mapping table needs to be implemented with an ordered structure (like a tree) to support efficient range scan. The *scan* operation will first query the mapping table on the logical (user) key and retrieve the physical (device) key and then retrieve and extract the user value.

2.4.3 Garbage collection

Garbage collection (GC) is a critical component in our KVRAID design to trade-off I/O amplification and storage efficiency. In our current implementation, the garbage collection is triggered

by a storage utilization threshold. A separate thread does garbage collection for each slab in the background periodically. Figure 2.9 demonstrates how garbage collection works in each epoch. As mentioned in Section 2.4.1, we maintain the *invalid-alive* objects in a hash-table structure. The GC thread first scans the invalid queue and find the candidate reclaim groups which are under the minimum invalid alive entries threshold. Then it will read the *valid alive* records in the reclaim groups from devices, extract the user keys and put them into the slab request queue. After the *valid-alive* object in the reclaimed group is committed in a new parity group, the GC thread will update the mapping table with the new metadata. Finally, we can delete the data and parity objects in the reclaimed groups from devices to release device space.

Take an example of the EC[5, 4] configuration, consider 12 objects are updated in four parity groups as shown in Figure 2.9. In the ideal case, update without GC kick-in would cost 1.25 writes per updated-object under the *batch writes* and *lazy deletion* criteria. Reclaiming the 4 parity groups will cost 1.25 writes per *valid-alive* object (4 in total). The average write amplification for those 4 parity groups that were updated becomes $\frac{12\times1.25+4\times1.25}{12} = 1.67$ which is still smaller than the replication cost (2 with the same redundancy level). To be more general, the average write amplification for KVRAID under GC is as follows.

$$WAF = \alpha \times \frac{k+r}{k} + \frac{1-\alpha}{R_{invalid}} \times \frac{k+r}{k} \tag{2.3}$$

In equation 2.3, k is number of data chunks and r is the number of code chunks in erasure code. α stands for the ratio of non-updated parity groups for the overall parity groups. $R_{invalid}$ is the average ratio of the *invalid-alive* objects per parity group ($\frac{12}{16}$ in the previous example) before GC. In our implementation, we can tune the $R_{invalid}$ parameter for garbage collection aggressiveness to trade off between storage efficiency and write amplification.

2.4.4 Recovery

Our KVRAID design aims to provide data protection for common failure cases, i.e. the device failure and the host failure. The device failure can be handled by the erasure code across the

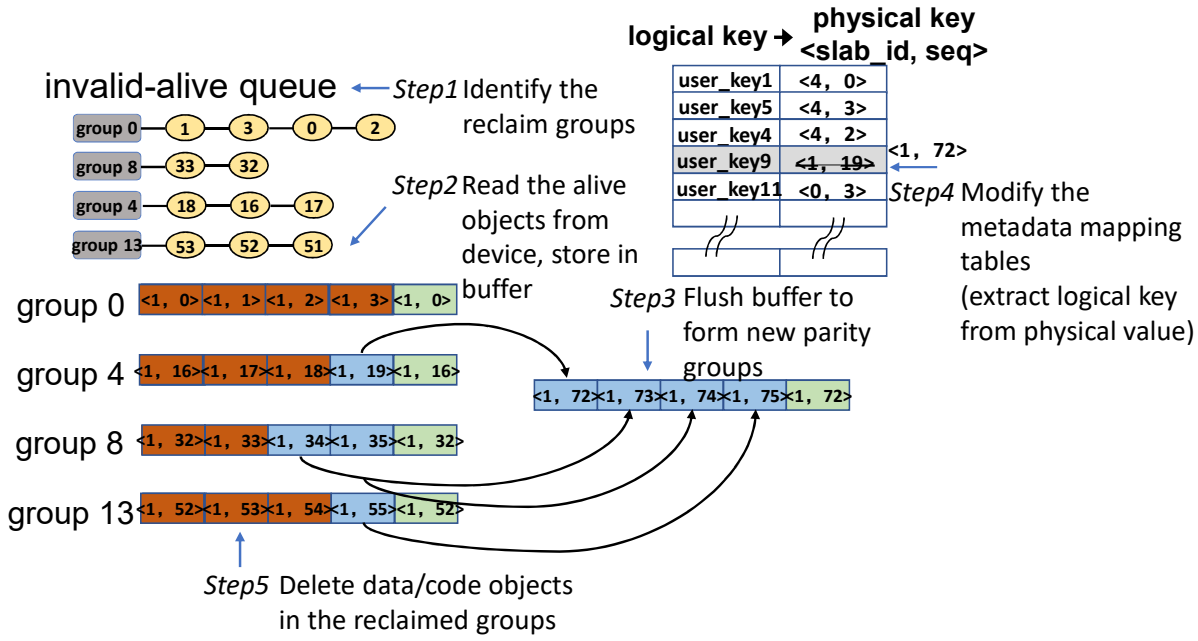


Figure 2.9: Garbage collection implementation "Reprinted from [1]".

devices. However, unlike replication and block RAID systems, our KVRAID design is not stateless (cf., logical to physical key mapping). Thus, we also need to consider host side failures. Currently, our KVRAID design doesn't support recovery from simultaneous device and host failures.

2.4.4.1 Device failure

The erasure code determines the number of device failures that can be tolerated. For example EC[6, 4] can tolerate at most two simultaneous device failures. By the design of the construction of the physical keys, we can extract all physical keys (data and parity chunks) of each parity group from any logical key from the group by indexing the logical to physical key mapping table. On device failure, we can then issue concurrent read I/Os to retrieve the surviving data/parity objects in the parity group for erasure code decoding and recover the erased objects on the failed devices. For the in-storage metadata implementation, the metadata itself is also protected by erasure code which can also be rebuilt.

2.4.4.2 Host failure

Due to maintaining the logical to physical key mapping table, host side failure (either hardware or software) may corrupt the mapping table. Another source of corruption from host failure is the "compaction" process for the invalid-alive queue when closing KVRAID. We may terminate before the "compaction" process is completed or lose the invalid-alive queue data since it's volatile. For the mapping table corruption, we can rebuild the mapping table purely from device since we embed the logical (user) key in the physical value. The recovery process will retrieve over all the physical key value objects on all KVSSDs, identify the data objects (from the physical key) and extract the logical keys inside and rebuild the mapping table. For the invalid-alive queue corruption, thanks to the monotonically increasing sequence number, it implicitly conveys the version information. By comparing all the physical keys' sequence number with the same logical key, we can distinguish the invalid-alive objects and the valid-alive one (with the largest sequence number).

2.5 Evaluation

In this section, we evaluate KVRAID system performance on real KVSSD hardware and compare with state-of-art software KV stacks (RocksDB) on block SSDs with RAID and state-of-art erasure coding management for KV devices.

2.5.1 Experimental setup

2.5.1.1 Hardware and configurations:

We evaluate KVRAID on a real system (Intel Xeon Gold 6152 platform with 256GB DRAM) with six Samsung KVSSDs devices. We evaluate our KVRAID with an EC[6, 4] erasure coding configuration.

For RocksDB (on block SSDs) configuration, we disabled compression to make a fair comparison with KVRIAD (current KVSSD firmware doesn't support compression). We set up 2GB block cache and use direct I/O for flush and compaction which is a common industry setup [49]. For KVRAID, we employ 4 uniformly distributed slabs based on value size range (100B to 4000B). We implemented a host side LRU cache for a fair comparison with RocksDB's block cache.

2.5.1.2 Comparison schemes:

In our evaluation, we compared seven different schemes to thoroughly investigate our KVRAID design. RocksDB-raid10 and RocksDB-raid6 are running on six block SSDs with software RAID. Other 5 schemes are running on KVSSDs. The block SSDs and KVSSDs share the same SSD hardware but with different firmware. All schemes have the same, device-failure tolerance level (tolerate 2 simultaneous device failures).

I **rocksdb-raid10** RocksDB on block devices with Linux nested RAID10 configuration (two raid1 devices each with 3 SSDs, i.e. mirroring, and then apply raid0 on the two raid1 devices, i.e. striping).

II **Mirroring** Two replicas in different KVSSD devices which is equivalent to **Rocksdb-raid10** in case of storage efficiency.

III **rocksdb-raid6** RocksDB on block devices with linux RAID6 configuration.

IV **Small writes IS (KVMD)** Update data and corresponding code objects in a parity group in sequence with metadata stored on storage devices (This resembles *KVMD packing* approach for **update** operations which requires read and update code objects in a parity group. For **insert**, KVMD packing is similar to Batch writes IS. We mainly use this scheme to model update operations of KVMD and StripeFinder.).

V **Batch writes IS** KVRAID which employs batch writes and lazy deletion techniques with metadata stored on storage devices.

VI **Batch writes IM** Same as V except for supplementing with in-memory metadata.

VII **Batch writes with packing IM** KVRAID which employs packing 2 logical objects into an erasure code chunk along with batch writes with in-memory metadata.

2.5.1.3 *Workload:*

We use the Yahoo! Cloud Serving Benchmark (YCSB) [50] to generate our workloads with 64 client threads. We evaluate with dataset with variable length from 100B to 4000B in uniform distributions, which is a good representation of a real dataset. The distribution of request operations is zipfian. To cover a full spectrum of real workloads, we evaluate the systems with YCSB default workloads and different write/read (update/get) ratios, including (90%:10%), (70%:30%), (50%:50%), (30%:70%) and (10%:90%).

2.5.2 Experimental results

We ran the YCSB benchmark on real KVSSD and SSD devices, for each of our experiments, we first load 200 million records with variable value sizes (around 400GB of total data). The key size is ∼25 bytes. Then we perform another 200 million operations with different update/get ratios on the dataset (all keys exist). For KVRAID, we used a soft capacity of 320GB for each KVSSD device to make sure GC kicks in during the run phase. For RocksDB experiments, we perform an extra warm-up phase to finish compactions in the load phase. The raw data written in each experiment varies from ∼0.66TB to ∼2.3TB. Before each experiment, we format the devices to reset the internal device state.

2.5.2.1 *Throughput performance*

Figure 2.10 (a), (b) show the overall throughput performance and CPU utilization (through Linux time utility) for different redundancy schemes (for rocksdb, we collected overall performance due to its irregular compactions, for KVRAID schemes, we collect the steady performance by omitting small period of start and end of each run). In data load phase, RocksDB with software RAID6 performs significantly worse (∼28x in pure insert workloads) with much higher CPU utilization (∼31x) compared to our best KVRAID implementation (VII) on KVSSD. In mixed update/get workloads, KVRAID (VII) still outperforms RocksDB with software RAID by ∼4x and reduces CPU utilization on an average by 4.1x. This demonstrates the performance advantage of our KVRAID design compared to the traditional block RAID on software KV stores. The

saved CPU cycles can benefit other jobs, which is critical in cloud environments [31]. Besides, KVRAID (VII) outperforms KVMD packing (IV) by 3.7x and reduces CPU utilization by 4.6x for update intensive workloads because of batch writes and lazy deletion techniques. (KVMD needs to read-modify-write the code objects for updates).

Within the KVRAID implementations, *batch writes* with in-memory mapping table implementation (VI) achieves similar performance compared to mirroring. Packing 2 objects into an erasure code chunk (VII) can outperform mirroring by 59% for 100% insert case (loading data) and sustain performance for heavy update workloads. For read heavy workloads, batch writes with packing performs slightly worse compared to non-packing due to the latency cost of waiting for more records to group into a parity group.

2.5.2.2 Tail latency

Figure 2.10 (c), (d) show the 99% tail latency for update/get queries. Compared to KVRAID, RocksDB incurs higher update tail latency for write-heavy workloads due to compaction. Even with in-storage metadata implementation (V), KVRAID reduces the update tail latency by 33% compared to RocksDB in RAID6 (III). With packing and in-memory metadata (VII), KVRAID significantly reduces the update tail latency (more than 10x) compared to RocksDB in RAID6 (III). Batching the writes and packing incurs higher update tail latency for workloads with less write traffic. However, thanks to our adjustable backoff timer, the increase of update tail latency for 10% update ratio compared to 90% ratio is less than 47%. Compared to KVMD packing (IV), KVRAID(VII) reduces update tail latency by 22.5x for update-intensive workloads since we avoid in-place-update for code objects.

For get operations, RocksDB requires multiple read I/Os on different LSMT levels, which leads to worse get tail latency. KVRAID(VII) consistently requires only a single I/O for each get query (for in-storage metadata, extra I/O is required for key mapping lookup. However, due to the key mapping size is relatively small, the amortized I/O for metadata access is considerably small). For batch writes with in-storage metadata (V), KVRAID can still reduce get tail latency by $\sim$4.3x compared to RocksDB in RAID6 (III).

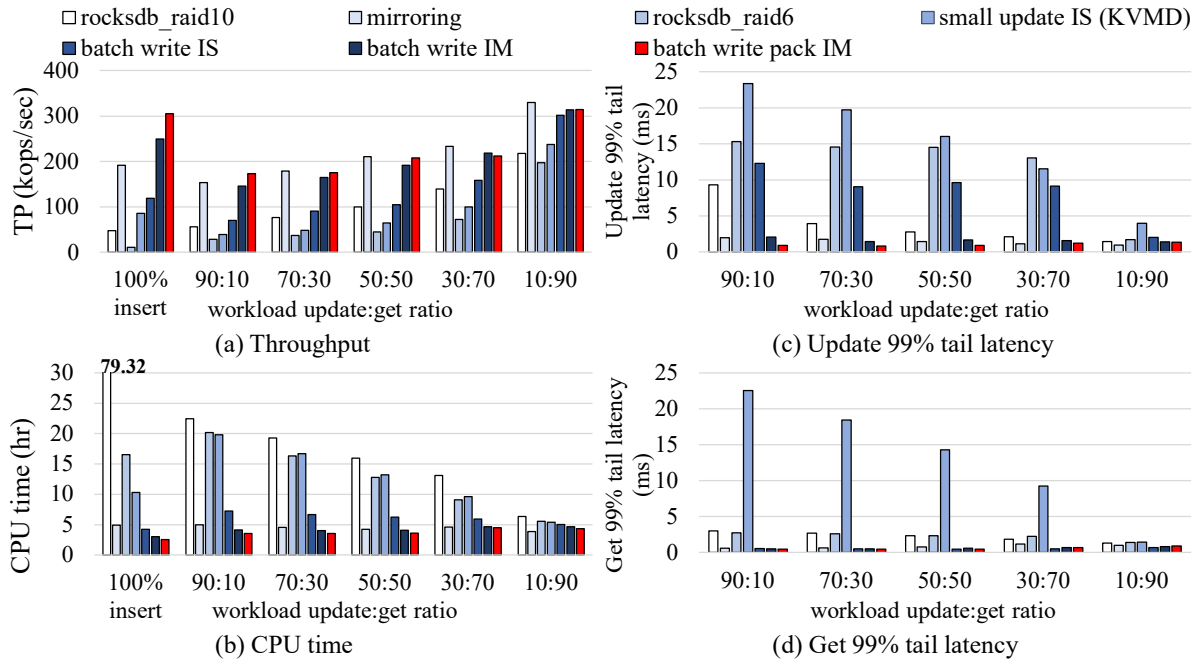


Figure 2.10: Performance and CPU utilization for different redundancy schemes under YCSB workloads "Reprinted from [1]".

2.5.2.3 I/O Amplification

KVRAID's **out-of-place** update design through batch writes and lazy deletion enabled by the logical keys to physical keys translation greatly reduces the read/write amplification compared to state-of-art KVMD [38] and StripeFinder [39]. Figure 2.11 demonstrates the comparison for different redundancy schemes for KVSSDs. Here we only show KVMD results since StripeFinder uses a similar mechanism. Compared to mirroring, KVMD requires more I/Os for an update since it needs to read the old data and code objects to calculate the new erasure code objects. However, KVRAID(V) always forms a new erasure code group for the updated objects which yields much less overall I/O amplification. For 90% update ratio workload, KVRAID(V) yields 1.7x less I/O amplification compared to Mirroring(I) and 5x less compared to KVMD(IV)). For KVMD(IV) the update I/O cost is more than what is shown in Table 2.2 since the variable length object may change value size during update (changes slab), while Table 2.2 only considers fixed length objects. By applying packing technique (two logical objects into a physical object), KVRAID(VII) reduces

overall I/O amplification by 9.6x compared to KVMD(IV). Packing more logical objects will yield better I/O amplification reduction. In our implementation, we didn't use per object metadata design of KVMD(IV). KVMD/StripeFinder will result in more read I/Os for reading the metadata objects.

For normal get operations, KVMD and StripeFinder can use the user keys to retrieve value from device directly. While KVRAID needs to go through logical key to physical keys translation which may require additional I/Os for in storage metadata implementation. However, the indirection overhead for get can be alleviated by caching of the LSM-Tree mapping table. For in-memory metadata implementation, this indirection overhead is negligible.

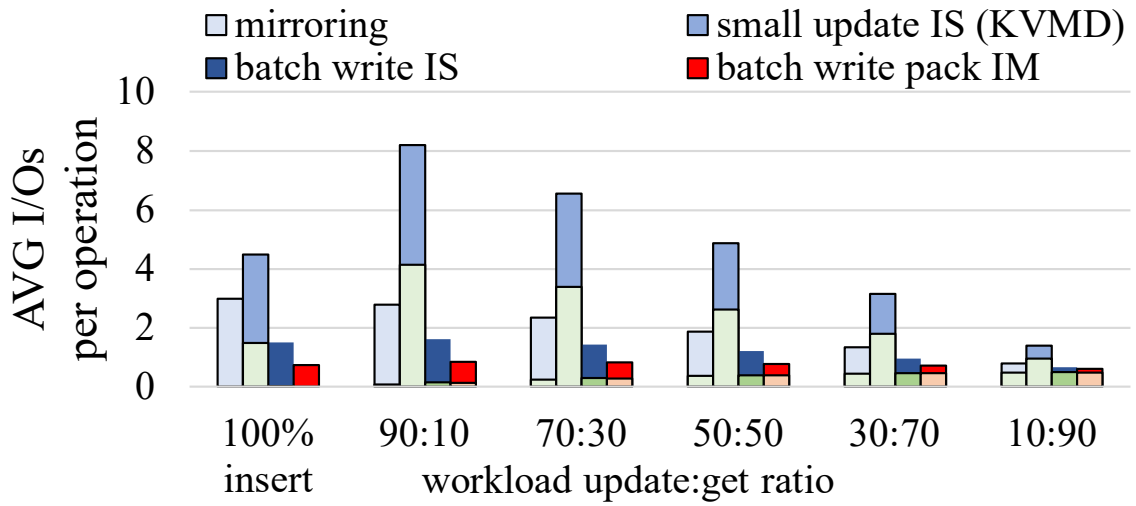


Figure 2.11: Average write/Read I/Os amplification comparison. (Top and bottom half of the stacked bar show the average write and read I/Os respectively.) "Reprinted from [1]"

2.5.2.4 Write Amplification Factor

As the flash technology evolves (from SLC to TLC, or even QLC), device lifetime is becoming a bigger concern [51, 52]. A major advantage of our KVRAID design is reducing write amplification factors (WAF), as a result of erasure coding and our batch writes design. As shown in Figure 2.12, RocksDB on software RAID6 (III) introduces ∼18x WAF in total compared to our best KVRAID implementation (RAID10 is even higher). This is mainly due to the fact that the

multiple software layers (LSM tree, block RAID, etc.) are independent and unaware of each other. Our KVRAID design, however, can directly manage erasure codes at KV object level to comprehensively optimize the WAF. Compared to KVMD packing (IV), KVRAID (VII) reduces ∼3.8x WAF for update-intensive workloads. KVMD needs to rewrite all code objects in the parity group for every data object update, while KVRAID's batch writes keep the WAF similar to pure insert case. Our batch writes design can also take advantage of the data reduction from erasure coding by delaying the code/parity updates (by retaining data as invalid-alive) which reduces the WAF.

In KVRAID, batch writes with packing (VII) significantly reduce the amount of WAF compared to mirroring (II) (63% reduction across all workloads). This advantage can translate into increased device lifetime.

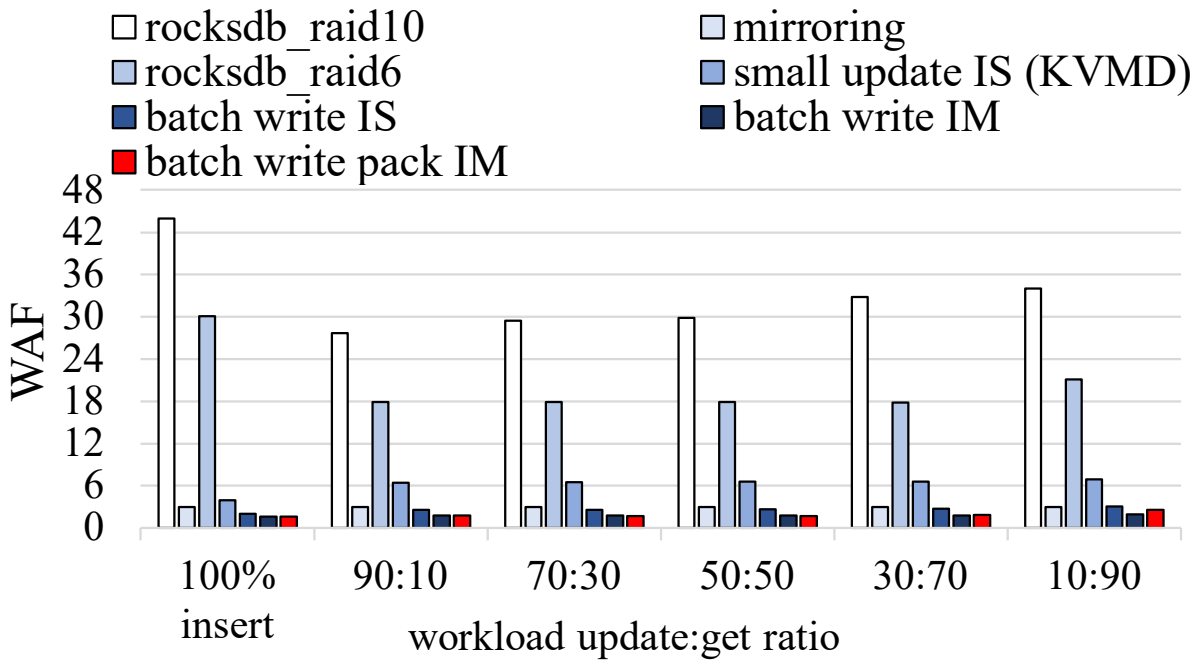


Figure 2.12: Write amplification factor for different redundancy schemes (WAF is measured as total data write to devices from redundancy schemes over the application write data). "Reprinted from [1]"

2.5.2.5 *Storage Efficiency*

Storage efficiency is the main advantage of erasure codes. In our KVRAID design, the extra storage overhead mainly comes from four parts. First, for the in-storage mapping table implementation, the mapping table itself needs to be stored on devices (with erasure coding protection). This metadata overhead per key is 31×1.5 B (25 bytes of user key and 8 bytes of physical key). Because the average size of the value is $\sim$2000B, the overall metadata overhead for mapping table is around $\sim$2%. Second, we pack extra metadata into the value (logical key and value size information) for garbage collection use. Third, the parity objects of the erasure codes. Since we manage records into multiple slabs, this can reduce the storage overhead for the parity objects. (suppose we only use single slab which needs to be 4000B, then all the parity objects need to be the maximum size of the value). Fourth, for batch writes, there will be invalid-alive records which are not yet cleaned by garbage collection. This contributes most to the storage overhead. However, this can be gradually reclaimed by GC during application idle time. This overhead can be tuned by appropriately adjusting the garbage collection aggressiveness.

