KVAccel: A Novel Write Accelerator for LSM-Tree-Based KV Stores with Host-SSD Collaboration

Kihwan Kim[1,*], Hyunsun Chung[1,*], Seonghoon Ahn[1,*], Junhyeok Park[1], Safdar Jamil[1]
Hongsu Byun[1], Myungcheol Lee[2], Jinchun Choi[2], Youngjae Kim[1,†]

[1]Dept. of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea
[2]ETRI, Daejeon, Republic of Korea

Abstract—Log-Structured Merge (LSM) tree-based Key-Value Stores (KVSs) are widely adopted for their high performance in write-intensive environments, but they often face performance degradation due to write stalls during compaction. Prior solutions, such as regulating I/O traffic or using multiple compaction threads, can cause unexpected drops in throughput or increase host CPU usage, while hardware-based approaches using FPGA, GPU, and DPU aimed at reducing compaction duration introduce additional hardware costs. In this study, we propose KVAccel, a novel hardware-software co-design framework that bypasses write stalls by leveraging a dual-interface SSD. KVAccel allocates logical NAND flash space to support both block and key-value interfaces, using the key-value interface as a temporary write buffer during write stalls. This strategy significantly reduces write stalls, optimizes resource usage, and ensures consistency between the host and device by implementing an in-device LSM-based write buffer with an iterator-based range scan mechanism. Our extensive evaluation shows that for write-intensive workloads, KVAccel outperforms ADOC by up to 17% in terms of throughput and performance-to-CPU-utilization efficiency. For mixed read-write workloads, both demonstrate comparable performance.

Index Terms—Key-Value Store, Log-Structured Merge Tree, Solid State Drive, Write Stall Mitigation

I. INTRODUCTION

Log-Structured Merge (LSM) tree-based Key-Value Store (KVS) systems, such as RocksDB [1] and LevelDB [2], are commonly used in write-intensive applications due to their ability to handle high-throughput writes efficiently. However, LSM-based KVSs (LSM-KVSs) often experience performance degradation due to write stalls that occur during compaction [3]–[8]. These write stalls block incoming write operations, resulting in a significant reduction in throughput and an increase in tail latency, which undermines system reliability in time-sensitive workloads.

To alleviate write stalls, many software-based solutions have been explored and deployed. RocksDB [1], one of the most widely used LSM