Figure 2.13 shows the storage utilization results for different redundancy schemes. The storage utilization results (sum of all 6 devices) is collected directly from device through **nvme** status query which reflect the internal flash usage inside the device. For the batch writes experiment, we collect storage utilization before "compacting" all invalid-alive records to new parity groups. After "compaction", the storage utilization will be the same as after loading the data (100% insert).

Compared to RocksDB with software RAID6 (III), KVRAID with batch write and packing(VII) reduces storage overhead by 2.2x on average. This is mainly due to the software KV stores with block devices introduce more WAF [53] compared to KVSSD as discussed in Section 2.5.2.4. With in KVRAID, compared to mirroring with 2 replicas (II), the EC[6, 4] KVRAID (V-VII) configurations achieve $\sim$2x storage efficiency for 100% insert workloads. Since all incoming keys are new keys, batch writes can form full compacted parity groups to achieve the best storage efficiency. Batch writes with packing (VII) can further improve the storage efficiency by 6% compared to non-packing implementation (VI). For update/get workloads, due to the out-of-

place update effect for the batch writes, there will be invalid-alive records taking up space on the device. We will further analyze the performance versus storage efficiency trade-off under different GC configurations.

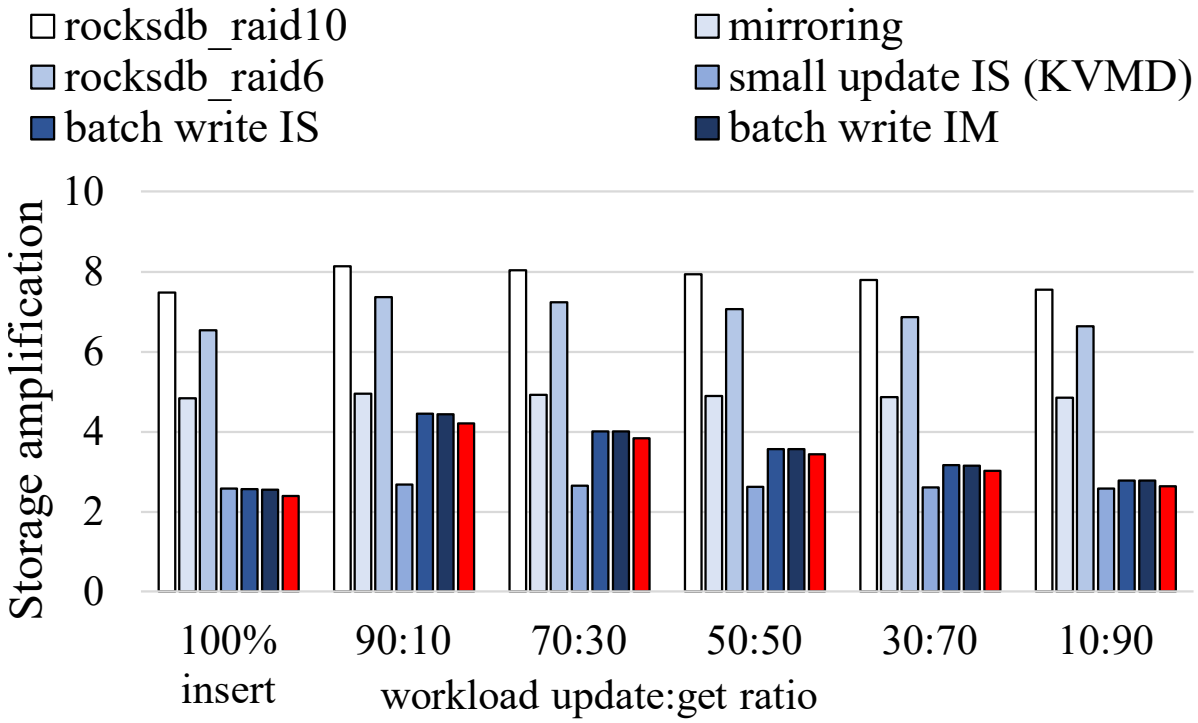


Figure 2.13: Overall storage (metadata included) overhead for different redundancy schemes (storage utilization is measured through nvme status query).

2.5.2.6 Garbage collection trade-off

As mentioned in sections above, garbage collection (GC) has a major impact on storage efficiency as well as performance. We conduct experiments to analyze the performance and storage efficiency trade-off under various GC levels. Figure 2.14 shows the performance (throughput) and storage efficiency (relative to mirroring) under 50%:50% update/get workloads. As we apply more aggressive GC, the performance decreases due to higher overheads of searching for candidate reclaim groups and issuing necessary I/Os to move valid records to new parity groups. This also increases WAF due to moving more valid records to new parity groups. In the meantime, storage

efficiency will go up since we clean up (delete) more invalid-alive records to save device space.

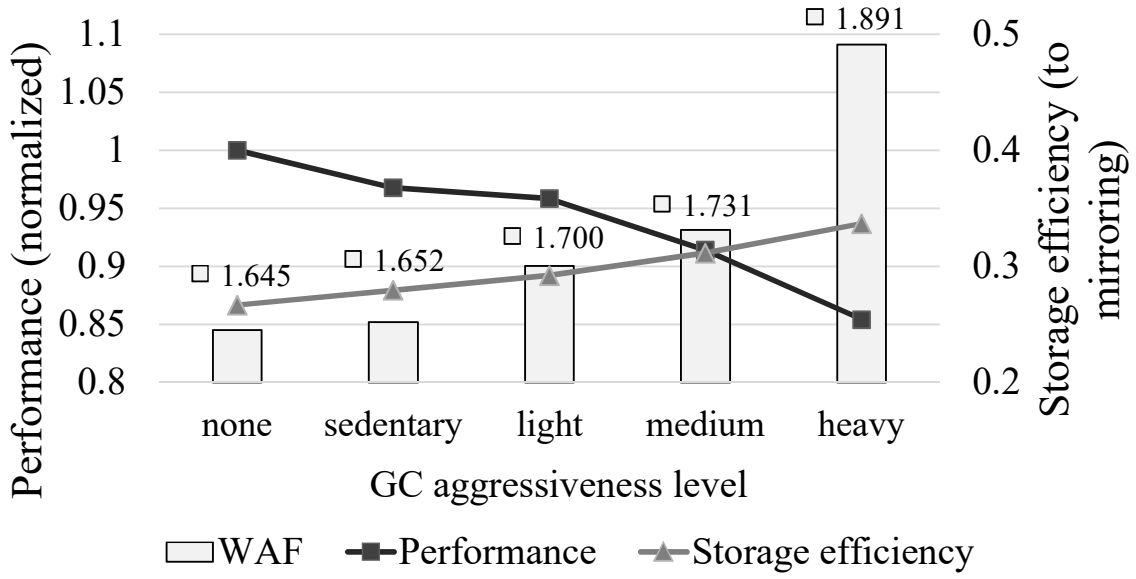


Figure 2.14: Performance (normalized to mirroring) and storage efficiency (against mirroring) trade-off under different GC levels.

2.5.2.7 Recovery efficiency

We also perform experiments on single device failure case to evaluate recovery efficiency for KVRAID. In the experiments, we disable one KVSSD and perform a 100% read workload (we assume software is aware of the device failure and requests hit on devices uniformly). For mirroring, if the primary copy of the requested record is on the failed device, the software will retrieve the replica from the secondary device. For KVRAID, if the requested record is on the failed device, it will retrieve other objects in the same parity group (in parallel through asynchronous I/O) from all other surviving devices and rebuild the requested record through erasure decoding. As shown in Figure 2.15. KVRAID only loses 12% and 9% of throughput performance compared to mirroring for in-storage metadata and in-memory metadata respectively. In-storage metadata requires $\sim$1.5% more get I/Os due to checking the in-storage metadata.

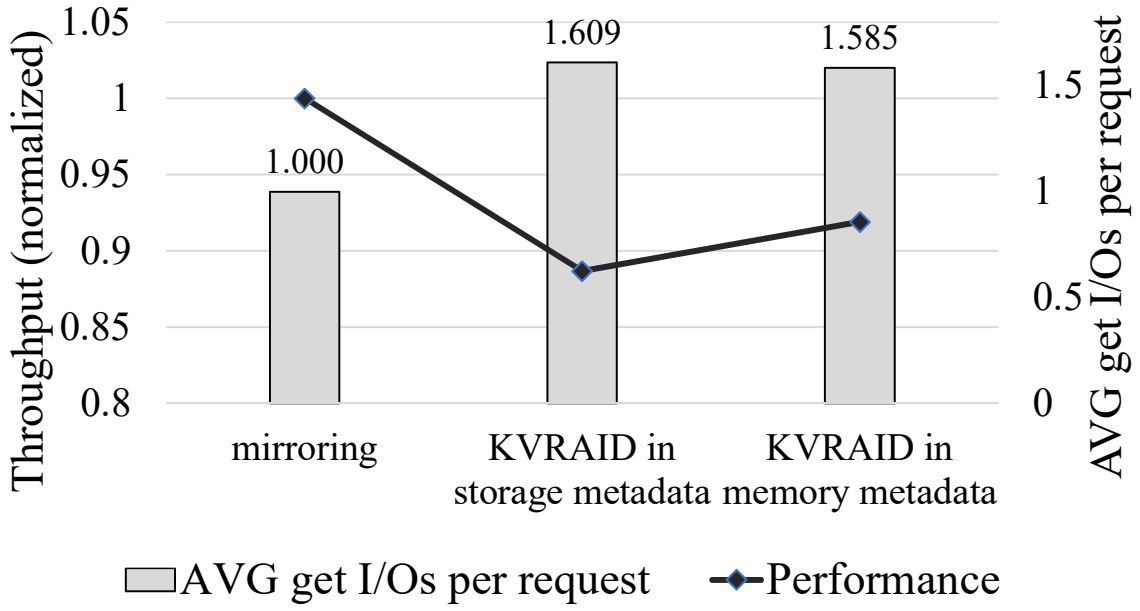


Figure 2.15: Recovery efficiency (normalized to mirroring) for single device failure.

2.6 Related Work

Replication [54] is widely used in large scale storage systems to achieve data redundancy for data stores [55, 56, 6, 9, 57]. While simple, replication has high storage overhead and write overhead.

RAID [46, 58] is a standard data redundancy scheme for storage systems typically for block interfaced devices. For block devices, it's easy to apply erasure codes with fixed block size. In our paper we tried to manage erasure codes on key-value interfaced devices which is significantly different from the traditional RAID studies.

Erasure codes have been widely adopted in storage systems to achieve reliability with space efficiency [44, 42, 41, 59, 60]. Current studies on erasure codes for storage systems focus on minimal regenerate code [61, 62] to optimize recovery bandwidth. Our KVRAID work is orthogonal to those earlier works, our design can employ different erasure codes without losing generality.

RAID optimization of SSD arrays are also discussed in previous works. Diff-RAID [63] creates an age differential in SSD arrays to wear out devices at different rates. EPLOG [64] redirects parity traffic to separate log devices to mitigate parity update overhead and improve performance and

endurance. SWAN [65] uses a log-structured idea to manage block SSD arrays to alleviate device level GC overhead. Kim, *et al.* [66] proposed Elastic Striping, similar to our batch writes, which reduces the parity writes across flash chips within a SSD. These works target on traditional block based interface.

Earlier works have studied applying erasure codes for in-memory KV stores [67, 68]. Cocytus [67] uses a hybrid replication and erasure coding scheme for in-memory KV stores. EC-cache [68] seeks to leverage erasure codes for in-memory cache in case of load imbalance and failures, to improve cache capacity. Unlike our work, in-memory KV stores do not consider device level write amplification. Our work mainly focuses on persistent key-value stores and how to build storage efficient RAID on KVSSDs to provide high performance and low write amplification.

KVSSDs were proposed to simplify the software stack for KV Stores [28]. Our work here propose to add a thin indirection layer on top of the devices to manage multiple devices as a parity group enabling more efficient protection from failures.

KVMD [38] introduces a hybrid data reliability manager with different reliability mechanisms for key value devices which lose efficiency on update-intensive workloads. While, the focus of our work lies on packing multiple records to apply erasure coding and maintain high performance and low WAF on update-intensive workloads. We focus our work on designing efficient techniques for managing write amplification with erasure coding-based protection of write endurance limited KVSSDs.

2.7 Summary

This paper proposed, implemented and evaluated a novel design, KVRAID, an erasure coding-based redundancy scheme for KV SSDs. KVRAID extends the idea of "slab allocator" to maintain erasure codes in multiple sizes to handle variable key-value object lengths. KVRAID employs a level of indirection from logical keys to physical keys that allows multiple objects to be packed into a single object on the device. By leveraging *batch writes* and *lazy deletion* with garbage collection, KVRAID can achieve high performance, low write amplification while maintaining the storage efficiency realized from erasure coding. Our measurements show that KVRAID outper-

forms existing software KV stack and replication schemes on performance, I/O amplification and write amplification, which can provide better energy efficiency and lifetime to KVSSD devices.

3. FAST, RESOURCE-EFFICIENT RANGE QUERIES FOR KVSSDS

Emerging key-value (KV) storage device is a promising technology for persistent key-value storage applications, featuring fast put/get operations. However, existing KV storage devices don't support direct range queries, which was proven by recent studies a critical performance factor, especially for analytic applications. In this section, we present KVRangeDB, an ordered log structure tree based key index that supports range queries on a hash-based KV-SSD. In addition, we propose to selectively pack smaller application records into a larger physical record on the device through a key translation mechanism.

3.1 Introduction

Existing block-oriented interfaces to Flash-based solid state storage require complex firmware, called a Flash Translation Layer (FTL), that allows logical block addresses (LBAs) to be mapped to multiple, arbitrary physical page locations as the LBA is written and updated [69, 70, 71]. Because locations may only be written from an erased state, and writes and erasures occur along different physical boundaries, FTL firmware must manage pages with combinations of live and stale data and independently decide when an update operation should trigger a process that recovers device capacity currently consumed by stale data. Thus, a single LBA update may trigger a large number of much slower internal device operations that read and write multiple pages of data within the device in order to construct a single large erase block before writing the new data. As larger and larger amounts of device capacity is consumed, the FTL must perform expensive space reclamation operations with greater frequency. For device users the result is multi-modal performance that is essentially unpredictable because the device characteristics are hidden behind the simple block interface.

Due to the unpredictable performance of block-oriented solid-state disks, Flash vendors have provided a variety of alternative interfaces to flash-based storage devices to users that wish to prevent or predict high-latency operations. Open Channel SSDs moved the majority of the FTL

into software allowing users to manage the physical placement of blocks and access the device's internal parallelism[72], more recent Zone Namespace (ZNS) devices provide an interface that allows users to leverage a block-oriented page append interface and indicate to the devices groups of blocks that can be erased efficiently[73, 74]. Most recently, the storage industry has standardized a Key-Value device interface [75, 73] that seeks to simplify device interface [28], simplify the mapping of popular key-value software interfaces to the device interface [30, 36, 29], and improve the performance predictability of solid-state storage devices [29].

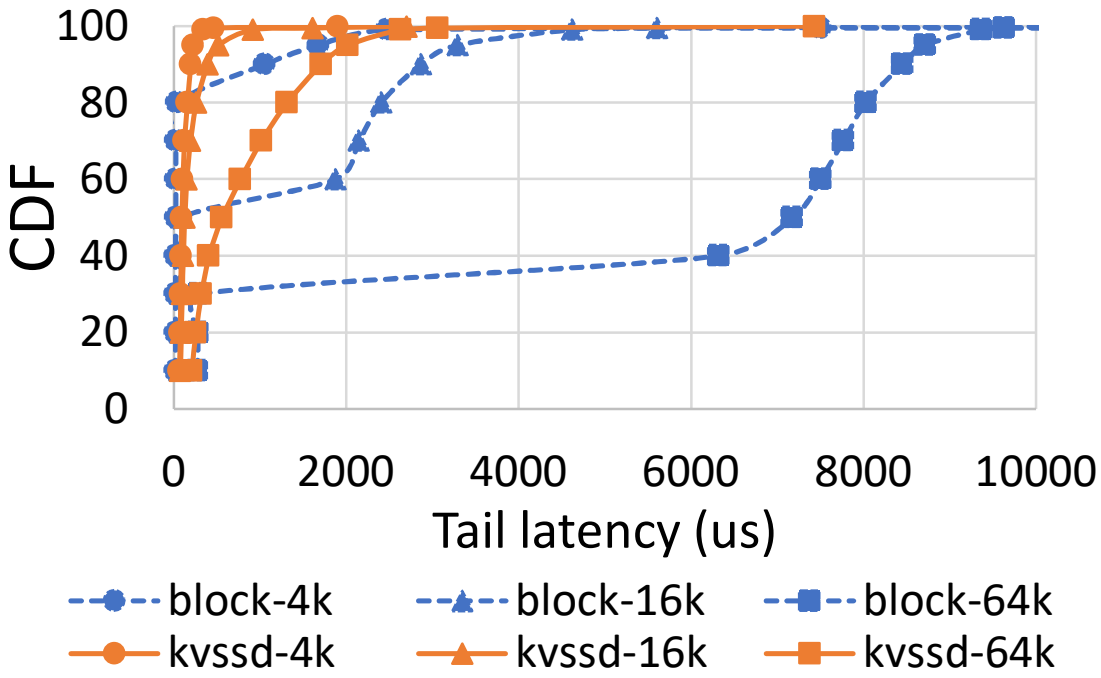


Figure 3.1: Device write I/O tail latency profile.

Figure 3.1 illustrates the device write tail latency comparison under different I/O sizes for block SSD and KVSSD (with the exact same hardware but different firmware). We observe much lower tail latency performance on KVSSD, which can further translate to more predictable application write performance. The KVSSD use a simple hashed based design [28] to map key to physical

block address for the entire record instead of block device which requires complex hybrid of page and block level mapping [71, 70] from logical block address (LBA) to physical block address (PBA). (The results are collected using fio under half device utilization for both device. The experimental setup details are described in Section 3.5.)

Modern key-value software accesses block-based storage devices using Log-Structured Merge (LSM) Trees. The LSM tree data structure provides efficient access for software updating and retrieving variable-sized data blocks by translating those into efficient block-oriented access operations [19, 17] . Key-value stores, such as RocksDB and WiscKey, have further refined the LSM data structure to specifically perform well on flash-based SSDs. Recent studies [76, 2, 77] show that range query or scan operations have become increasingly important for key-value stores applications, especially analytics applications [76, 78]. Also, some scientific workloads [79, 80] are seeking better storage and indexing solutions to accelerate simulation. B-Trees and LSM-Trees based key-value stores can efficiently support range queries as they manage data in an ordered fashion. However, maintaining ordered structure results in both high CPU utilization and lower get performance, as the KV-store application has to traverse the whole tree before finding a KV pair. Several researches highlighted the problem that LSM-Tree based key-value engines incur high external and internal write amplifications [25, 26, 35, 27]. The high write amplifications leads to performance degradation, high host CPU utilization and shortened device lifetime.

While it may appear that mapping this software to key-value devices (KV-SSDs) would be trivial, its important to understand the hardware interface and performance characteristics of existing KV-SSDs in order to efficiently map key-value software to key-value hardware. First, KV-SSDs provide hardware native interfaces for operations put, get, and delete; however, even though the standard describes iterator functionality, existing KV-SSDs do not provide range-based iteration or internal ordering that can be determined at the host. This may be the consideration of the internal write amplification [28], device performance limitation, cost and flexibility [28, 29]. Second, current KV-SSDs provide lower numbers of read operations per second (IOPs) and lower write throughput for large accesses compared to existing block-oriented SSDs. Finally, KV-SSDs gener-

ate many fewer internal write operations and thus have improved write latencies, lower operation latency variance, and generate less internal device wear. It is important to consider all of these characteristics when designing a key-value software layer that provides a full-featured key-value interface.

In this work we present KVRangeDB, a software implementation that leverages the KVSSD device characteristics to provide an efficient key-value store that includes support for range queries. KVRangeDB extends existing efforts such as RocksDB and WiscKey that optimize the LSM data structure for block-oriented solid-state disks. Similarly, KVRangeDB leverages a mixture of device native operations and LSM-based index data structures to provide fast key-value access and efficient range queries. KVRangeDB has been further tuned to specifically support the types of short range queries common in high-performance computing data center workloads.

The main contributions of this work are as follows:

- We propose a KV store design that employs an LSM tree index mapped to a key-value interface to support range queries efficiently on KV-SSDs.
- Using key index allows the device pack multiple small records and compact records, translate user keys to physical keys to reduce the number of keys to be managed in the device to maintain high *put* performance for the KV store.
- We employ multiple optimizations such as user hints, index/data cache separation and range filter, to adapt towards a wide spectrum of different range queries workloads such as open queries, closed queries, empty queries, etc.
- We implemented KVRangeDB and evaluate on real KV-SSD devices. We also presented the performance difference between traditional block SSDs and KV-SSDs for different real workloads.

The remainder of the section is organized as follows. Section 3.2 describes the background on how software key-value engines support range query and limitations that motivate us to implement

range query on the current KV-SSD technology. Section 3.3 demonstrates the design choices of KVRangeDB to support range queries efficiently on a KV-SSD. In Section 3.5, we evaluate KVRangeDB in a real system and compare with state-of-art software key-value store on block SSDs. Section 3.6 discusses some limitations of the work. Section 3.7 discusses the related work and Section 3.8 summarize the chapter.

3.2 Background and Motivation

3.2.1 Range query and emerging applications requirement

Range query or scan are common operations that retrieve all existing records with keys that are between given upper and lower boundaries. With the growth of data analytic workloads, recent studies [76, 2, 77] show the increase of popularity of range query operations in modern key-value store applications. For example, for database applications [76] which use KV store as backend storage, table scan will translate into a range query on the KV store. Time series database [81] also ask for range queries based on keys.

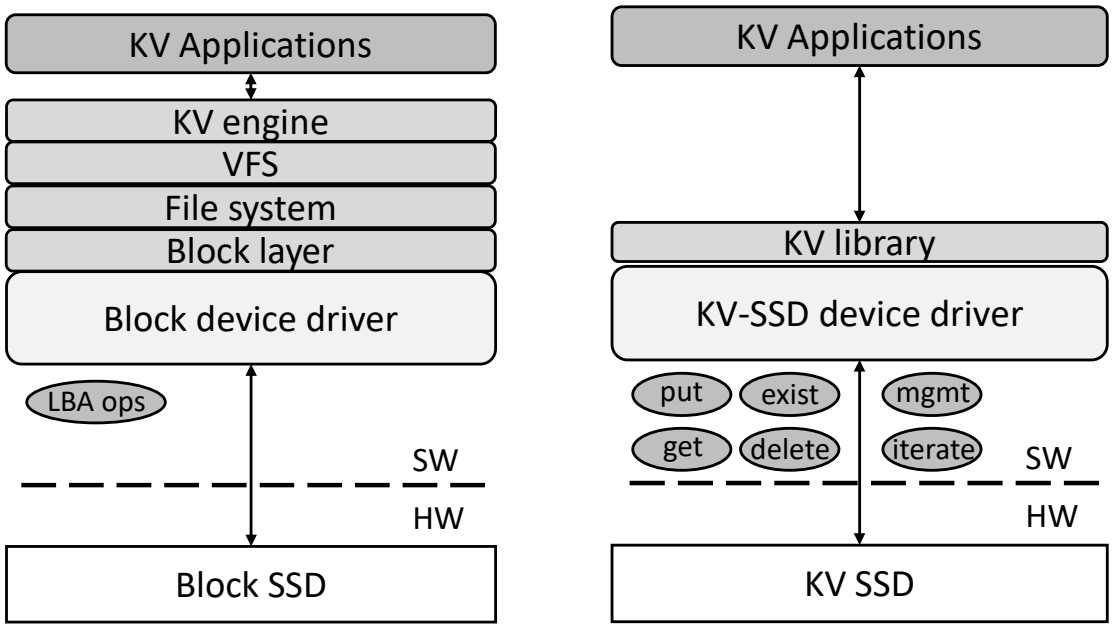
Modern key-value stores' API use an iterator interface for range queries. The iterator interface mainly contains *seek(key)*, *next()/prev()*, *key()*, *value()*, *valid()* calls. The user will first call *seek(key)* to locate the iterator to the start key. Then, it can call *next()/prev()* to move the iterator. At each iterator position, the key and value for the record can be retrieved through *key()* and *value()*. The *valid()* call is used to check whether iterator is valid or not.

3.2.2 Key-value SSD

The idea of key-value interface device has been proposed in both academia [36, 30] and industry [28]. Currently, Samsung provides KV-SSD products with a hash table implementation [28] targeting fast put/get performance and low write amplification.

Figure 3.2 illustrates the system stack for KV-SSD based systems. Compared to traditional software KV stores which involve complex key-value to file translations, file system, block layer, device driver, it provides a much thinner layer of software stack including a device driver and user space KV library. It contains basic device management, put, get, exist, delete and iterator

interfaces. The iterator interface enables traversing a group of keys (with 4 bytes prefix bitmask), but with no key order.



(a) Software KV system stack

(b) KV-SSD system stack

Figure 3.2: Comparison between (a) traditional software KV system stack with (b) KV-SSD system stack.

3.2.3 Limitations of Key-value SSD

Due to the limitation of computational resources on the storage devices and the general requirements for fast put/get operations, Key-value storage devices may not store the KV records based on key order. Figure 3.3 demonstrates the layout of hash based Key-value SSD [28]. The user key will be hashed and stored in the local hash table and then merged into a global hash table in the NAND flash array. Hence, there is no key order in the device. In order to perform a range query natively from the device, it needs to retrieve all the keys stored on the device, then process them in memory to find the target keys in the given range, to obtain acceptable performance.

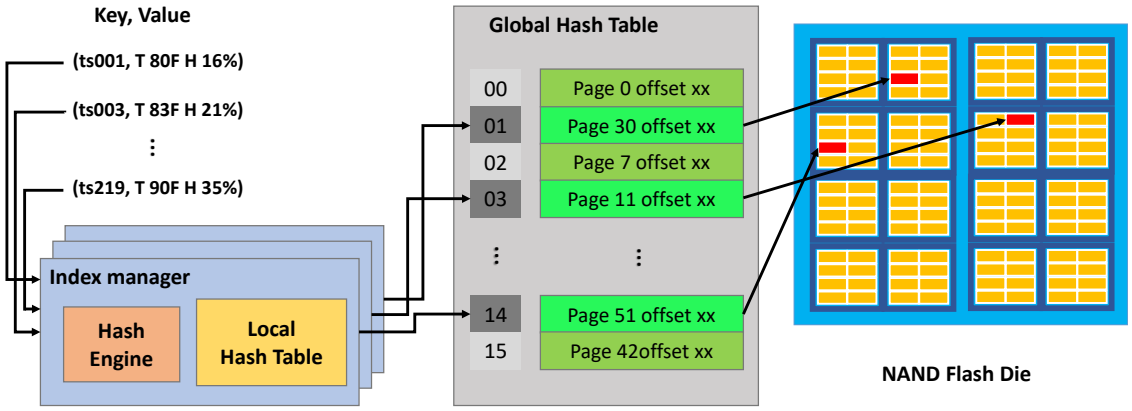


Figure 3.3: Hash based Key-value SSD layouts.

We implemented a range query using the device iterator interface which retrieves all the keys stored on the device and used a priority queue to store the keys from the seeking key up to scan length. We suppose using small memory budget which cannot keep all the keys in memory for carrying out the range query. (Consider a dataset with 1 Billion 32 bytes keys, the size of all the keys in the dataset will exceed 32GB). We conduct experiments to perform range queries on a KV-SSD for data set with 0.1, 0.5, 1 and 10 million records. Figure 3.4 shows the latency breakdown for the range queries. Nearly 70% of the latency cost is in retrieving keys from device (using the device native iterator interface) and $\sim$30% of the latency is in processing the keys in-memory for finding the keys in the queried range. The average query latency scales to tens of seconds for 10 million records which is unacceptable for practical applications.

Another limitation of the KVSSDs is the performance slowdown with the number of records stored on the device. Software KV stores use the Log Structure Merge (LSM) [19, 17] Trees as the storage structure allowing them to scale to larger number of records. KVSSDs choose a hashed based design for storing records on the device. We conducted a performance test to write 1 Billion records with 100B, 1000B, 2000B value sizes respectively into the device and logged the instant throughput every 10 seconds. The experimental setup details are described in Section 3.5. We observe similar put throughput performance scale down as the number of records stored increases no matter how large the value size is as shown in Figure 1.3. This indicates that the performance

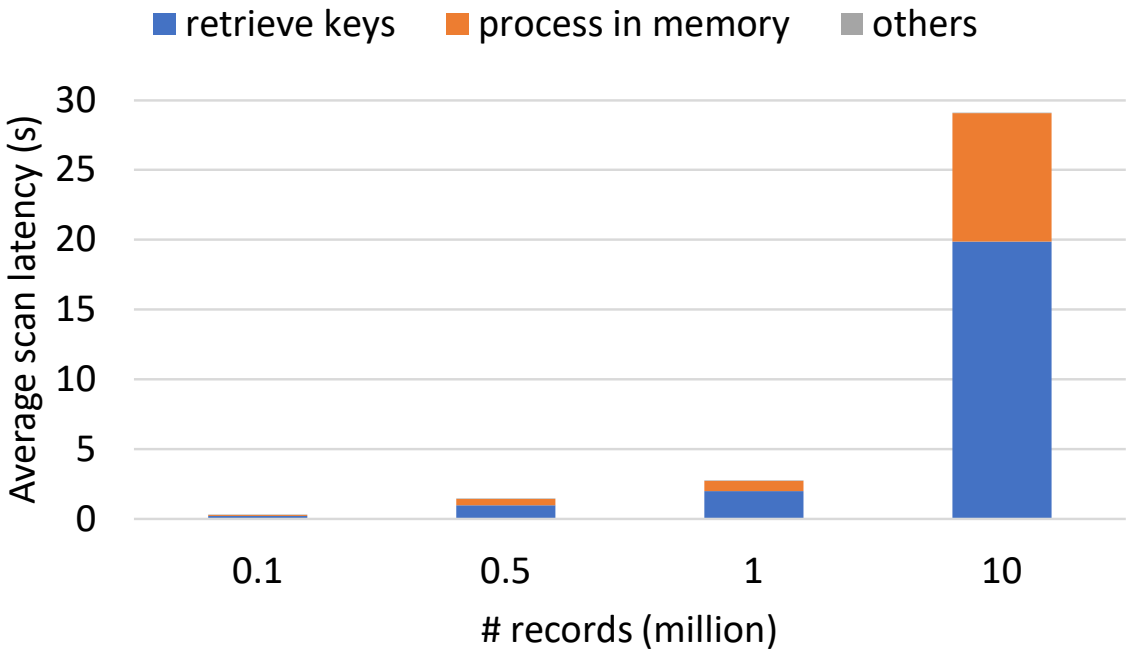


Figure 3.4: Latency breakdown of a range query using native KV-SSD interface.

of the device will drop significantly with larger number of records, irrespective of the value size. The performance deteriorates even with small records, even though there is plenty of capacity in the device. It is observed that the performance is dependent on the number of records stored on the device, irrespective of the size of the records.

These observations point to the need for providing support for efficient range queries on the KV-SSDs and the need for mitigating the performance slowdown with larger number of records on the device. We propose a solution that efficiently supports range queries on the device while improving the scalability of KV-SSDs performance with larger number of records.

We expect the proposed structures to be helpful for the design of future generation KV-SSDs and potentially be applied to the KV-SSD software/hardware stack.

3.3 KVRangeDB

KVRangeDB is designed to support efficient range queries on hash based KV storage devices while maintaining the native put/get performance benefits from the device. The main idea is to

manage an ordered key index separately from the data. For a range query, we will first check the key index and find the target keys in the queried range and then retrieve the values from the device. The idea seems straightforward, however, there are many problems to consider. First, how to implement efficient index structure on a key-value interface rather than on top of file system interface? Second, when performing a range query, key access and value access involve separate data paths, how can we amortize the latency for data access after we get the target keys in the queried range? Third, for smaller value size records, KV storage device cannot saturate internal flash bandwidth natively from the device interface. In addition, how should we scale performance with larger number of records in the database?

With these fundamental questions in mind, we will describe our design choices in the following sections.

3.3.1 Basic APIs

KVRangeDB provides a key-value semantics with range query support. We use an iterator interface to perform range query or scan operations. We define the following APIs for our KVRangeDB: (the user hint APIs will be discussed in Section 3.3.5)

- ***put(k, v):*** Put new key-value pairs.
- ***get(k, v):*** Retrieve value from key.
- ***delete(k):*** Delete key-value pairs.
- ***iterator :*** Iterator for range query.
  - ***seek(k):*** Moves the iterator to the first key-value pair which key is greater than or equal to the seek key.
  - ***next():*** Move the iterator to the next key-value pair.
  - ***valid():*** Whether iterator is valid.
  - ***key():*** Return the key of the current iterator.

– ***value():*** Return the value of the current iterator.

– ***hint.upper_key:*** Specify the user hint for end key from the seeked key.

– ***hint.scan_length:*** Specify the user hint for the scan length.

3.3.2 Packing smaller records

In the rest of the section, we use logical keys and user keys interchangeably as the application keys. We define physical/device keys as the actual key written to the device with the KVSSD KV interface. For smaller size records, packing multiple values into a single physical record can yield better write throughput and mitigate the performance scale down for large number of keys. The logical keys to physical keys mapping can serve as a key index to fulfill range query capability over logical keys, killing two birds with one stone.

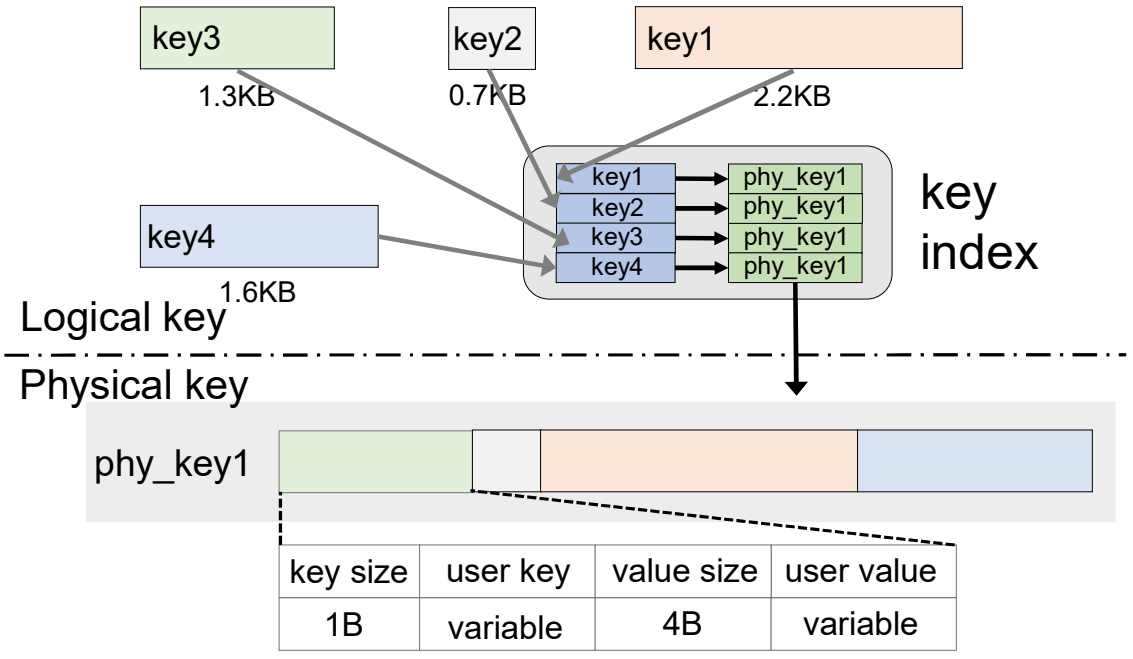


Figure 3.5: Packing smaller records and translate user keys.

Figure 3.5 illustrates how smaller KV records are packed into a large physical record. Multiple logical records can be packed to a singular large physical record to yield higher write throughput

and reduce the number of physical keys managed in the device. The key index keeps the logical keys to physical keys mapping for retrieving records by the logical keys, which requires a linear scan on the physical record to extract the user record). In order to support range query on the logical key. We use LSM tree to maintain the logical key to physical key translation. The main reason to choose LSM tree as the data structure for logical to physical key mapping instead of traditional B/B+ tree is to achieve higher write performance [19, 17, 18]. Otherwise, it may contradict the purpose of packing.

When performing point queries on the packed records, it will first consult the LSM tree key index to find the physical key and then retrieve the value from the packed physical record. Similarly, for range queries, it will traverse the LSM tree key index to find the physical keys mapped to the target user key range. Then, the corresponding values will be retrieved and extracted from the packed records.

3.3.3 Building key index for range query

This section will describe in detail how we use the key index to support range queries on a key-value storage device. As we mentioned in Section 3.3.2, we choose an LSM tree based key index for logical key to physical key mapping when we pack smaller records, to achieve high write/put throughput performance. For larger records, on the other hand, we simply leave the logical keys in the LSM tree key index for sake of logical key order for range query and use the logical key as the device key directly without the need of logical to physical keys translation. This is the core difference compared to Wisckey [25] which also employs the idea of separating the key and values. For the Wisckey design on a conventional block device, the key index keeps the mapping between the logical keys of the KV records and pointers (file offset and value size) for the corresponded values in the continues value log. For both point and range query, Wisckey needs to consult the key index in order to retrieve the values from the value log which requires more than one I/O. However, for KVRangeDB, we can directly use the logical key to retrieve the value from the device with exactly one I/O for large unpacked records. For example, as shown in Figure 3.6, *lkey1, lkey7, lkey12*, etc. are unpacked records that can be retrieved directly from device through

the logical keys. *lkey3, lkey52* are packed into a physical record (physical key pkey12) and need to go through key translation to retrieve the value of the records.

To balance write and range query performance, we carefully design the LSM tree structure. The core in storage data structure for LSM tree is SSTable [17] which contains multiple data blocks which contains sorted KV pairs and index block which contains key ranges for each data block to accelerate key lookup per SSTable. There is another manifest structure to index each SSTable. For software LSM tree based KV engine on block device, the SSTable is written as file to leverage to file semantics for better write/read performance. However, on key-value semantic device, if we simply store each SSTable as a single record, the LSM tree lookup performance will be heavily degraded since we need to read the whole SSTable every time even if we only need to read 1 KV in the data block.

In our LSM tree index design, we use separate keys to store each data block and the index block. (Here block is not fixed size block in the block device, it can be any size) Figure 3.6 illustrates the overall architecture for KVRangeDB. Similar to levelDB and rocksDB, the LSM tree index contains memtable, multiple sorted SStables based on logical keys, log and manifest. The log and manifest use a single KV record. For SStables, we use separated KV pairs to store the data block and index block. The data block keys use SStable number plus offset. There is a single KV pair for index block using SStable number as key which contains the information of key range and offset for each data block for data block access.

For cases that there is hybrid key translations which partial of the small records got packed with logical key to physical key translation and the rest of the records use logical key as the physical/device key. In the context of *get* operations processing, if requires checking the key index to make sure whether the queried key is translated or not. In our design, we leverage a small bloom filter [82] to reduce the overhead of key index checking when the keys are not translated and can be directed retrieved from the device with the logical/user keys. Assume there is only a small portion of un-translated *hot keys* in the dataset, we store those keys in bloom filter with a small memory footprint. As illustrated in Figure 3.7, when we process the *get* operations, we'll consult the bloom

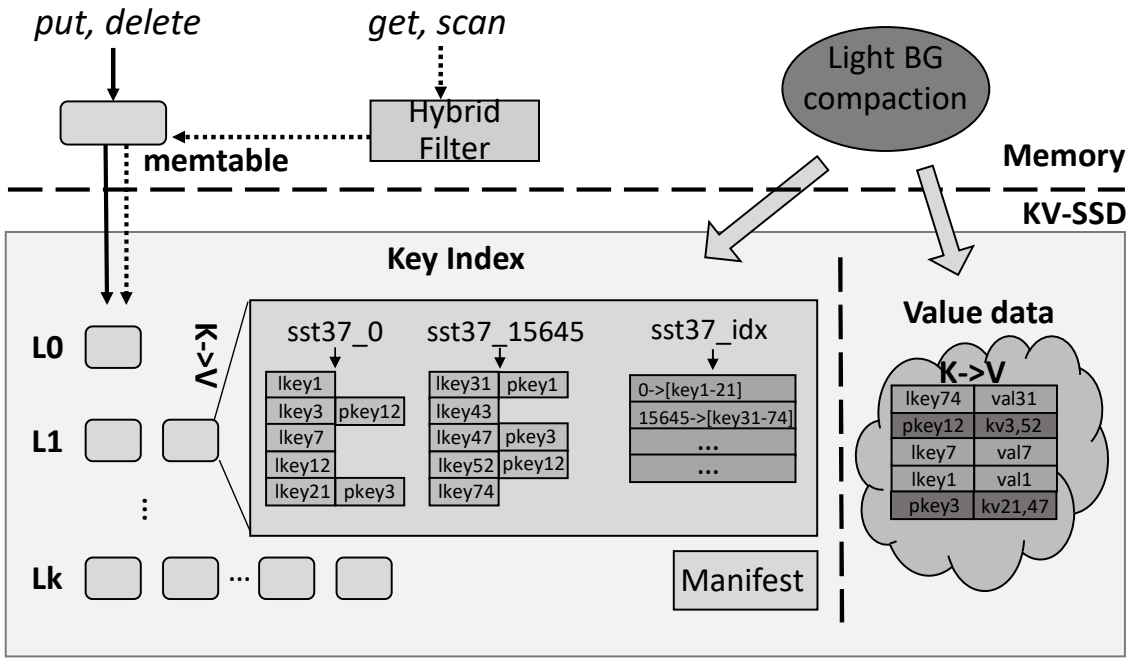


Figure 3.6: Overall architecture for KVRangeDB design.

filter. If the filter returns negative (dashed arrow) which means the queried key is definitely not translated. We directly retrieve the value from the device with the logical key. Otherwise (solid arrow), we'll consult the key index to find the physical key for the value.

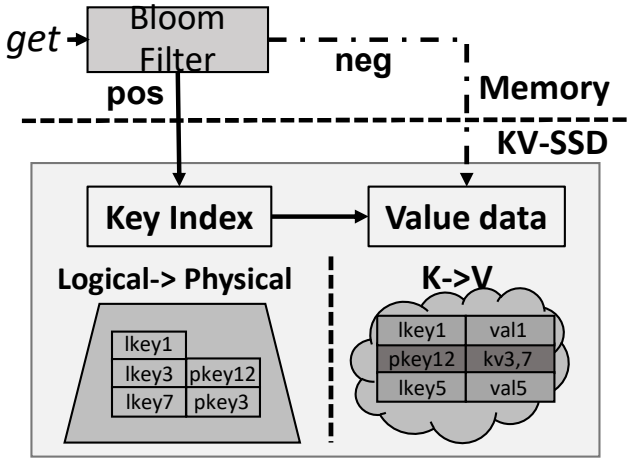


Figure 3.7: Bypassing index checking for hybrid key translations.

3.3.4 Range filter for empty queries

Given the size of key lengths and typical number of records in a store, most stores don't have more than a small fraction of key space occupied. As a result queries may result in empty/negative replies or that we need an efficient mechanism for deciding that keys don't exist. In KVRangeDB, we design filters to filter out empty/negative queries for both point and range queries. Filters are compact/compressed structure that can be completely stayed in memory. For a typical data store with 1 billion keys, the key index size may be tens of gigabytes which exceed the common machine memory size and needs to be designed as in-storage data structure. However, typical filters only require very small memory footprint (1-2 gigabytes per billion keys) and can be fit in memory.

Bloom filters are efficient data structures which can distinguish the membership of an element in a set which helps point query ("Is key 7 in the store?"). LSM-tree based KV stores [17, 18] have already adopted bloom filters for reducing unnecessary disk I/Os when the queried keys are not exist for point queries. However, simple bloom filter are not efficient to handle range queries ("Are keys from 3 to 100 exist in the store?"). We can query individual keys from bloom filters multiple times (from key 3 to key 100) to determine whether the queried range exist. However, such method suffers from high computation cost and high false positive rate.

Recently, more advanced filters were proposed [2, 83] with similar purpose for range queries especially for those short range queries with high probability of being empty.

Unlike the prior works [2, 83] which designed block based range filters target on LSM-tree based KV stores. We proposed a lightweight unified in-memory range filter for accelerating both empty queries point and range queries. As opposed to storing filters for each sorted-run, we don't store any filter data in storage, but build the filter on the fly when opening the database (we can also cache the filter data in a block or KV devices). There are two main reasons for such design. First, unlike LSM-tree based KV-stores which need to scan the entire database to retrieve all keys in the database, our KVRangeDB separates sorted key index from value store and can retrieve the entire keys efficiently. Second, building the range filters on the fly is more flexible to accommodate the fast shifting of workloads by altering the filter designs. For example, workloads that are dominated

with empty point queries may only need a simple bloom filter with less memory cost. Workloads that rarely encounter empty queries may simple discard the filter.

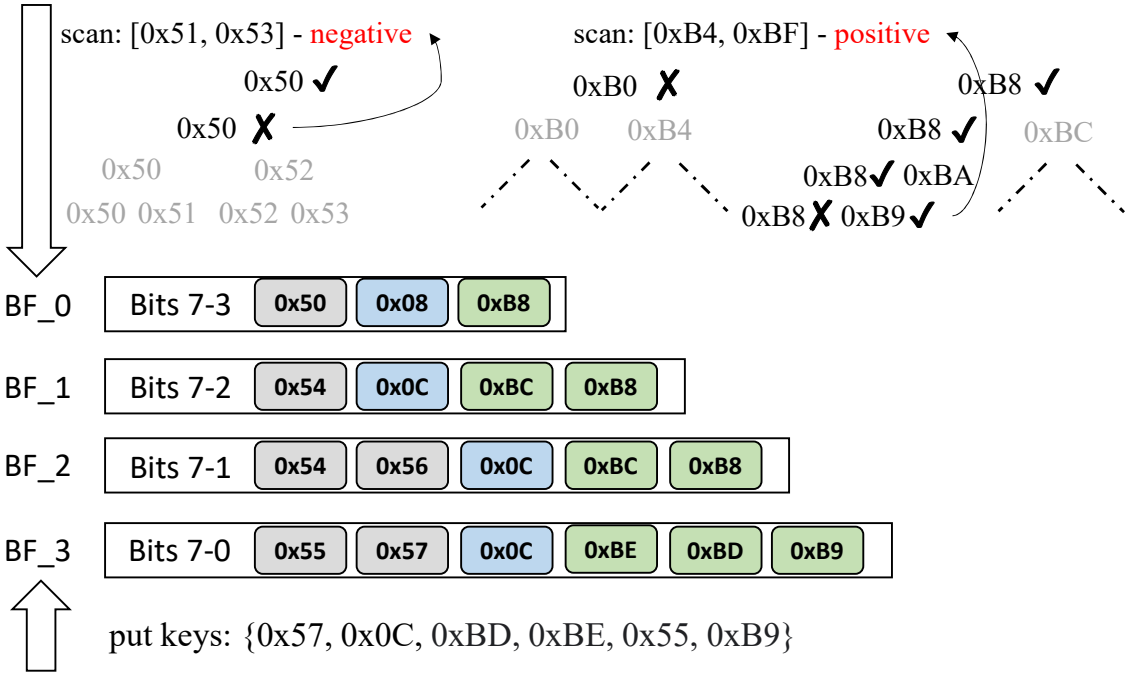


Figure 3.8: Hierarchical bloom filter for range queries filtering.

Figure 3.8 illustrates how our range filter works. In our design, we extend the idea of prefix bloom filter [18] and use multiple layer of prefix bloom filter, each with different size of prefix to enable efficient range filtering. When building the filter, each key in the database stores various size prefixes into each level of bloom filters (the bottom level stores the full key bloom filter which also works for point queries). For range queries that consult the filter, it will break down to multiple prefix sets according to the top-level bloom filter prefix length. For each prefix set which range is covered by the top-level prefix length, it can then recursively probe the lower level bloom filters to determine if there are potential keys in the checked range. As long as there is one possible key exist in the queried range, the filter will return positive and requires to check the key index in storage for the query. On the other hands, if the filter return negative, which means the queried range is

definite empty, we can directly return and save the I/O cost for checking the key index.

For the memory allocation to the Hierarchical bloom filter design, which means how much memory footprint allocate to each level of bloom filter. Intuitively, higher level filters may contain less distinct keys due to shorter prefix length and may require less memory cost for the filter. In our design, we use a simple strategy to allocate memory footprint as follows, which proves to be working well.

$$M_i = M\frac{i+1}{\sum_{n=1}^{N} n}, i = 0, 1...N-1$$

Where N is the number of levels for the Hierarchical bloom filter. M is the total memory budget for the range filter. M_i is the memory budget for the ith level filter (i start from 0 to N).

3.3.5 User hints for efficient queries

The key index brings up two problems hindering efficient query process. First, unlike LSM based software KV stores [17, 18] which pack key and values together, the range query in our design will need to consult the index first and get the target keys in the queried range. Then, we need to separately issue I/Os to retrieve the values associated with the target keys if the user also asks for values. This requires additional I/Os to finish the range query. Second, the records packing introduces key translation from logical keys to physical keys, the value may not be directly retrieved with the logical key for simple point queries. In such a case, the point query performance will be impacted by the additional I/Os for index look up.

We propose two approaches to resolve these issues. First approach is to leverage user hints for prefetching the values to overlap the value retrieval latency. We implemented two additional read options for range query, i.e. *scan length* and *upper bound key*. Since user may have prior knowledge of the queries, (For example in table scan, what is the approximate number of entries in the table. Or in a query for events between two timestamps, what is the end timestamp, etc.), by applying those hints, we can prefetch the values in advance to hide the latency for accessing the values separately. Besides user domain knowledge, proper profiling can be also used to help

extract hints information to better leverage our hint interface. We also design prefetch throttle mechanism to prevent too many in-flight prefetch requests that may increase the device queueing time and affect the other demand I/O requests.

Similarly, for get operations, user can provide hints on whether the record might be packed according to the user prior knowledge of the record size or hotness to guide bypassing the unnecessary index lookup.

3.3.6 Other optimizations

3.3.6.1 Cold records compaction

Since the device write performance is heavily impacted by the number of records written to the device, it will be beneficial to reduce the number of physical records written to the device, through packing. With our key index design, we are able to conduct key translation and pack multiple records to a single physical record to reduce the number of records written to the device. This allows us to reduce the performance slowdown from larger number of records. In our design, we embedded some metadata in the values to indicate the hotness of a record. Since we are doing *out-of-place* update for update/delete operations, garbage collection is required. During the garbage collection, we will examine the hotness of the records and pack cold records into a larger record to reduce the overall number of physical records written to the device. We consider the hotness of the records during the garbage collection to pack multiple cold records into a larger record. This benefits performance through a reduction of the number of records on the device and also makes future garbage collection more efficient by separating hot and cold records.

3.3.6.2 Separating the index and value cache

Separating the keys and values was first proposed by Wisckey [25] to reduce the write amplification issue of the LSM-tree based KV stores. In our design, we kind of follow the same philosophy of keys and values separation by storing a separate key index to support range query. Besides the reduction of write amplification, we are also able to separate the cache for index and value.

3.4 Crash recovery

For failure recovery, here we only consider host side failures such as a system crash or a software failure since we didn't consider device redundancy in this work. The LSM Tree index can be made failure resistant by applying logging for every index write. However, if the logging is disabled due to performance reasons, we are still above to recover the uncommitted *volatile* memtable and keep the consistency of the LSM index. This is done by retrieving all the keys from the device and rebuilding the LSM tree index. For example, if there are objects (without key translation) written to device but not reflected in the LSM tree index due to crash, during the recovery process, we will retrieve all the keys on the device and compare them with all the keys stored in the LSM tree index and add the lost keys in the index to make sure the consistency of the index.

For objects with key translation, since we embedded the user keys into the packed objects, we are able to rebuild the latest key translation mapping by the monotonically increasing sequence number OF THE PHYSICAL KEYS. As for the deleted objects with key translation, we keep a small separate log for the deleted keys on the device in case we loose the deletion information in the LSM tree which cannot be recovered by the monotonically increasing sequence number.

3.5 Evaluation

This section presents the experiment results of YCSB benchmark [50] and two other real-world KV applications that rely on range queries. We compare KVRangeDB against the state-of-art software KV-store on block device Wisckey [25] and state-of-art industry counterpart RocksDB [18] which ports to KVSSD. Then we analyze how each optimization technique contributes to the overall performance improvement, and their impacts on different types of KV operations. Based on these observations, we discuss the potential use cases of KVRangeDB.

3.5.1 Methodology

3.5.1.1 Experiments setup:

Table 3.1 lists the detailed hardware information. Block SSD and KV-SSD use the same SSD hardware device except that the firmware are different.

Table 3.1: Hardware Specification

Component	Description
CPU	Intel Silver 4216 @ 2.1GHz, 16 cores
Memory	96GB DDR4 @ 2133MHz
SSD	PM983 3.84TB
KV-SSD	PM983 3.84TB
Memory	128GB DDR4
OS	Linux version 4.15

Since the Wisckey source code is not disclosed to public. We implemented Wisckey according to the paper for our evaluation. Instead of using LevelDB to store the user key to <log offset, value size> mapping in the original paper, we use RocksDB. In order to make comparison under same memory budget and exclude the page cache effect. We use direct I/O mode for RocksDB in the Wisckey implementation. The designs under test in the experiments are listed as below:

- **Wisckey:** Wisckey implementation on conventional block SSD. The value are packed in a contiguous log file with 1MB log buffer. The key to log offset mapping for each record is stored in RocksDB.
- **RocksKV:** RocksDB implementation port to KVSSD. Using key-value interface instead of filesystem interface to store the SSTable files and metadata files. For SSTable files, we store each data blocks with separate record using combine of SSTable file number and block offset as the key. Manifest file are stored as a monolithic record.
- **KVRangeDB:** KVRangeDB implementation described in this section.
- **KVR-prefetch:** KVRangeDB optimized with value prefetch for scans.
- **KVR-2level:** KVRangeDB optimized with 2 level hybrid point and range filters.

3.5.1.2 Workloads:

We conducted three categories of experiments to evaluate our KVRangeDB design and compare to the state-of-art software based KV-store Wisckey on block SSD and popular industry KV-

store RocksDB implementation ported to KVSSD under different workloads. First, we measured the range query performance with comprehensive micro-benchmarks including scan operations of various length, with/without retrieving value, as well as simple *put*, *get*, *seek* operations under Yahoo! Cloud Serving Benchmark [50]. Quantitative description for each query workloads is explained in the following sections. Second, we run several file system applications under TABLEFS [84] which utilize KV-store as the metadata management engine. This application generates the directory tree, find, list file/directory, list stats which are all composed of mixed *put*, *get*, *scan* queries. Third, we executed a time-series application from Surf filter [2] to evaluate the performance for range queries that mix with empty and non-empty queries.

Micro-benchmarks: We use Yahoo! Cloud Serving Benchmark (YCSB) [50], a popular benchmark for key-value store and cloud service providers as platform to evaluate a variety of micro-benchmarks. First case is a dataset with 250 Million records with 16B key size and 4000B value size which is approximately 1 TB in capacity. We don't pack records for this case, i.e, the index just keeps the sorted user keys and a get operations can be fulfilled without an index lookup. Second case employs 1000B value records [85] and 1 Billion records, resulting in approximately 1 TB size. For this case, we packed 4 records in a physical record written to device. The index contains the key translation from user keys to physical keys. We evaluate various singular KV operations and range queries workloads described as follows (All workloads are generate under uniform distribution).

I **Get:** Also known as point query. Retrieve a KV record with a given key.

II **Scan keys :** Find the next closest N keys. This is composed of a seek() call and N next() calls based on scan size and retrieving only key without value.

III **Scan keys&values:** Find the next closest N records with keys larger or equal to the target start key. This is composed of a seek() call and several next() calls and retrieving key and value for each next() call.

IV **Empty queries:** Point queries and range queries which return empty/non-exist results. For

range queries, it specifies lower andupper bound key and there is no key in the dataset exist in the key range.

We use 16 threads for load and query phase for all three implementations.

TABLEFS: TABLEFS [84, 86] is a fast and efficient filesystem using KV store as metadata management engine. Unlike traditional filesystem (like ext4 or BTRFS) which manage directories and files in a tree structure, TABLEFS use KV-store to manage directories and files. Each file/directory corresponds to a record in the KV-store which key is composed of parent path inode number and file/directory name as suffix and value is the actual inode data (4KB in size). The directory path lookup is performed recursively starting from the root directory and readdir (a filesystem API that read all files/sub-directories from a directory) can be done with range scan throughput the directory inode number).

We use real filesystem trace from Los Alamos National Lab and generate various filesystem workloads to evaluate the performance of our KVRangeDB as follows. In all following experients, we create 256MB directory cache for TABLEFS to reduce I/O overhead for path resolve. For KV-store, we compare using no cache and with 256MB cache (block cache for Wisckey and KVRangeDB's LSM-tree index and block cache for RocksKV).

I **Generate directory tree:** Generate a huge directory tree using actual filesystem trace from LANL (Los Alamos National Lab) which contains around 5 million directories and 120 million files in parallel (16 worker threads). Then conduct two rounds of updates. Each round of updates includes removal of 25 million of the files, updating 25 million inode contents (chmod/utime) and re-insert 25 million files. The total raw database size finally is around 450GB (only for metadata for the entire filesystem, not including actual file data).

II **Parallel ls -l:** Randomly pick 500 thousands directories from the filesystem and list the files/sub-directories in parallel with detailed stats (16 worker threads per core).

III **Parallel ls:** Randomly pick 500 thousands directories from the filesystem and only list files/sub-directories name in parallel (16 worker threads per core).

IV **Parallel find:** Randomly pick 75 directories from upper 10 levels and traverse each directory in breadth-first fashion in parallel (16 worker threads per core).

V **Parallel lstat:** Randomly pick 5 million files/directories from the filesystem and list the file/directory stats in parallel (16 worker threads per core).

Time-series workloads: Time-series database is an popular data store for various emerging applications in IoT, cloud computing, etc. Popular time-series databases such as InfluxDB [87], LittleTable [88] and kairosdb [9, 89] use data structures similar like LSM-tree. We use the time-series workloads which generated hundreds of millions events from distributed sensors in SurF [2] to evaluate the KVRangeDB performance on time-series workloads.

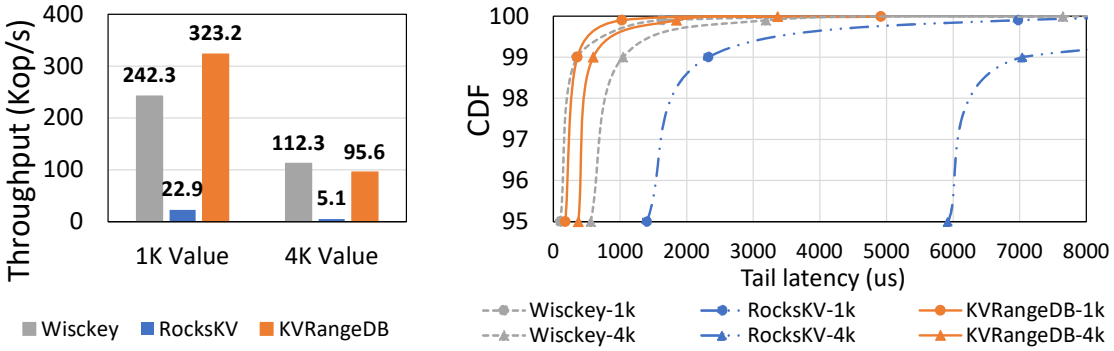
The time-series application [90] simulated 10K sensors recording time-series events. The key for each record is 16 bytes comprised of a 64-bit timestamp. The associated value for each event record is 1 KB long. The occurrence of each event detected by each sensor follows a Poisson distribution with an expected frequency of one every 0.2 seconds. Each sensor operates for 10K seconds and records $\sim$500 million events in total writing to the database with 16 threads. The total size of the raw records is approximately 500 GB. For the queries workloads, the application has three types of query. First, *point query* (or simple *get*) find the associated event record for a given timestamp and sensor ID if exist. Second, *open query* find the next N events from a given starting timestamp. Third, *closed query* find all the events within an given time period.

3.5.2 Results for YCSB

First both experiments (with 1000B and 4000B value size), we first load all the data on the device (the index is written with the data). Then, we run different types of query workloads and examine the performance.

3.5.2.1 Write performance

Figure 3.9 (a) demonstrates the throughput performance of loading data onto the device. For smaller records, packing can be useful in improving the overall write throughput and reducing the number of keys managed by the device as we discussed in Section 3.2 and Section 3.3.2.



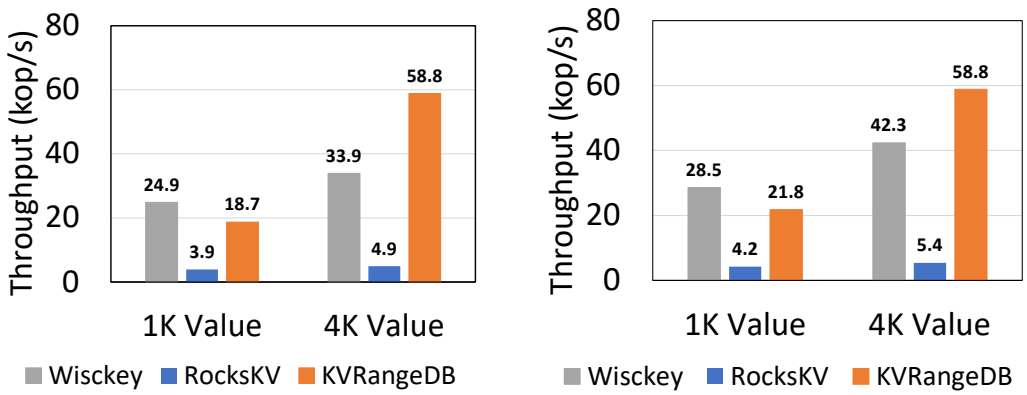
(a) Throughput

(b) Tail latency

Figure 3.9: YCSB write performance (16 threads).

KVRangeDB load throughput outperforms RocksKV (require heavy compaction since it packs keys and values together) by 14x and Wisckey by 1.3x. Packing more records into a physical record yields higher write throughput, thus it enhances the data loading efficiency. KVRangeDB is beneficial to write heavy use cases which contain lots of small records. For 4000B value size, KVRangeDB can achieve 18.8x better performance compared to RocksKV. KVRangeDB performs slightly worse compared to Wisckey (∼15% worse) because Wisckey leverage large sequential I/O for write. However, Wisckey's implementation suffers on remove and update (requires host side garbage collection) compared to KVRangeDB which can directly remove and update records from device through the user key. We evaluate it in the filesystem workloads in section 3.5.3.

Figure 3.9 (b) shows the tail latency for loading data from application level. For 1000B value size case, KVRangeDB outperforms RocksKV by 6.8x and Wisckey by 1.6x for 99.9th tail latency. For 4000B value size, KVRangeDB outperforms RocksKV by 9.8x and Wisckey by 1.7x for 99.9th tail latency. KVRangeDB achieves much better tail latency for two reasons. First, it only maintains a small LSM-Tree key index which introduce much less compaction compared to traditional LSM-Tree KV-store implementation which pack keys and values together. Second, KVRangeDB leverages device's KV interface which doesn't need to do large synchronized write when log buffer is full as Wisckey.



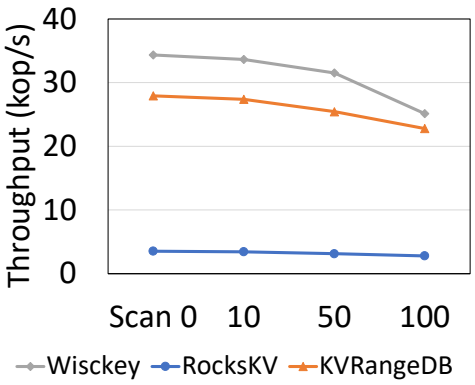
(a) Get throughput with no cache (16 threads) (b) Get throughput with 1GB cache (16 threads)

Figure 3.10: YCSB Get performance

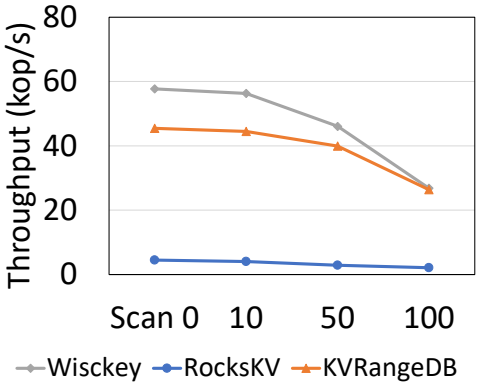
3.5.2.2 Point query

For RocksKV, a *get* operations requires to examine several sorted-runs in each level in the LSM-tree to finally retrieve the records which introduce multiple I/Os. Wiskey needs to look up the LSM-Tree to find the log offset of a record based on user key and then retrieve the value from the log. However, for KVRangeDB without packing, the *get* request can be fulfilled by a single I/O using the user key through the KV interface provided by the device. For KVRangeDB with packing, similar to Wiscky, it only requires to traverse a small LSM-tree to translate the logical key to physical key and then retrieve the value from the device use the physical key. However, a small index cache can help reduce the I/O overhead from index lookup.

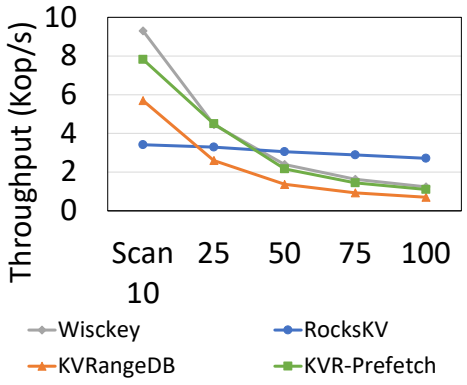
Figure 3.10 demonstrates the performance of simple *get* (or *point query* workload. KVRangeDB outperforms RocksKV by significant amount for both no cache and 1GB scenario. Compared to Wiskey, KVRangeDB performs slightly worse for 1000B value size (packing) due to the block device provide better read performance compared KVSSD. However, KVRangeDB outperforms Wiskey when packing is not initiated (4000B case) by 73% (no cache) and 39% (1GB cache).



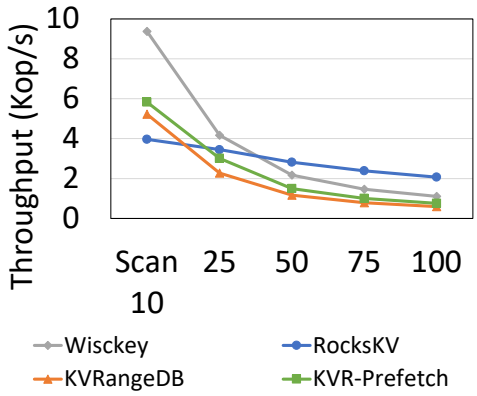
(a) Scan keys throughput for 1K value (16 threads)



(b) Scan keys throughput for 4K value (16 threads)



(c) Scan keys&values throughput for 1K value (16 threads)



(d) Scan keys&values throughput for 4K value (16 threads)

Figure 3.11: YCSB range query performance

3.5.2.3 Scan keys

For the scan key workload, since KVRangeDB only needs to traverse a relatively small LSM tree (only contains keys) compared to RocksKV's LSM-Tree that includes both keys and values which may require more I/Os. KVRangeDB achieves much better performance, ∼8x better compared to RocksKV with 1GB cache as shown in Figure 3.11 (a) and (b). KVRangeDB performs slightly worse compared to Wisckey due to the device read performance disadvantage of KVSSD (Wisckey also only need single I/O to retrieve value after locate the log offset).

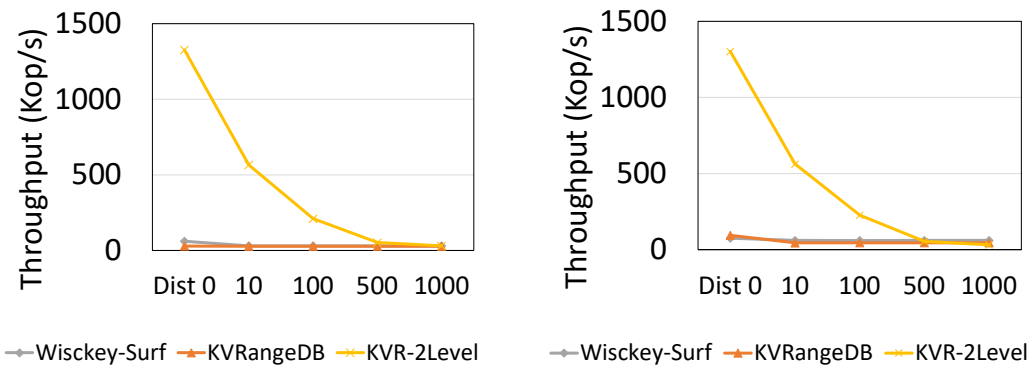
You may wonder if the scanning the keys only (without retrieving values) make sense in real world applications. Here is an example of a typical filesystem workload (which will elaborate in detail in Section 3.5.3). Consider an everyday use command line utility **ls** which list files and sub-directories. In TABLEFS, a **ls -l $path** command translates to a scan on the target directory which needs to retrieve value for parsing stats in the inode. However, a simple **ls $path** command only needs to iterate on the keys for file/sub-directory name resolve without the need to read the value (inode that contains detailed stats).

3.5.2.4 Scan keys and values

On the flip side, for range queries (scans) that needs to retrieve values, KVRangeDB doesn't perform well since it needs to pay separate I/O for each value retrieve. As shown in Figure 3.11 (c) (d), as the scan length passes 40, KVRangeDB= performs worse compared to RocksKV, The optimization of value prefetch with use hints may improve the performance to some extent (∼56%). From analysis of real key-value workloads [76], the average scan length is less than 20. So, it may not worth to pack key and value together like RocksKV which mostly benefit long scans.

3.5.2.5 Empty queries

Figure 3.12 demonstrates the results on various empty point and range queries (Dist 0 denotes point query) for 4000B and 1000B value size. In such cases, filters can be leveraged to reduce the unnecessary I/Os to the device. We compared with Wisckey equipped with the state-of-art range filter design (Surf [2]). Thanks to the unified filter design, KVRangeDB performs 1.7x-21.6x



(a) Empty queries throughput for 1K value (16 threads)

(b) Empty queries throughput for 4K value (16 threads)

Figure 3.12: YCSB empty queries performance (Dist 0 stands for point query)

better than Wisckey for various empty queries cases (For both Wisckey-Surf and KVRangeDB-2levels, filters are fully loaded into memory). If the workloads doesn't contains those empty quires, KVRangeDB can just abandon the filters to save memory usage.

3.5.2.6 Packing with hybrid keys compaction

In this section, we will demonstrate how we can improve get performance by leveraging intelligent keys compaction. The main reason for degradation of get performance for packing is the overhead for logical key to physical key translation. Range query performance isn't significantly impacted by packing since it needs to lookup the key index anyway. To tackle this problem, we propose to compact keys during garbage collection. This process can achieve two benefits. First, less frequently accessed records can be more tightly packed to reduce the number of physical keys managed by the device to yield better put performance. Second, we can selectively remove key translation for *hot keys*. We leverage a small footprint bloom filter to better guide us to avoid this key translation lookup.

Consider the example of the 1 billion records with 1000B value size case. If the portion of *hot keys* are 10%, we can compact 4 cold records into a physical record, we can reduce the number of keys managed by the device by $\sim$ 70%. If we access all *hot key* records with direct user key

access (no key translation), we can achieve comparable get performance while improving the put performance considerably.

Figure 3.13 shows potential gain of the packing approach with key compaction on various mixed put/get workloads. It is noted that we are able to achieve same range query performance compared to non packing case. If we can capture more than 70% of the locality of hot keys for get requests, packing can yield better overall performance on 50:50 put/get workloads. For read heavy workloads, as long as the get requests have good locality, it's useful to employ packing for small records.

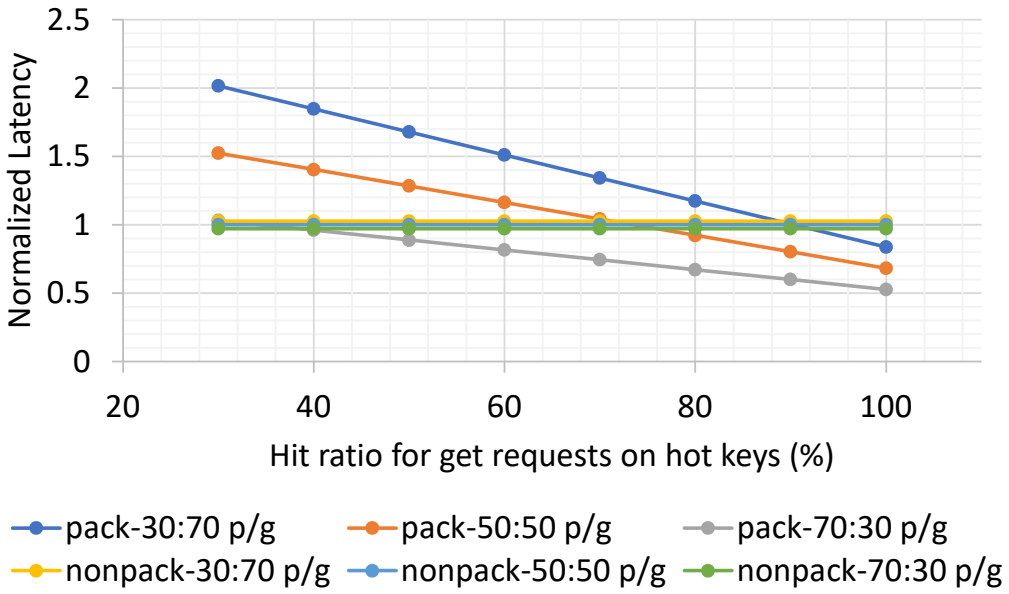
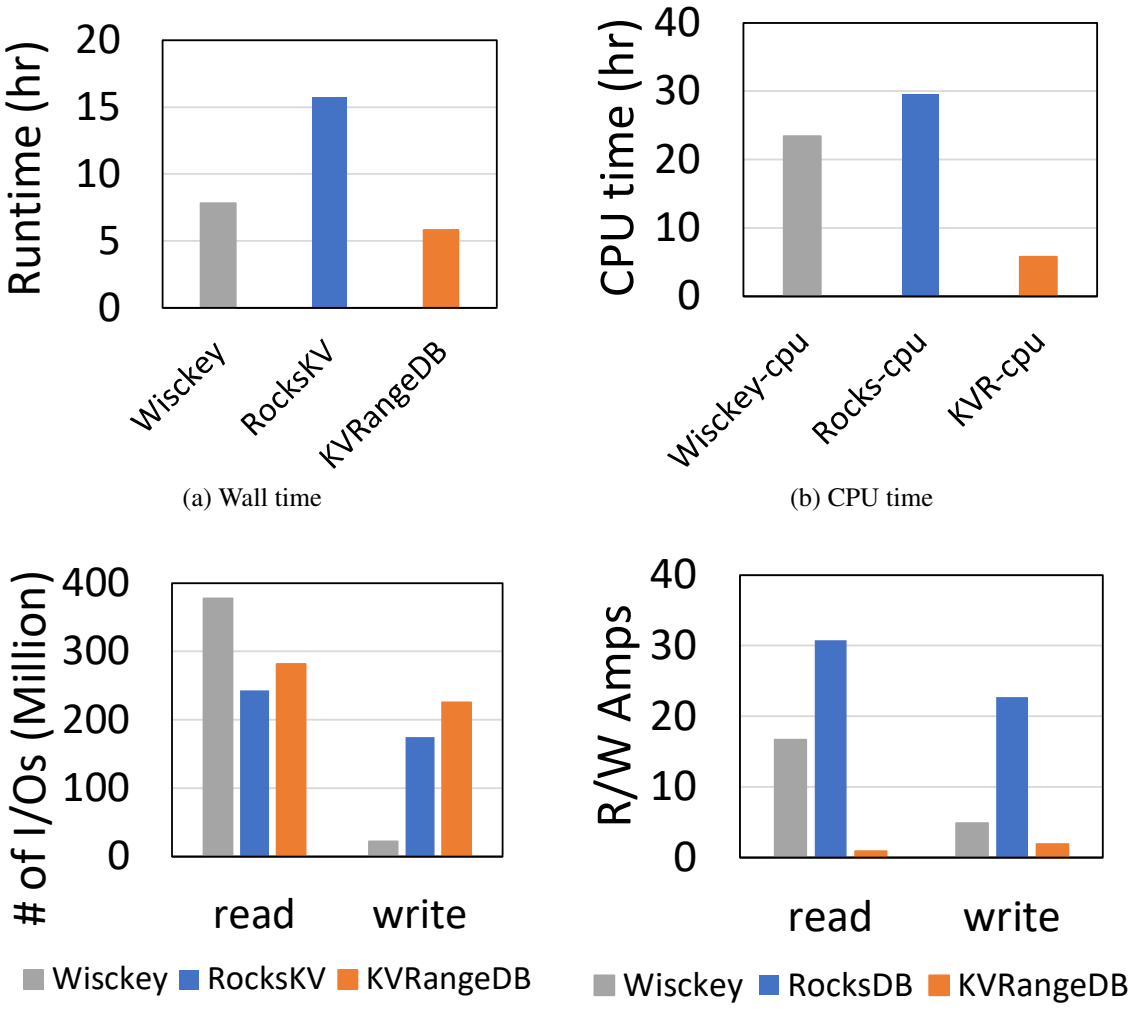


Figure 3.13: Average query latency (normalized to non-packing performance) packing with hybrid keys compaction on mixed put/get workloads.

3.5.3 Results for TABLEFS

For the filesystem workloads, we use real filesystem trace from Los Alamos National Lab which contains approximately 130 million files and directories (∼5 million diretories and ∼125 million files) and load to TABLEFS [84]. The loading phase is consist of multiple file operations



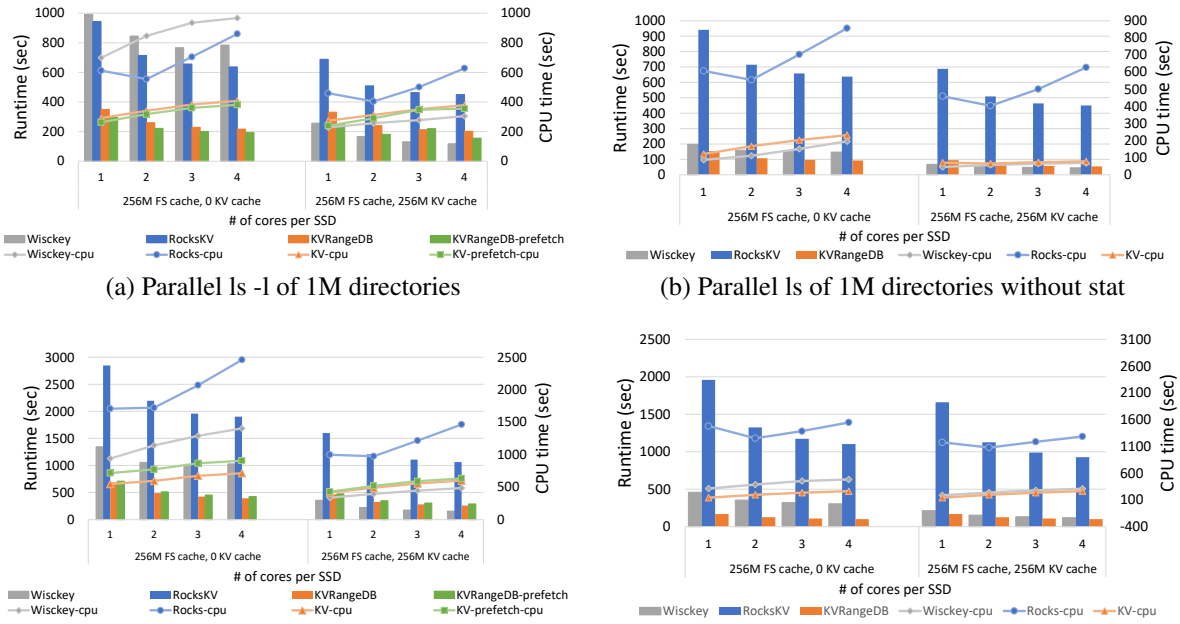
(a) Wall time

(b) CPU time

(c) # of I/O requests

(d) # bytes read/write from/to device

Figure 3.14: Performance for loading filesystem tree to TABLEFS.



(a) Parallel ls -l of 1M directories

(b) Parallel ls of 1M directories without stat

(c) Parallel find from root

(d) Parallel lstat of 5M files/directories

Figure 3.15: Performance for TABLEFS workloads.

including path resolve, opendir, mkmod, unlink, chmod, etc. which translate to combination of **put/**,**get**,**delete** workloads to the KV-store. We compared Wiskey on block SSD, RocksKV and KVrangeDB on KVSSD as the KV-store of TABLEFS.

Figure 3.14 demonstrates the results of loading the directory tree into TABLEFS. KVRangeDB outperforms RocksKV and Wiskey by 2.7x and 1.35x respectively. Besides, KVRangeDB also saves CPU cost by 5x and 4x respectively. We also collect the number of I/O requests and read-/write amplifications from/to the device. RocksKV (conventional LSM-Tree implementation) incurs significantly larger write amplification factor (WAF), 12x worse than KVRangeDB, due to constant compaction of the sorted-runs. KVRangeDB achieves less read amplification compared to RocksKV and Wiskey since direct **get** interface from the device. Wiskey issues larger number of read I/O due to it needs to lookup key to log offset mapping for every get operation (check file path existence) and host garbage collection after remove and update of the records.

Figure 3.15 shows the performance results of for metadata-intensive filesystem workloads as described in Section 3.5.1.2. We use limited number of CPU resources (1-4 physical cores) to

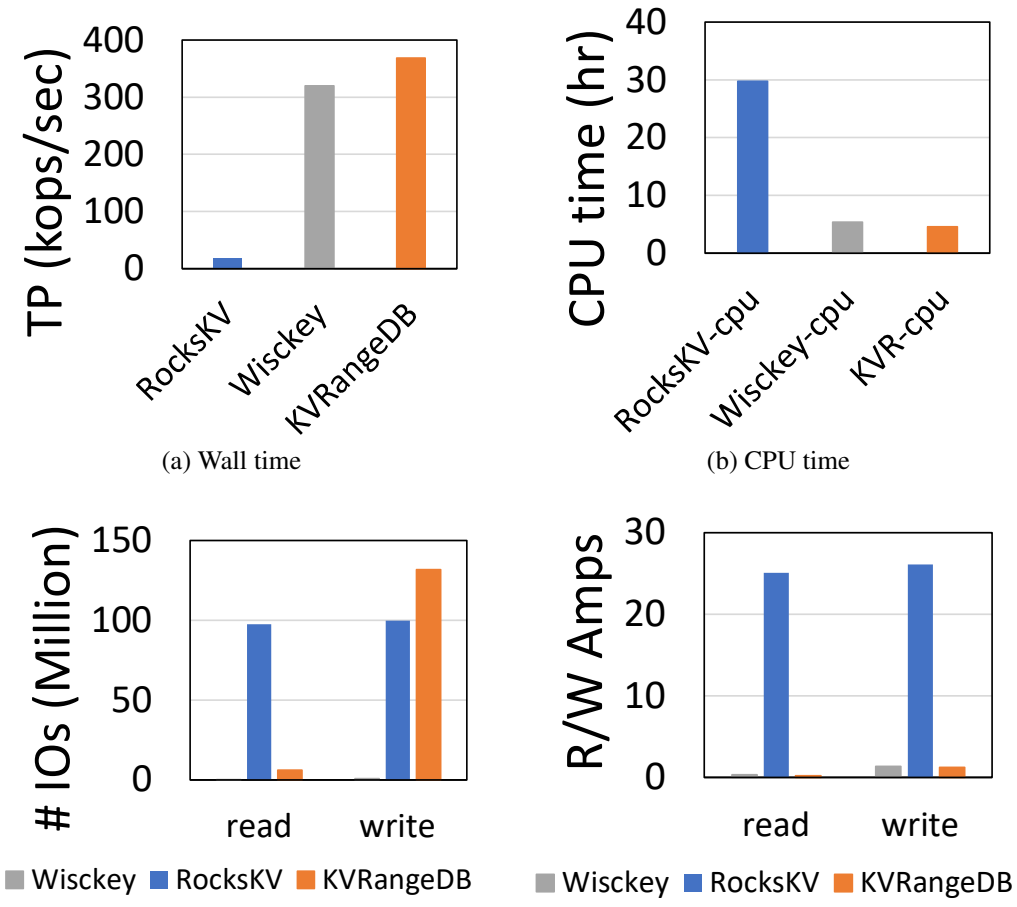
emulate cloud or multi-tenants scenario. We assigns 16 client threads for each physical core. Parallel "ls -l" contains path lookup and readdir which translate to **get** and range query with value retrieval with various scan length (depends on number of files and sub-directories within a queried directory). KVRangeDB which value prefetch yields 3.2x better performance in average compared to RocksKV. This is due to in real filesystem directory tree, there are lots of directories with very few sub-directories and files (leads to short scans). KVRangeDB outperforms Wisckey by 3.8x for smaller cache configuration (Figure 3.15 (a) on the left). This mainly due to the advantage of **get** opertaions for KVRangeDB. With large memory budget, Wisckey performs better simply due to block SSD has better read I/O performance. For simple parallel "ls" without stat (no need to retrieve inode), which converts to range query without value retrieval, KVRangeDB performs 6.6x better compared to RocksKV and 1.5x better compared to Wisckey in small cache configuration. Since RocksDB's SSTable packed key and value together, range queries with keys only requires almost same amount of work compared to range queries that also retrieve values. For both "ls -l" and "ls" workloads, KVRangeDB also use less CPU cycles since the simplified device KV interface which removes the block alignment issue which leads to less read amplifications.

Parallel find workloads which traverse all files/directories in a breadth first search fashion as shown in Figure 3.15 (c), performs similar to "ls -l" workloads. The main difference compared to "ls -l" workloads is that the average scan length is shorter due to lots of the directories in the tree are empty. Thus, prefetching value doesn't gain any performance benefit since there are unnecessary overprefetch.

Figure 3.15 (d) demonstrate the performance of parallel lstat workload which consists of *get* operations only. Compared to RocksKV and Wisckey which requires multiple I/O in average per *get* operation (for RocksKV, it needs to examine multiple sorted-runs or SSTable files, for Wisckey, it needs to lookup the log offset from user key before retrieve the value from log), KVRangeDB only requires single I/O per *get* through the device interface. For small cache configuration, KVRangeDB outperforms Wisckey by 2.9x and reduce CPU usage by 2x. The filesystem workloads showcases the advantages of KVRangeDB in resource (CPU/Memory) limited scenar-

ios.

3.5.4 Results for Time-series Workloads



(a) Wall time

(b) CPU time

(c) # of I/O requests

(d) # bytes read/write from/to device

Figure 3.16: Performance for loading time-series data.

Figure 3.16 demonstrates the results for loading a 500 million events from ten thousands sensors and performing typical time-series queries on it. KVRangeDB achieves 19x better write throughput with 6.5x less CPU usage compared to RocksKV and 15% better performance with 17% CPU reducation compared to Wisckey thanks to the packing of multiple logical records to larger physical records to the device. Time-series workloads usually are handling real-time data

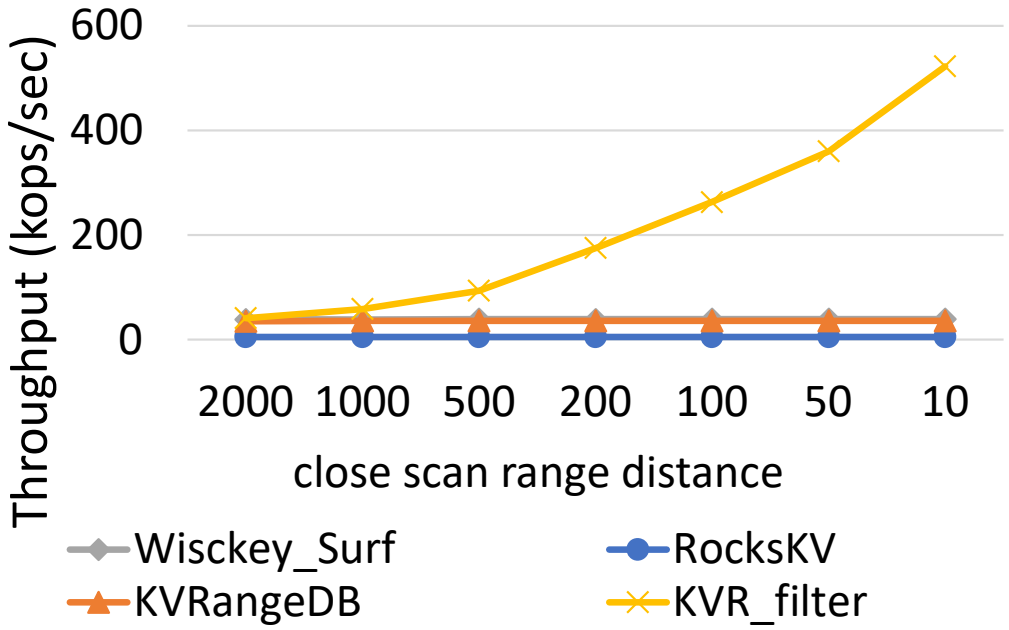


Figure 3.17: Close range query for time-series workloads [2] (Both Surf filter and KVRangeDB hybrid filter cost 16 bits memory per key).

and are write dominated [91, 92, 93]. The superior write advantage of KVRangeDB can greatly benefit such applications.

On the query sides, Figure 3.17 shows performance of mixed of empty and non-empty range queries (donated closed query in Surf [2]) with different scan distance (between start and end key) for the time-series workloads. As shown in the results, for smaller scan distance (less than 500), range filter can significantly improve the overall query throughput by reducing the queries to the key index and device.

3.6 Discussion

3.6.1 KVSSD adoption

The emerging KVSSD is a promising approach for key value workloads. Future NVMe standard will also adopt new I/O command set for key value interface [94]. The most appealing advantage of KVSSD is to remove multiple layers of software (KV engine, file system, block layer) to reduce the host side CPU utilization and external write amplification for key value store appli-

cations. However, such devices may not provide full spectrum of features to satisfy applications requirements.

In this work, we identify that range queries/scan operations are important part of workloads of modern applications, and that these operations are not efficiently supported by the KVSSD devices. We propose a thin layer of software which leverages a log structure merge tree based key index for KVSSD to support efficient range queries. The key index can be also used for packing records through simple key translation to improve the write performance for smaller size records. Our experiments show for certain use cases, our design can achieve similar range query performance compared to state-of-art RocksDB on block SSD, but maintain better put/get performance, making KVSSD devices more appealing to practical applications.

3.6.2 Moving our design to the hardware

It will also be reasonable to move our design concept into the device and expose native range query interface from the device. Such integration may require higher hardware cost and may result in less I/O performance. There are some prototypes that use LSM-Tree based KV engine internally on open channel SSD platform [36]. We hope our work can provide some useful insights for future design of key-value storage devices.

3.6.3 Other structure for index

Besides LSM-Tree, we also considered other external data structures such as B-Tree to implement the key index. We also implemented a prototype B-Tree key index for the KVSSD. Each B-Tree node is write as KV pair which key is the smallest key in the node. However, the downside of B-Tree based index is write performance. Since it needs to perform multiple small updates to each node, it will heavily impact the overall write performance. In the future, we may experiment with more external data structure for ordered index for KVSSD.

3.7 Related Work

Wisckey [25] proposes the idea of separation of key and values for Log Structure Merge Tree based key-value store. The rationale behind it is LSM tree introduce heavy write amplification

during compaction for levels. Wisckey only stores key and block pointer using LSM tree and keep values separately in a log which greatly reduces write amplification. Our KVRangeDB design employs similar idea to store a separate ordered key index, but emphasize on range query on hash based KV-SSD.

Kang *et al.* [28] present first commercial hash based KV-SSD product and some performance evaluation compared with state-of-art software KV stores.

Zhang *et al.*propose SuRF citeSurf which use a compact trie structure as range query filter to accelerate range query performance on LSM tree based software KV stores. However, it only helps for very sparse key space and range queries with large key range.

Kim *et al.*introduce Compound Commands [95] for transaction support for KV-SSD. The compound command idea can also be leveraged in our KVRangeDB design. It can be adapted to logging and SSTable writes for writing the LSM Tree index which can yield overall better write performance for KVRangeDB.

TellStore [77] is a distributed KV store which prioritize the range query. It uses a number of advanced implementation techniques such as piggy- backing garbage collection, to improve locality of scans in the distributed use case. However, our design focus on scan and put/get performance on single node KV store which is in a different problem space.

Various of designs proposed on building key-value store on emerging memory/storage hardware. RocksDB [18] and SILT [33] are designed to leverage the fast flash devices. SLM-DB [96] and FlatStore [97] are key-value stores designed for emerging persistent memory which leverage the byte addressability. Microsoft proposed KV-DIRECT [98] which is designed to leverage emerging programmable network interface [99, 100] for in-memory key-value stores. Our work leverage the fast direct key-value interface device and propose efficient structure for supporting range query capability which the device don't provide natively.

The concept of data lake [101, 102, 103] was proposed to for big data era to store large volume of unstructured or semi-structured data with low cost storage and limited query ability. The data will then extract, transform and load (ETL) from the data lake and stored in data warehouse

or to the applications such as in-depth data analysis, machine learning, etc. In such architecture, our KVRangeDB can be a perfect platform for the data lake systems with a balance of performance (high write performance), cost(large volume, low CPU utilization) and query ability (efficient point/range query capability).

3.8 Summary

In summary, we proposed and implemented KVRangeDB to support efficient range query capability on hash based KVSSDs. Our design leverages a log structure merged tree based secondary key index which can optionally pack records through logical to physical key translation to mitigate the key management overhead in the KVSSD device. Besides, we employ user hints for value prefetching to accelerate scans with value retrieval and leverage state-of-art range filter to efficiently improve empty range/point queries.

Our evaluation on real world applications justify the superiority of KVRangeDB over the state-of-art software KV engine (RocksDB) on conventional block SSD on faster *put*, *get*, short scans performance and lower host CPU utilization. For workloads that require extremely long scans, KVRangeDB may not be the best platform to serve that. However, KVRangeDB will shine out if the workloads rarely retrieve value during scans.

4. A GENERIC FPGA ACCELERATOR FOR MINIMUM STORAGE REGENERATING CODES*

This section presents a generic FPGA accelerator for MSR codes encoding/decoding for future storage application offloading. We first introduce a background of Minimum Storage Regenerating (MSR) code. Then, a detailed design and implementation is described in the design section. The evaluation section compares our design with the state-of-art CPU and GPU implementation of MSR code.

4.1 Introduction

With the explosive growth of data in the era of cloud computing, reliability is a major concern in storage systems as their underlying components are highly susceptible to write induced wear [104]. Traditionally, replication schemes are used to provide fault tolerance. However, as the enormous scale of data volume demands, more sophisticated erasure coding techniques are used to minimize storage overhead. Currently, Maximum Distance Separable (MDS) codes, such as Reed-Solomon codes, are widely employed in both local storage systems [105] and large distributed storage systems [106, 107].

Although MDS codes provide significantly better reliability, while sacrificing the least amount of storage overhead, they impose a huge burden on repair bandwidth when rebuilding data in the event of failure [108]. Recently, a new class of erasure codes called Minimum Storage Regeneration (MSR) codes have been proposed [108, 109, 110] as an alternative to MDS codes. MSR codes minimize the data required for rebuilding while maintaining optimal storage efficiency. Although MSR codes reduce the amount of data required for rebuilding, the computation cost for encode and decode remains high, comparable to MDS codes, which are highly CPU and memory intensive [111, 112, 113]. Table 4.1 shows the experimental results for a specific MSR code (Zigzag

*Reprinted with permission from "A Generic FPGA Accelerator for Minimum Storage Regenerating Codes" by M. Qin, J. H. Lee, R. Pitchumani, Y. S. Ki, A. L. Narasimha Reddy, P. V. Gratz 2020. Proceedings of the 25th Asia and South Pacific Design Automation Conference, Copyright 2020 by IEEE

code) encoding using GF-Complete library [4] on a modern Intel CPU. As shown in the table, the encoding throughput doesn't scale well with increased number of threads. This is caused by poor cache performance which saturates the system DRAM bandwidth. Thus, it's worth considering designing more efficient hardware architecture to offload erasure coding computation from CPU.

Table 4.1: Zigzag encode performance for 64MB object size using GF-Complete library [4] "Reprinted from [3]".

# of threads	1	4	8	12	16
Throughput (GB/sec)	2.18	7.67	10.64	10.96	10.98
LLC hit rate	0.4	0.014	0.02	0.007	0.007
DRAM util (GB/sec)	9.53	40.99	59.55	63.22	64.60

Traditional accelerators such as GPUs and FPGAs suffer from extra data movement between host and accelerator memory [114]. However, recent efforts of RDMA NICs [115] and the emerging PCIe peer-to-peer (P2P) communication between PCIe devices [116] (such as NVMe SSDs, NICs and accelerators) enable inter and intra server data movement to be almost free with minimum CPU intervention. With these efforts, the offloaded erasure coding computation can be carried out in the accelerator on the fly without moving data back and forth between the host and the accelerator. These makes offloading erasure coding computation further appealing.

The above observations motivate us to design efficient accelerators for MSR erasure code, which can free the host CPU and memory for supporting other applications; a solution that is both economical (cheap hardware versus expensive server CPU) and power/energy efficient. Considering erasure coding is pure fixed-point computation, FPGA is a more efficient platform compared to floating-point optimized GPU.

In this section, we describe a generic FPGA accelerator to perform the code construction and data rebuild for Minimum Storage Regenerating Codes. In our design, we leverage the abundant logic and memory resources in FPGA to provide massive parallelism for encode/decode computa-

tion and reduce unnecessary data movement between off-chip DRAM and FPGA on-chip BRAM buffer through analyzing the memory access pattern for MSR code construction and data rebuild. We implement our accelerator on a Xilinx VCU1525 board and compare against the state-of-art software MSR code implementation with GF-Complete library [4]. Our proposed design shows superior benefits on both performance and power efficiency.

To summarize, we make the following contributions:

I A generic hardware architecture to process code construction and data rebuild for MSR codes. This architecture maximizes parallelism for the finite field operations used in erasure codes and minimizes data movement from off-chip memory, to address the problems in traditional CPU implementation.

II Demonstration of a flexible and easy to maintain OpenCL implementation leveraging Xilinx High Level Synthesis to implement such an accelerator for MSR code construction and data rebuild.

III Experimental evaluation of the proposed approaches on a state-of-art FPGA accelerator card, comparing performance with CPU and GPU implementation.

The rest of this section is organized as follows. In Section 4.2, we briefly describe the code construction and rebuild algorithms for a specific MSR codes named Zigzag codes which is used in this work. Then, we introduced the emerging PCIe P2P architecture and MSR codes software implementations limitations which motivate this work. Section 4.3 describes our proposed architecture for generic MSR codes accelerator. The overall system implementation for the accelerator and experiments evaluation are presented in Section 4.4. Finally, we summarize the chapter in Section 4.5.

4.2 Background

In this section, we briefly describe the theory of erasure coding and Minimum Storage Regeneration (MSR) codes. Then we demonstrate the code construction and data rebuild algorithms for a specific MSR code called Zigzag [109] code.

4.2.1 Erasure Code and MDS codes

In storage systems, erasure codes are exploited to tolerate storage failures with less extra storage. Maximum Distance Separable (MDS) codes achieve ideal storage overhead. Consider an erasure coded system composed with total number of n nodes. We split them into k information nodes and $r = n - k$ parity nodes. We denote the erasure code configuration as *{n, k}*, and we refer to a node as an independent failure point such as a disk or a storage node in the data center. We stripe the data object (a.k.a. stripe) into k even size information fragments and apply erasure codes to generate r even size parity fragments and store them in the information nodes and parity nodes respectively. MDS codes have the property that they can recover from up to $n - k$ failures of any nodes.

The encoding procedure of MDS codes can be generalized as linear arithmetic operations in Galois Field as shown in equations 4.1 where each element in the matrix is a codeword (minimum data size to operate in Galois Field). The decoding procedure for m-node failure ($m \leq n-k$ where $n - k$ is the maximum number of nodes failure that MDS codes can tolerate) can be achieved by solving the linear equation 4.1 (the coefficients matrix C must be invertible to guarantee the feasibility of decoding).

$$\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,k} \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} \tag{4.1}$$

4.2.2 Minimum Storage Regenerating (MSR) codes

Regenerating codes were first introduced by Dimakis *et al.* [108] to reduce the high repair bandwidth of MDS codes in distributed storage systems. Minimum Storage Regenerating (MSR) codes offer the same storage-availability trade-off as MDS codes while minimizing the repair bandwidth. Here we will briefly introduce the specific MSR code used in this section, Zigzag [109] code, with

an example to intuitively illustrate how MSR codes generally work. Other MSR codes [110, 117] follow the same principles.

Zigzag encode: The data object to be stored will be first split into k even fragments. Each fragment is further partitioned into m data elements as shown in Figure 4.1 (when $m = 1$, it degenerates into to MDS code). In this section, we will refer *{n, k, m}* as the configuration parameters for Zigzag code where n is the total number of storage nodes. (For detailed zigzag code parameters, please refer to [109].) The Zigzag code parities are encoded as follows:

I For each data element in a parity fragment, find a specific data element in each information fragment (the specific data element index is determined by the code design), totally k data elements.

II Each data element in the parity fragment is generated by the k corresponded information data elements using Galois Field operations with the following formula:
$p_i = \sum_{j=1}^{k} C_j d_{j,i} \quad (1 \leq i \leq N)$
Where N is the number of codewords in each data element.

We generalize several parameters for the above procedure. For each data element in the parity fragments, there is a set of indices $\{I_1, I_2, \cdots, I_k\}$ indicating the location of the data element in each information fragment and a set of coefficients $\{C_1, C_2, \cdots, C_k\}$ for calculating the parity data element. In total there are $(n - k) * m$ sets of those indices/coefficients parameters to finish the entire encode procedure.

To better understand the description above, consider an MSR coded storage system with 4 information nodes and 2 parity nodes as shown in Figure 4.1. Each data fragment contains 8 data elements. Codewords in the first and third data element of the first parity fragment are calculated as:

$$P1_{r1} = 1 * D1_{r1} + 1 * D2_{r1} + 1 * D3_{r1} + 1 * D4_{r1} \tag{4.2}$$

$$P2_{r3} = 1 * D1_{r3} + 2 * D2_{r4} + 1 * D3_{r1} + 1 * D4_{r7} \tag{4.3}$$

The corresponding indices sets are $\{1, 1, 1, 1\}$, $\{3, 4, 1, 7\}$. The coefficients sets are $\{1, 1, 1, 1\}$, $\{1, 2, 1, 1\}$.

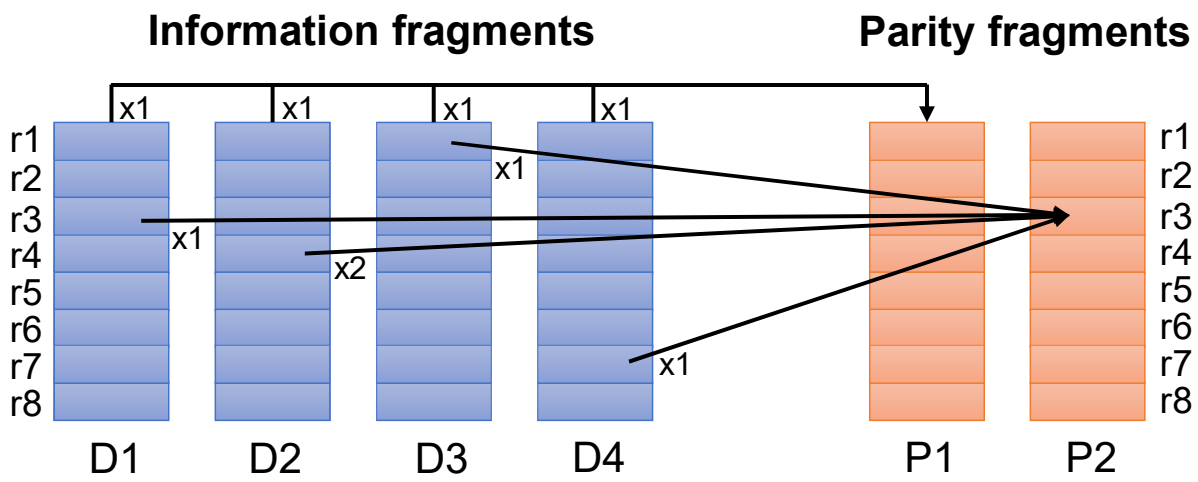


Figure 4.1: MSR codes encode example "Reprinted from [3]".

Zigzag decode: In this work, we focus on MSR code rebuild for only the single erasure case, since single node failure is the most common case [118].

The data rebuild formula for single erasure is nearly identical to the code construction formula (linear algebra transformation). Similarly, we define indices set $\{I_1, I_2, \cdots, I_k\}$ to indicate the location of the data elements in surviving information/parity fragments needed for rebuild (each rebuild data element is generated from k information/parity data elements [109]) and coefficients set $\{C_1, C_2, \cdots, C_k\}$ for calculating rebuild data words in each data element. As shown in Figure 4.2, the rebuild data in the first and third data element in the erased fragment is calculated as:

$$D1_{r1} = 1 * P1_{r1} + 1 * D2_{r1} + 1 * D3_{r1} + 1 * D4_{r1} \tag{4.4}$$

$$D1_{r3} = 1 * P2_{r3} + 2 * D2_{r4} + 1 * D3_{r1} + 1 * D4_{r7} \tag{4.5}$$

As illustrated in Figure 4.2, the rebuild for single erasure case for MSR codes require much less the data compared to conventional MDS codes such as Reed-Solomon codes.

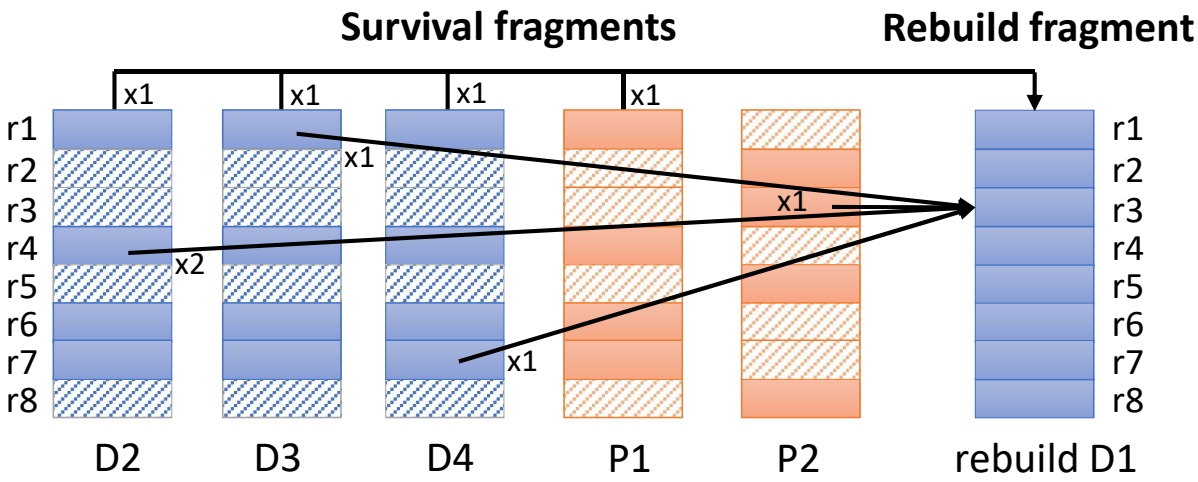


Figure 4.2: MSR decode example (The solid filled boxes are the data needed for rebuild.) "Reprinted from [3]"

4.3 Proposed Architecture

In this section, we will describe the accelerator architecture for encode/decode offloading for Zigzag code. While it is intended for Zigzag code, this architecture can be easily extended to other MSR codes.

The overall diagram for our proposed architecture is shown in Figure 4.3. The architecture is mainly composed of two components. First, the *memory unit* holds the information and parity fragments that are transferred from host memory or storage devices. The memory unit uses the off-chip DDR memory connected to the FPGA. Second, the *processing unit* which process the data

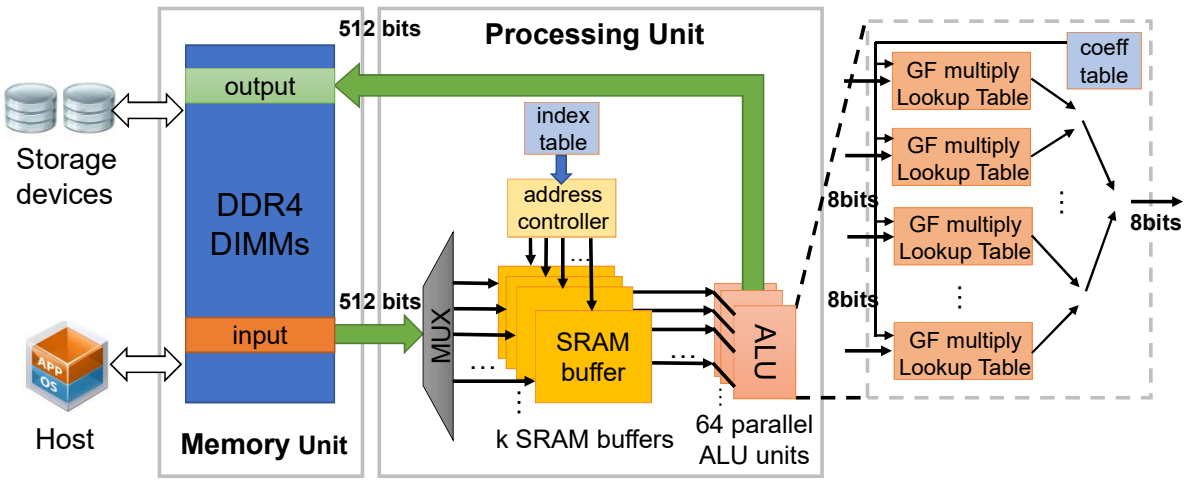


Figure 4.3: Overall accelerator architecture "Reprinted from [3]".

from the memory unit and perform the actual encode/decode computation. There can be more than one *processing unit* connected to the *memory unit* to fully utilize the off-chip DRAM bandwidth and hide memory latency, assuming FPGA resources are available.

4.3.1 Memory Unit

The memory unit temporarily holds the input data for the encoding/decoding processing and the output results (parity fragments and rebuild information fragment for encode and decode respectively). For encoding, the information fragments will be transferred to the memory unit from the host. The encoded parity fragments will be written back to the memory unit after processing unit fetches the information fragments and finishes processing. Finally, the information and parity fragments will be transferred to the storage nodes through P2P transfer. For decoding, the data fragments needed for rebuild will be transferred to the memory unit from the surviving storage nodes through P2P transfer. After the processing unit finishes the decoding process, the rebuild data will be stored in the memory unit and transferred back to the host or to a new storage node depending on the recovery process. All the input/output buffers in the memory unit are allocated/deallocated through the OpenCL framework dynamically.

4.3.2 Processing Unit

The processing unit consists of mainly three parts. The SRAM buffers which hold all or part of the input data for the encoding/decoding process. The address calculation controller which manages how the data is fed in the SRAM buffers from the memory unit and how the data is read from the SRAM buffers for encode/decode computation and how the results are written back to the memory unit. The ALU unit which computes the Galois Field multiply-add arithmetic.

SRAM buffers: In each processing unit, we use k separate SRAM buffers where k is the number of information nodes in our Zigzag code configuration to hold partial or all of input data for computation. The SRAM buffers are implemented using the BRAMs in the FPGA. The SRAM buffers are a key design to minimize the traffic to the memory unit. Taking encode process as an example (which is similar to the decoding process), remember that each codeword in the parity fragments is generated by operating on k codewords from k different information fragments with different relative offsets. To improve the data reuse rate, we need to buffer all the data elements for every information fragment in the SRAM for future use. Thus, k SRAM buffers will buffer all the codewords required to calculate the codewords for all parity fragments. With the design of k separate SRAM buffers, each byte of the input data only needs to be read **once** from the memory unit to the SRAM buffers once which significantly minimize the data movement between off-chip DDR memory and FPGA logic.

To maximize the memory unit bandwidth utilization and the process throughput, the data are packed to 512 bits when being transferred from or to the memory unit. Each memory buffer is organized as 512 bits width dual-ports RAM. Thus, the data is read, written and processed in 512 bits granularity per cycle in the processing unit.

The detailed illustration for the memory layout of the input and output data in the memory unit and how data is moved into the SRAM buffers will be demonstrated in section 4.3.3.

Address calculation controller: The address calculation controller is the most complex control unit. It has three tasks.

- Read the data from memory unit input buffer to k SRAM buffers. This includes slicing the

data elements and read per sliced data in each data element to the SRAM buffers when stripe size is too large.

- Read the data from SRAM buffers in parallel and feed to the ALU units for the encoding/decoding computation (Galois Field arithmetic).
- Write the results (parity codewords or rebuilt data words) to the output buffer in the memory unit.

Once the Zigzag code configuration $\{n, k, m\}$ is fixed, the indices sets for accessing the information fragments to generate each parity data element are also fixed. We pre-calculate these indices sets offline and use a table to store these indices sets in the FPGA. These indices sets will be used for the address calculation controller to fetch the data from SRAM buffers to the ALU units to perform the computations.

ALU unit: The ALU unit are the core computation logic to perform the Galois field arithmetic to generate the parity and do data rebuild for Zigzag code. As we discussed in section 4.2, both encode and decode process for the Zigzag code or any other erasure codes are composed of only Galois Field multiply-add operation. Thus, our ALU unit is designed to perform only Galois Field multiply-add operation. In our implementation, we use lookup tables to implement Galois Field multiply and bitwise XOR to implement Galois Field add which can make the most of the massive LUT resources in FPGA. All Galois Field operations are in 8-bit granularity which is a good parameter for lookup table size. Unlike the "single-instruction-multiple-data" (SIMD) unit in the CPU which only operates on two input operands, we leverage the abundant logic resources in the FPGA and designed a pipelined tree structure to perform multiple inputs gf multiply-add operations in pipeline as shown in Figure 4.3 on the right side. Similar as the indices sets, we store the fixed coefficients sets as tables in the FPGA to compute parities.

The pipelined tree structure for Galois Field multiply-add operation in our design has two advantages compared to the SIMD unit in the CPU. First, data is processed with better parallelism. Second, to generate each output codeword, each input codeword (operand) only needs to be read once from the SRAM buffers. While in the CPU implementations, this needs to be done in a loop

to read the input codewords (operands) from cache iteratively. Since the useful cache lines may be evicted to lower level cache or even DRAM, this will cause stalls in the SIMD pipeline and extra power to move the data.

4.3.3 Process Stages

To better demonstrate how our accelerator works, we will describe the process stages for single encode or decode task. Since the computation and data flow for encode and decode are similar, we do not differentiate encode and decode.

The processing unit is able to handle an arbitrary length stripe size. This is important for erasure codes since different storage systems may require different stripe sizes. Thus, the process stages for each encode/decode task may include one or more passes, each process pass contains three phases as follows.

Read phase: In the read phase, the address calculation controller will control the memory read from the off-chip memory unit and write to the SRAM buffers. Each SRAM buffer holds part or all of the input data fragment. If the size of the input data is small enough that can be filled entirely in the SRAM buffers, the whole process will be done in one pass. However, if the size of the input data is larger than the SRAM buffers, the input data will be partitioned properly and read into to the SRAM buffers for further processing. In this way, the whole process will be done in several passes.

To maximize the off-chip memory unit bandwidth performance and reduce energy, the partitioned areas are 4KB to match the internal DRAM page size to improve row locality. If the stripe size is small enough that can be filled entirely in the SRAM buffers, all the data will be fed into the SRAM buffers in one pass (sequentially read for each fragment). If the stripe size is too large, the data will be read from memory unit data slice by data slice to the SRAM buffers.

Computation phase: In the computation phase, the processing unit will apply the code construction and data rebuild algorithm described in section 4.2.2. The indices sets and coefficients sets for the data element will be applied here for each data slice. The address calculation controller unit will control the memory read according to the pre-stored indices table and read the correct

data slices from the k SRAM buffers simultaneously. The read data will be fed to the ALU units for parity calculation or data rebuild as described in section 4.2.2.

Write phase: Since computation phase is fully pipelined, the output results from the ALU units can be written to the off-chip memory unit immediately. It can be considered as adding one more pipeline stage after the XOR tree. Since the data is partitioned when read into the SRAM buffers, the output results' written back to the memory unit is also partitioned. In the first process pass, the parities generated will be written to the output fragments. In the second process pass, the parities generated will be written to the output fragments.

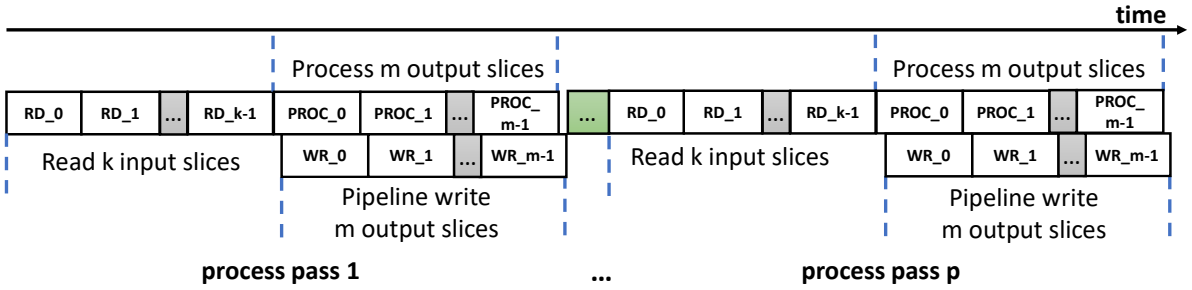


Figure 4.4: Timing diagram of the process stages workflow "Reprinted from [3]".

Figure 4.4 illustrates the timing diagram of the three process stages for the accelerator. Consider Zigzag code n, k, m, there are k input data fragments and $r = n - k$ output data fragments. Let's take encode procedure as an example here (for decode is similar). The input data is larger than the internal SRAM buffers size and it needs p passes to process the whole input data and p is equal to the number of data slices partitioned for each input data element.

4.3.4 Other Considerations

Multiplexing resources for encode and decode: Since both the data flow and computation for encode and decode are similar, as shown in section 4.2.2. We can multiplex most of the hardware resources (SRAM buffers, ALU units) to conduct both encode and decode procedure. In our design, we have separate tables to store the indices/coefficients sets for encode and decode.

The host can set up different kernel parameters to control the kernel launch of different functions (decode or encode).

Batch processing: For processing small size data, the kernel launch overhead and data migration overhead from host to accelerator and vice versa is non-negligible. In our design, we also implement a batch process to process multiple same size input data in single kernel launch. The batch size is also a separate parameter for setting up the kernel. The batch process support is implemented by slightly modifying the address calculation controller to continuously read, compute and write after finishing each encode or decode task.

4.4 Implementation and Evaluation

4.4.1 System Setup

We implemented our accelerator for a *{6, 4, 8}* Zigzag coding system on a Xilinx Virtex UltraScale+ FPGA VCU1525 acceleration card with 4 DDR4-2400 SDRAMs. The GPU implementation is on a Tesla K80 GPU acceleration card with 240 GB/sec GDDR5 memory. The host machine has a 2.1GHz Intel Xeon Gold 6152 CPU with 22 cores and a 30.25MB L3 cache. There are 4 DDR4-2666 SDRAMs on the host machine. Although we only implement and evaluate on a local storage system, the results can be also extended to distributed storage systems. Our FPGA accelerator is developed in Xilinx SDaccel toolchain. Software CPU implementation is developed in C++ with GF-Complete library [4]. GPU implementation is developed using CUDA toolkit.

We evaluate Zigzag code encoding/decoding a wide spectrum of object size (stripe size) from few kilobytes to tens of megabytes for potential use cases. Usually RAID systems use smaller (64KB to 256KB) stripe sizes [105, 109], while the cloud storage [119, 120, 110] industry tends to use much larger stripe sizes, on the order of tens of MB.

4.4.2 Resource utilization

The FPGA resource utilization and kernel frequency are shown in Table 4.2. This implementation uses all 4 DDR4 channels on board and each channel (memory unit) implements three processing units (PUs). We use 32KB SRAM buffers for each PU (4KB buffer per storage node

to maximize the DDR bandwidth utilization). The resource utilization and timing result include platform cost for implementing OpenCL framework and are post route results.

Table 4.2: System resource utilization on VCU1525 accel. board "Reprinted from [3]".

Resource Type	Used	Available	Util%
CLB Registers	552005	2364480	23.35
CLB LUTs	376287	1182240	31.83
Block RAMs (36Kb)	1050	2160	48.61
Kernel clock frequency	300MHz		
Platform clock frequency*	300MHz		

* Platform clock include the clock domain for OpenCL implementation (memory controllers, PCIe endpoints, interconnect, etc.)

4.4.3 Performance of Zigzag encode/decode

Here we compare our FPGA implementation against the state-of-art CPU implementation leveraging SIMD instructions [4] and GPU implementation. For the software CPU implementation we use different numbers of threads to process in parallel (each thread processes a complete encode/decode task). For GPU implementation, each thread processes only a few 32 bits GF multiply-adds for a encode/decode task to fully exploit the "single-instruction-multiple-threads" (SIMT) parallelism. We conduct experiments on a wide range of data object sizes from (tens of kilobytes to tens of megabytes). For software implementation, the CPU runs at 2.1GHz with 85.3 GB/sec memory bandwidth. The FPGA accelerator runs at 300MHz with 76.8 GB/sec memory bandwidth. GPU accelerator runs at 875MHz with 240 GB/sec memory bandwidth. As shown in Figure 4.5, compared to peak CPU implementation, our FPGA accelerator achieves similar performance for smaller stripe size and 3.1x better on encode and 2.4x better on decode for larger stripe sizes. Our FPGA accelerator also surpasses the GPU implementation by $\sim$2-3x.

There are two reasons that our accelerator achieves better performance. First, our accelerator design optimizes the data fetch and store from the memory unit to the on-chip SRAM buffers and

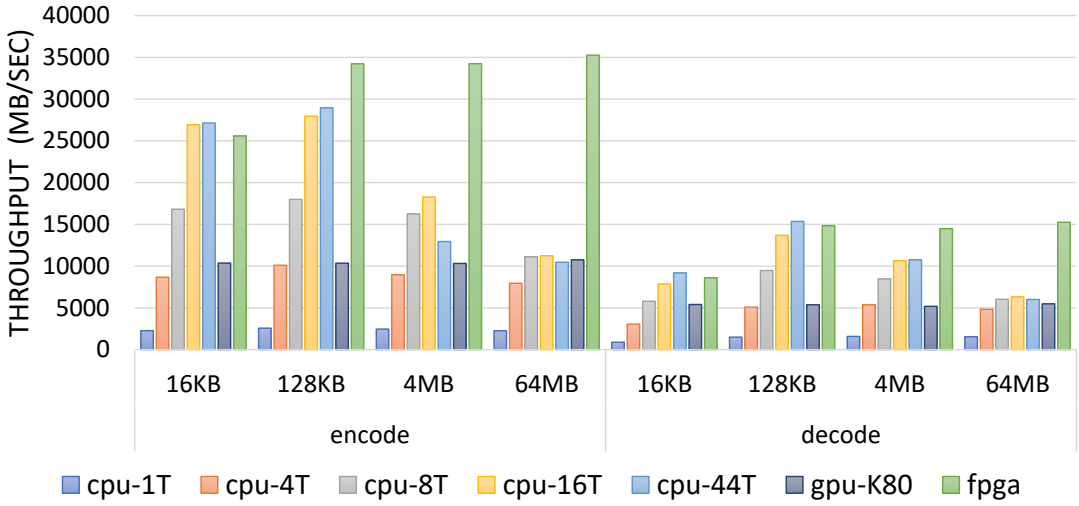


Figure 4.5: Encode/decode throughput performance results (We enable 4MB batch mode for 16KB and 128KB stripe size for both FPGA and GPU) "Reprinted from [3]".

has much better DRAM bandwidth utilization. We collected the memory traffic for the software implementation via performance counters and compared against our accelerator. Our accelerator can reduce up to 20% of the DRAM traffic compared to CPU and 43% compared to GPU. The extra DRAM traffic in the CPU implementation is caused by poor cache performance and cache thrashing in multi-core workloads for large stripe size. Second, our accelerator achieves better computation parallelism by using multi-operand GF multiply-add ALUs compared to two operands SIMD ALUs in CPU architecture. Compared to GPU, the hardware level parallelism in FPGA is much more efficient than SIMT. Thus, even though our accelerator runs at much lower frequency and memory bandwidth ($\sim$3x less than GPU) the performance still surpasses the CPU SIMD and GPU implementation.

Although GPU is not efficient for erasure coding workloads due to GPU is not optimized for fixed-point computation. We still implement Zigzag code on GPU to compare performance. The better performance of GPU on encode mainly comes from high memory bandwidth. The Tesla V100 card has 900GB/sec memory bandwidth which is 10x better than our CPU and FPGA platform (The memory bandwidth utilization on GPU card is 78% and 63% for encode and decode

respectively). We believe if the FPGA platform has comparable memory bandwidth, our FPGA implementation will perform much better than GPU.

4.4.4 Power efficiency

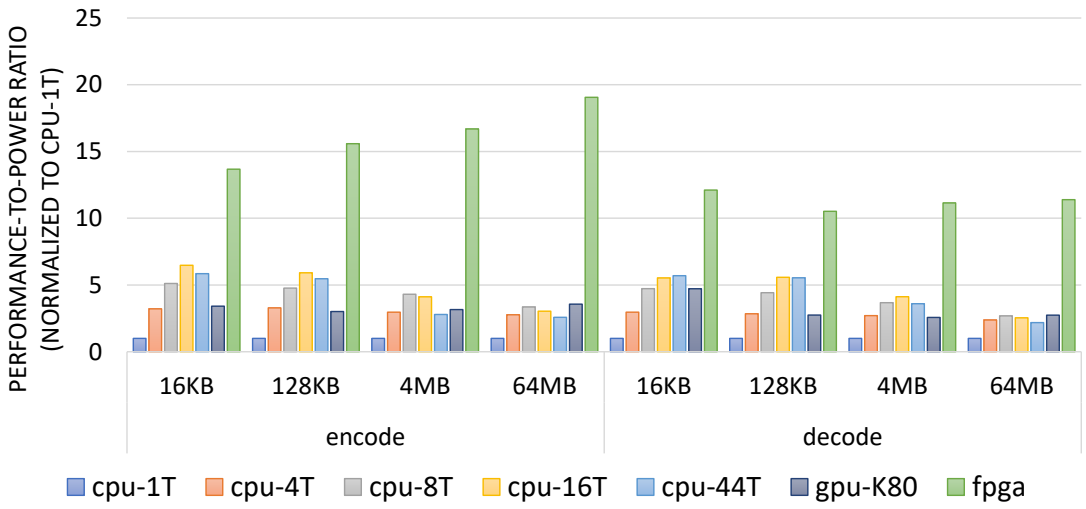


Figure 4.6: Encode/decode performance-to-power ratio (We enable 4MB batch mode for 16KB and 128KB stripe size for both FPGA and GPU) "Reprinted from [3]".

We also did the performance-to-power ratio analysis to estimate power efficiency. We calculate the total power of core (CPU or FPGA) and off-chip DRAMs. For the CPU implementation we obtained the dynamic power consumption through the Intel Performance Counter Monitor. For GPU implementation we obtained the overall power consumption through the GPU driver. We use the Xilinx SDaccel toolchain to estimate the FPGA power (worst case scenario) and a power calculator by Micron to estimate the accelerator DRAM power consumption to get the overall power of our accelerator. Figure 4.6 shows performance-to-power ratio comparison. Our accelerator achieves up to 19.1x and 11.4x better compared to single thread CPU implementation on encode and decode respectively. Compared to the best CPU implementation results, our accelerator is 5.7x and 4.2x better on encode and decode respectively. Compared to GPU implementation results, our accel-

erator is 5.3x and 4.1x better on encode and decode respectively. We also analyze the raw power consumption data for the CPU implementation and our accelerator. We found our accelerator consumes less power on both core (worst case) and DRAM since the FPGA runs on a much lower clock frequency, and we significantly reduce the DRAM traffic.

4.5 Summary

In this chapter, we present a generic FPGA accelerator architecture for Minimum Storage Regenerating (MSR) codes in reliable storage systems. In our design, we leverage the abundant FPGA logic and memory resources to provide massive parallelism for encode/decode computation and optimize the data movement between off-chip DRAM and FPGA. Under evaluation on real systems, we show our proposed accelerator's performance surpasses the state-of-art multi-core CPU implementation on both throughput and power efficiency. The design can be beneficial for storage system acceleration especially with PCIE P2P communication enabled.

5. MULTI-TIERING FOR KVSSD AND OTHER USE CASES ENABLED BY LOGICAL KEY REMAPPING

5.1 Introduction

During our research, we found several interesting use cases empowered by the logical key remapping for key-value interfaced devices. First, remapping logical keys to physical keys enables packing multiple application records (or objects) into a single physical object [1]. Packing different number of records into a record may increase latency and throughput performance differentials when compared to non-packed records when seen from the application side. Conventional processor memory hierarchy [121, 122, 123] which unifies various tiers of memory/storage medium with different performance characteristics into a single level of memory interface. With key mapping and selective packing, we can effectively design different tiers of service levels within a single key-value device. Such multi-tiering functionality can be leveraged for applications with clear boundaries of hot and cold data [124, 93] or service-level agreement (SLA) [125, 126] driven Quality of Service (QoS).

Second, logical to physical key remapping paves the path for flexible key-value interfaced device arrays. It's similar to the idea of logical volume management [46, 127, 128, 129, 130], which unify multiple block-based storage devices into a single logical storage device. By enabling logical key to physical key remapping, we can easily manage multiple key-value interfaced devices while supporting I/O, capacity load balancing across multiple devices.

5.2 Design overview

5.2.1 Performance multi-tiering through logical key remapping

By remapping the logical keys or application keys to physical keys for key-value devices, we can effectively pack multiple logical KV records into a single physical KV record as we discussed in Section 2.3.3 and Section 3.3.2. Such packing introduces performance implications for both put and get operations. First, packing multiple logical records effectively reduces the number of

I/Os issued to the key-value device and increases the average I/O size. Thus, packing can improve the overall application put throughput. By tweaking the number of records packed together, we can provide different leves of application put throughput performance. Second, due to the logical key to physical key translation, which introduces extra I/O overhead when performing get queries, packing multiple logical records will impact get performance. Larger value size (packing more records) also decreases the overall get performance since it takes more device bandwidth for reading a larger amount of data (the other records packed with the target record we are reading).

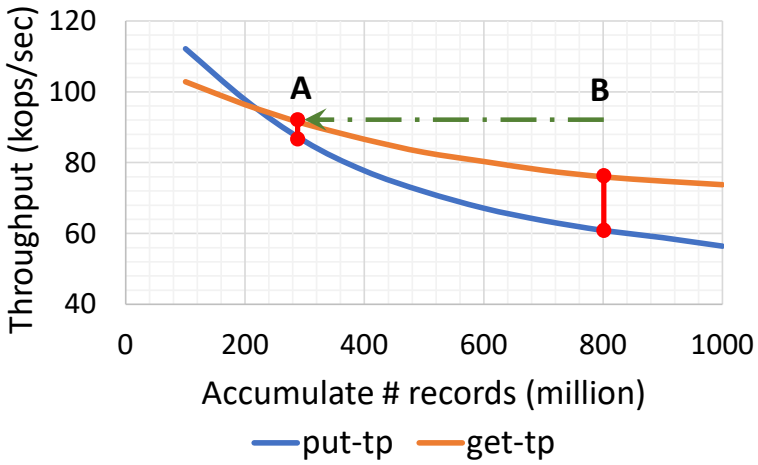


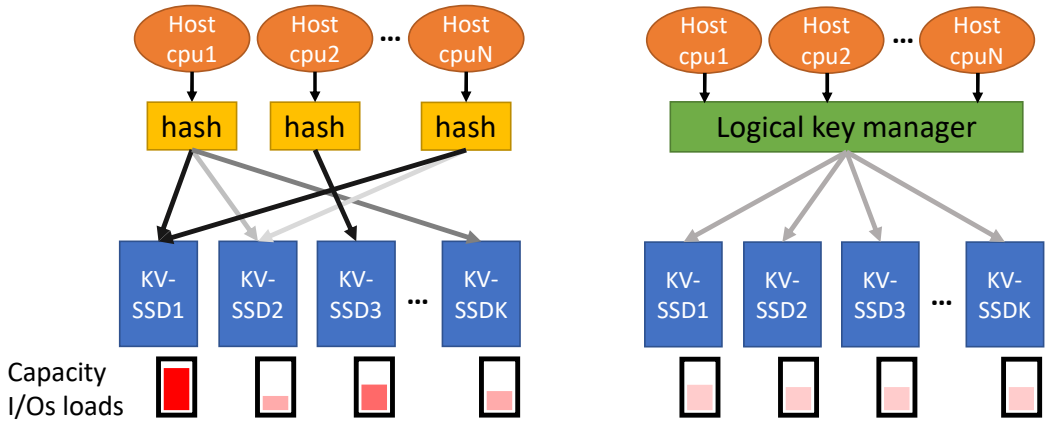
Figure 5.1: Moving operating point of KVSSD through packing enabled by logical key remmaping. (By reducing the total number of keys managed by the device from B to A, we can significantly improve the operating performance of the device.)

By selectively packing some records (through logical key remapping) and leaving some records un-packed, we can effectively implement multi-tiering on a single key-value storage device. Such multi-tiering can be useful if the workloads have pronounced hot and cold data characteristics. For instance, we can pack the cold data to improve the overall put performance while maintaining overall high get throughput since the hot data can be read without key translation overhead. More importantly, such packing will reduce the total number of physical keys inside the device and further improves the overall operating (put/get) performance. Figure 5.1 illustrates how we can improve the overall device operating performance by moving from operating point B (around 800

million records) to operating point A (around 280 million records) through packing enabled by logical key remapping.

5.2.2 Key-value storage device array management

Besides performance multi-tiering, another powerful feature enabled by logical key remapping for key-value storage device is key-value storage device array management. Unlike traditional block-based storage device array [46, 131, 132] which usually uses round-robin method to map logical block address (LBA) to device in the array, key-value interfaced storage devices don't have a continuous address space. Current works [38, 39] employ simple hashing to mange key-value device array. The basic idea is to hash on the keys of a given record to determine the device in the array to perform key-value operations. The advantage of this approach is that it is simple to implement and incurs negligible management overhead. However, it lacks flexibility to manage various loads to each device.



(a) Simple hashing approach

(b) Logical key manager

Figure 5.2: Key-value device array management. (The darker shade arrows and lighter shade arrows denote different capacity and I/Os load respectively.)

Figure 5.2 illustrates a typical key-value device array system. From top to bottom, key value store applications running on multiple host CPUs distribute key-value I/O requests to the key-

value storage devices. Figure 5.2 (a) demonstrates using simple hashing approach to distribute key-value requests to the device array. Due to the imperfect nature of hashing, the requests may not be distributed evenly to the devices. Such imbalance can arise for different reasons. For example, KV-SSD1 may be allocated with larger value size records, causing higher capacity utilization. Another example might be that many small records are stored on KV-SSD1 resulting in disproportion of I/O requests. Figure 5.2 (b) shows our proposed key-value device array management with logical key remapping techniques. By replacing the simple hashing with logical key manager to translate logical/application keys to physical/device keys, we can selectively implement packing, dynamic load balancing to different devices. Such logical key manager greatly improves management flexibility on key-value device arrays.

5.3 Preliminary evaluation

5.3.1 Experimental setup

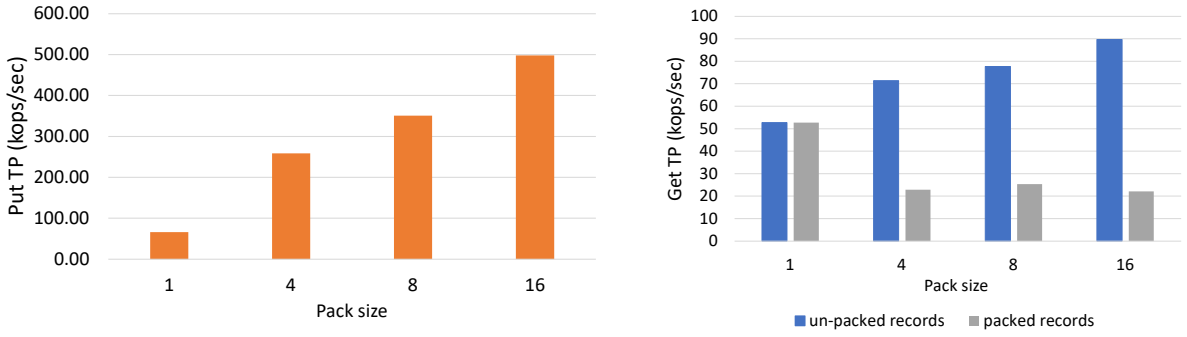
To evaluate the idea of KVSSD multi-tiering using the logical key remapping technique, we employed micro-benchmarks and evaluated on a real KVSSD system. Basic system configuration is described as follows:

- CPU: Intel Silver 4216 @ 2.1GHz, 16 cores
- Memory: 96GB DDR4 @ 2133MHz
- KVSSD: Sumsung PM983 3.84TB
- Operating System: Ubuntu 16.04 (linux kernel version 4.15)

5.3.2 Performance multi-tiering with different packing size

To evaluate the put and get performance implications from logical key remapping for key-value devices, we design experiments to perform put and get operations based on different physical size records. In each of these experiments, we first put 1 billion records on one KVSSD. We pack 900 million records with different packing size from 1 to 16 as shown in Figure 5.3 (1 stands for no

packing at all). The remaining 100 million records are un-packed and can be accessed through logical key directly from the key-value device. The key and value size for each KV record is 16 bytes and 1000 bytes respectively.



(a) Put tail latency

(b) Get tail latency

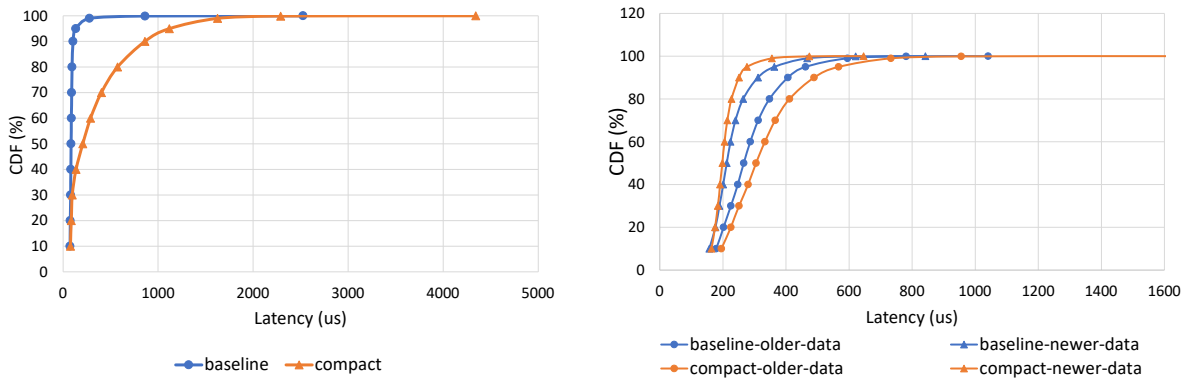
Figure 5.3: Throughput performance for multi-tiering KVSSD with different pack size

Figure 5.3 (a) illustrates the put throughput over different packing ratios. As we increased the packing ratio (from 4 to 16) we can obtain almost linear improvement in the overall put throughput. The reason that put throughput scaling is not linear with the packing ratio scaling (packing ratio from 4 to 16, the put performance increases by around 2x) comes from two reasons. First, there is a noticeable overhead of metadata write for logical key to physical key remapping. Second, larger I/O size (from packing multiple logical records into a single physical record) will reduce the device put throughput (IOPS) as shown in Figure 1.2.

Figure 5.3 (b) shows the get performance for different packing ratios. Due to the extra layer of logical key to physical key translation, packed records obtain lower performance. However, the get performance of unpacked records is higher since the device performance improves at a lower number of stored records. CHECK THIS.

5.3.3 Compaction for timeseries workloads

In this section, we exploit the performance of multi-tiering technique based on selective packing of records in timeseries workload [93, 2, 90]. Timeseries workloads usually embody distinct temporal locality [124], i.e. latest data are more likely to be queried compared to older data. For example, consider a timeseries workload that collects performance metrics of servers in a data center. The data for the current week is more likely to be queried than the data a year ago. Thus, we can automatically pack (we refer it to compaction) the older data to ensure high get, range query performance for the recent data, while reducing the overall number of physical keys managed in the device.



(a) Put tail latency

(b) Get tail latency

Figure 5.4: Tail latency performance for timeseries workloads with compaction

In these experiments, we issue steady put and get requests for the timeseries workloads (500 million 1000B records in total). We automatically compact older records (after exceeding the recent 100 million records) in background. Figure 5.4 demonstrates the tail latency of put and get performance with our compaction technique, compared with the baseline (without any compaction). The put latency of multi-tier approach got worse compared to baseline due to background compaction. However, the get tail latency of recent data (newer data) improved compared to baseline. The 99.9% tail latency improved by $\sim$30%. Figure 5.5 shows the get throughput performance.

Compared to baseline, multi-tier approach also improves ∼30% throughput performance for newer data.

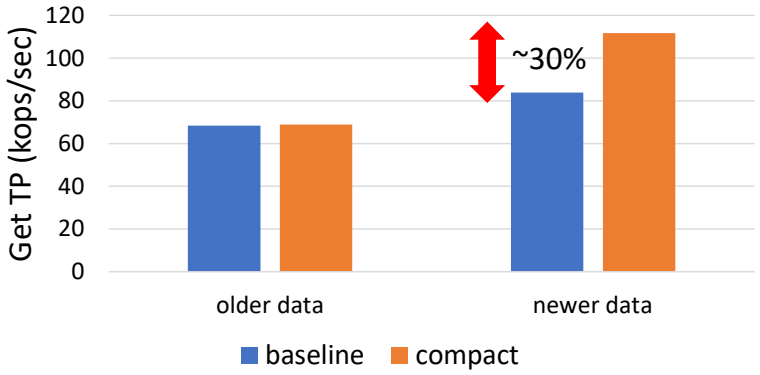


Figure 5.5: Get throughput performance for timeseries workloads with compaction

5.4 Summary

To summarize, in this chapter, we explore the opportunity to leverage the logical key remapping technique on a key-value storage device to implement performance multi-tiering which can be used potentially for hot cold data aware workloads such as timeseries data workloads or quality of service (QoS) required situations. In addition, we also discussed the possibility of managing multiple key-value storage devices (key-value storage device arrays) to implement capacity and I/O load balancing through logical key remapping technique.

6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Due to the explosive growth of data especially unstructured data, conventional datastores such as file systems and relational databases are moving towards highly scalable key-value stores. However, the conventional software system stack which is designed and optimized around traditional block based storage medium are not adequate to support the contemporary storage requirements. New storage devices, system software are urgently needed in the big data era.

In this dissertation, we focused on supporting key value applications on top of emerging key value devices. We proposed the core idea of logical key management layer that allowed remapping of application or logical keys into physical keys on the devices. Such a logcal key management layer enables efficient management of name spaces across multiple devices, parity protection, load balncing and other applications across multiple devices. In addition, we showed that the key remapping allows efficient range query support and multi-tiering on a single key-value device.

We presented two intelligent software techniques for supporting storage efficient data redundancy and supporting range query capability on key-value devices. First, we introduced KVRAID, a storage efficient, high performance, write efficient, update friendly erasure coding management scheme on key-value SSDs. The core innovation of KVRAID is to use logical to physical key conversion to efficiently pack similar size KV records and dynamically manage the membership of erasure coding groups. Second, we proposed KVRangeDB, an ordered log structure tree based key index that supports range queries on key-value storage devices. KVRangeDB also leverages logical key remapping technique to pack smaller application records into a larger physical record on the device to provide higher write throughput and efficient get and range queries. We also discussed hardware and architectural techniques for accelerating the recovery bandwidth efficient erasure codes. We presented a generic hardware acceleration architecture for emerging Minimum Storage Regenerating (MSR) codes which maximizes the computation parallelism and minimizes

the data movement between off-chip DRAM and the on-chip SRAM buffers. Finally, we briefly discussed other potential use cases for leveraging logical key remapping technique for key-value storage devices. We discussed the performance enhancements from multi-tiering a single KVSSD and I/O and capacity and load balancing across key-value device arrays.

Our proposed techniques and solutions can be easily employed in other storage system domains. For instance, the software based KVRAID design can be implemented on hardware key-value storage controllers with the same design principles. Our proposed ideas for KVRangeDB can be potentially merged into the key-value device flash translation layer (FTL) to enhance the key value device interfaces. The generic hardware architecture for Minimum Storage Regenerating (MSR) codes can be applied in other hardware platforms, for example, ASICs.

6.2 Future work

Throughout our work, we found several research opportunities with the emerging key-value storage devices that can help deliver better system performance for handling unstructured data in the big data era. For future works, we are mainly looking into two aspects of research domains as follows.

First, although range query as we discussed in Chapter 3 is powerful infrastructure for the data analytics applications, we found more complicated analytic demands such as filtering, aggregation, statistical analysis, etc. for emerging data-centric applications, including timeseries analysis, fraud detection, forecasting. Instead of conventional software query processing which introduces huge amount of CPU cycles and data movements between devices and host CPUs, we advocate offloading those complex data analytics query processing close to hardware (accelerators or smart storage devices) and exposing high level key-value based query interface to the applications.

Second, to better accommodate data driven applications such as artificial intelligence (AI) and machine learning (ML) applications which mainly dealing with unstructured data, we hope to re-examine the software and hardware stack for storage systems to find better storage and data processing/analytics interfaces for future storage devices and system software design.

REFERENCES

[1] M. Qin, N. Reddy, P. V. Gratz, R. Pitchumani, and Y. S. Ki, "Kvraid: High performance, write efficient, update friendly erasure coding scheme for kv-ssds," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR '21, (New York, NY, USA), ACM, 2021.

[2] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), p. 323–336, Association for Computing Machinery, 2018.

[3] M. Qin, J. H. Lee, R. Pitchumani, Y. S. Ki, N. Reddy, and P. V. Gratz, "A generic fpga accelerator for minimum storage regenerating codes," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 271–276, IEEE, 2020.

[4] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel SIMD instructions," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, (San Jose, CA), pp. 298–306, USENIX Association, 2013.

[5] A. Potnis, "White paper: Illuminating insight for unstructured data at scale," tech. rep., International Data Corporation (IDC), Aug 2018.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, (Seattle, WA), USENIX Association, 2006.

[8] Apache, "Hbase," 5 2013. https://hbase.apache.org/.

[9] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.

[10] B. Carrier, *File system forensic analysis*. Addison-Wesley Professional, 2005.

[11] S. D. Pate, "Unix filesystems: Evolution," *Design, and Implementation*, p. 19, 2003.

[12] S. Tripathy, "Windows nt file system internals: A developer's guide," *ITNOW*, vol. 40, no. 5, pp. 31–31, 1998.

[13] S. Sumathi and S. Esakkirajan, *Fundamentals of relational database management systems*, vol. 47. Springer, 2007.

[14] H. Darwen, *An introduction to relational database theory*. Bookboon, 2009.

[15] P. Beynon-Davies, *Database systems*. Springer, 2004.

[16] T. M. Connolly and C. E. Begg, *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.

[17] J. Dean and S. Ghemawat, "Leveldb: Google's fast key value store library," *Github release 1.2*, 2017.

[18] Facebook, "Rocksdb," 10 2015. https://rocksdb.org/.

[19] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[20] K. G. Ashar, "Magnetic disk drive technology: heads, media, channel, interfaces, and integration," *Thin Film Heads*, 1994.

[21] E. Grochowski, *The Continuing Evolution of Magnetic Hard Disk Drives*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

[22] A. Al Mamun, G. Guo, and C. Bi, *Hard disk drive: mechatronics and control*. CRC press, 2017.

[23] R. Micheloni, A. Marelli, and K. Eshghi, *Inside solid state drives (SSDs)*. Springer, 2013.

[24] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 181–192, 2009.

[25] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, (Santa Clara, CA), pp. 133–148, USENIX Association, Feb. 2016.

[26] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 497–514, ACM, 2017.

[27] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, ftl-aware key-value store," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, (Santa Clara, CA), pp. 207–219, USENIX Association, July 2015.

[28] Y. Kang, R. Pitchumani, P. Mishra, Y.-s. Kee, F. Londono, S. Oh, J. Lee, and D. D. G. Lee, "Towards building a high-performance, scale-in key-value storage system," in *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, (New York, NY, USA), pp. 144–154, ACM, 2019.

[29] J. Im, J. Bae, C. Chung, Arvind, and S. Lee, "Pink: High-speed in-storage key-value store with bounded tails," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 173–187, USENIX Association, July 2020.

[30] Y. Jin, H. Tseng, Y. Papakonstantinou, and S. Swanson, "Kaml: A flexible, high-performance key-value ssd," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 373–384, Feb 2017.

[31] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, (Renton, WA), pp. 405–417, USENIX Association, Apr. 2018.

[32] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley db," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, (Berkeley, CA, USA), pp. 43–43, USENIX Association, 1999.

[33] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 1–13, ACM, 2011.

[34] B. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, pp. 1414–1425, Sept. 2010.

[35] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti, "Flashield: a hybrid key-value cache that controls flash write amplification," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), pp. 65–78, USENIX Association, Feb. 2019.

[36] 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 *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 16:1–16:14, ACM, 2014.

[37] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu, "Lessons and actions: What we learned from 10k ssd-related storage system failures," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 961–976, USENIX Association, July 2019.

[38] R. Pitchumani and Y.-S. Kee, "Hybrid data reliability for emerging key-value storage devices," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, (Santa Clara, CA), pp. 309–322, USENIX Association, Feb. 2020.

[39] U. Maheshwari, "Stripefinder: Erasure coding of small objects over key-value storage devices (an uphill battle)," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, USENIX Association, July 2020.

[40] J. S. Plank, "Erasure codes for storage systems: A brief primer," *; login:: the magazine of USENIX & SAGE*, vol. 38, no. 6, pp. 44–50, 2013.

[41] Z. Zhang, A. Wang, K. Zheng, U. M. G, and Vinayakumar, "Introduction to hdfs erasure coding in apache hadoop," 2018. `https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop`.

[42] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, (Boston, MA), pp. 15–26, USENIX, 2012.

[43] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 49–60, USENIX, 2013.

[44] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Peer-to-Peer Systems* (P. Druschel, F. Kaashoek, and A. Rowstron, eds.), (Berlin, Heidelberg), pp. 328–337, Springer Berlin Heidelberg, 2002.

[45] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[46] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, (New York, NY, USA), pp. 109–116, ACM, 1988.

[47] "Intel® optane™ dc persistent memory." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[48] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[49] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, (Santa Clara, CA), pp. 209–223, USENIX Association, Feb. 2020.

[50] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.

[51] J. Jeong, S. S. Hahn, S. Lee, and J. Kim, "Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, (Santa Clara, CA), pp. 61–74, USENIX, 2014.

[52] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, (San Jose, CA), pp. 257–270, USENIX, 2013.

[53] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, (New York, NY, USA), Association for Computing Machinery, 2009.

[54] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2004.

[55] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.

[56] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 307–320, USENIX Association, 2006.

[57] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 143–157, ACM, 2011.

[58] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Comput. Surv.*, vol. 26, pp. 145–185, June 1994.

[59] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's warm BLOB storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 383–398, USENIX Association, Oct. 2014.

[60] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, (San Jose, CA), USENIX, 2013.

[61] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction," *IEEE Transactions on Information Theory*, vol. 57, pp. 5227–5239, Aug 2011.

[62] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi, "Clay codes: Moulding MDS codes to yield an MSR code," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, (Oakland, CA), pp. 139–154, USENIX Association, Feb. 2018.

[63] A. Kadav, M. Balakrishnan, V. Prabhakaran, and D. Malkhi, "Differential raid: Rethinking raid for ssd reliability," in *HotStorage 2009: 1st Workshop on Hot Topics in Storage and File Systems*, Association for Computing Machinery, Inc., October 2009. (best paper award.).

[64] Y. Li, H. H. W. Chan, P. P. C. Lee, and Y. Xu, "Elastic parity logging for ssd raid arrays," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 49–60, June 2016.

[65] J. Kim, K. Lim, Y. Jung, S. Lee, C. Min, and S. H. Noh, "Alleviating garbage collection interference through spatial separation in all flash arrays," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 799–812, USENIX Association, July 2019.

[66] J. Kim, J. Lee, J. Choi, D. Lee, and S. H. Noh, "Improving ssd reliability with raid via elastic striping and anywhere parity," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, June 2013.

[67] H. Zhang, M. Dong, and H. Chen, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, (Santa Clara, CA), pp. 167–180, USENIX Association, Feb. 2016.

[68] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding," in *12th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 16), (Savannah, GA), pp. 401–417, USENIX Association, Nov. 2016.

[69] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, p. 18–es, July 2007.

[70] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, 2002.

[71] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, (New York, NY, USA), p. 229–240, Association for Computing Machinery, 2009.

[72] M. Bjørling, J. Gonzalez, and P. Bonnet, "Lightnvm: The linux open-channel SSD subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, (Santa Clara, CA), pp. 359–374, USENIX Association, Feb. 2017.

[73] "Nvm express," 2020. `https://nvmexpress.org/`.

[74] M. Bjørling, "Zone append: A new way of writing to zoned storage," in *Proceedings of the 2020 Linux Storage and Filesystems Conference (Vault'20)*, (Santa Clara, CA), USENIX Association, Feb. 2020.

[75] "Key value storage api specification - snia," 2020. `https://www.snia.org/keyvalue`.

[76] zhichao Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, (Santa Clara, CA), pp. 209–223, USENIX Association, Feb. 2020.

[77] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquín, and D. Kossmann, "Fast scans on key-value stores," *Proc. VLDB Endow.*, vol. 10, p. 1526–1537, Aug. 2017.

[78] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.

[79] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008.

[80] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo, "Scaling embedded in-situ indexing with deltafs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, IEEE Press, 2018.

[81] C. Chrysafis, B. Collins, S. Dugas, J. Dunkelberger, M. Ehsan, S. Gray, A. Grieser, O. Herrnstadt, K. Lev-Ari, T. Lin, M. McMahon, N. Schiefer, and A. Shraer, "Foundationdb record layer: A multi-tenant structured datastore," in *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, (New York, NY, USA), p. 1787–1802, Association for Computing Machinery, 2019.

[82] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, p. 422–426, July 1970.

[83] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos, "Rosetta: A robust space-time optimized range filter for key-value stores," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, (New York, NY, USA), p. 2071–2086, Association for Computing Machinery, 2020.

[84] K. Ren and G. Gibson, "TABLEFS: Enhancing metadata efficiency in the local file system," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, (San Jose, CA), pp. 145–

156, USENIX Association, June 2013.

[85] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 191–208, USENIX Association, Nov. 2020.

[86] C. F. Systems, "Fast and efficient filesystem metadata through lsm-trees.," 2013. `https://github.com/pdlfs/tablefs/`.

[87] "The influxdb storage engine and the time-structured merge tree (tsm)," 2020. `https://docs.influxdata.com/influxdb/v1.8/concepts/storage_engine/`.

[88] S. Rhea, E. Wang, E. Wong, E. Atkins, and N. Storer, "Littletable: A time-series database and its uses," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), p. 125–138, Association for Computing Machinery, 2017.

[89] "Kairosdb," 2020. `https://kairosdb.github.io/`.

[90] "Succinct range filter (surf)," 2018. `https://github.com/efficient/SuRF`.

[91] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proc. VLDB Endow.*, vol. 8, p. 1816–1827, Aug. 2015.

[92] Huamin Chen, Jian Li, and P. Mohapatra, "Race: time series compression with rate adaptivity and error bound for sensor networks," in *2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE Cat. No.04EX975)*, pp. 124–133, 2004.

[93] T. W. Wlodarczyk, "Overview of time series storage and processing in a cloud environment," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 625–628, 2012.

[94] "Nvme key value (nvme-kv) command set," 2020. `https://nvmexpress.org/faq-items/what-is-key-value/`.

[95] S.-H. Kim, J. Kim, K. Jeong, and J.-S. Kim, "Transaction support using compound commands in key-value ssds," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, (Renton, WA), USENIX Association, July 2019.

[96] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi, "Slm-db: Single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, (Boston, MA), pp. 191–205, USENIX Association, Feb. 2019.

[97] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 1077–1091, Association for Computing Machinery, 2020.

[98] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 137–152, ACM, October 2017.

[99] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016.

[100] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pp. 13–24, IEEE Press, June 2014. Selected as an IEEE Micro TopPick.

[101] R. Hai, S. Geisler, and C. Quix, "Constance: An intelligent data lake system," in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, (New York, NY, USA), p. 2097–2100, Association for Computing Machinery, 2016.

[102] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena, "Data lake management: Challenges and opportunities," *Proc. VLDB Endow.*, vol. 12, p. 1986–1989, Aug. 2019.

[103] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan, "Azure data lake store: A hyperscale distributed file service for big data analytics," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), p. 51–63, Association for Computing Machinery, 2017.

[104] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.

[105] J. S. Plank, "The raid-6 liberation codes," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, (Berkeley, CA, USA), pp. 7:1–7:14, USENIX Association, 2008.

[106] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 307–320, USENIX Association, 2006.

[107] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, May 2010.

[108] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, pp. 4539–4551, Sep. 2010.

[109] I. Tamo, Z. Wang, and J. Bruck, "Zigzag codes: Mds array codes with optimal rebuilding," *IEEE Transactions on Information Theory*, vol. 59, pp. 1597–1616, March 2013.

[110] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi, "Clay codes: Moulding MDS codes to yield an MSR code," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, (Oakland, CA), pp. 139–154, USENIX Association, 2018.

[111] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *7th USENIX Conference on File and Storage Technologies (FAST 09)*, (San Francisco, CA), USENIX Association, 2009.

[112] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads," in *10th USENIX Conference on File and Storage Technologies (FAST 12)*, (San Jose, CA), USENIX Association, 2012.

[113] T. Zhou and C. Tian, "Fast erasure coding for data storage: A comprehensive study of the acceleration techniques," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, (Boston, MA), pp. 317–329, USENIX Association, 2019.

[114] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *2008 Symposium on Application Specific Processors*, pp. 101–107, June 2008.

[115] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, (New York, NY, USA), pp. 202–215, ACM, 2016.

[116] S. Bates, "Donard: Nvm express for peer-2-peer between ssds and other pcie devices," in *Storage Developer Conference, SNIA Santa Clara*, 2015.

[117] M. Ye and A. Barg, "Explicit constructions of optimal-access mds codes with nearly optimal sub-packetization," *IEEE Transactions on Information Theory*, vol. 63, pp. 6307–6317, Oct 2017.

[118] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?," in *5th USENIX Conference on File and Storage Technologies (FAST 07)*, (San Jose, CA), USENIX Association, 2007.

[119] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, (Boston, MA), pp. 15–26, USENIX, 2012.

[120] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," *Proc. VLDB Endow.*, vol. 6, pp. 325–336, Mar. 2013.

[121] J. S. Liptay, "Structural aspects of the system/360 model 85, ii: The cache," *IBM Systems Journal*, vol. 7, no. 1, pp. 15–21, 1968.

[122] G. H. Loh, "3d-stacked memory architectures for multi-core processors," *ACM SIGARCH computer architecture news*, vol. 36, no. 3, pp. 453–464, 2008.

[123] F. Wen, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Hardware memory management for future mobile hybrid memory systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3627–3637, 2020.

[124] N. Agrawal and A. Vulimiri, "Low-latency analytics on colossal data streams with summarystore," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), p. 647–664, Association for Computing Machinery, 2017.

[125] P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour, *Service level agreements for cloud computing*. Springer Science & Business Media, 2011.

[126] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[127] H. Mauelshagen, "Logical volume manager (lvm2)," *Red Hat Magazine*, 2004.

[128] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh, "The logical disk: A new approach to improving file systems," *SIGOPS Oper. Syst. Rev.*, vol. 27, p. 15–28, Dec. 1993.

[129] Microsoft, "How to use the disk management snap-in to manage basic and dynamic disks," 2020. `https://docs.microsoft.com/en-us/troubleshoot/windows-server/backup-and-storage/disk-management-snap-in-basic-dynamic-disks`.

[130] O. Rodeh and A. Teperman, "zfs-a scalable distributed file system using object disks," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.*, pp. 207–218, IEEE, 2003.

[131] S. Savage and J. Wilkes, "Afraid: A frequently redundant array of independent disks," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, (USA), p. 3, USENIX Association, 1996.

[132] D. Stodolsky, G. Gibson, and M. Holland, "Parity logging overcoming the small write problem in redundant disk arrays," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), p. 64–75, Association for Computing Machinery, 1993.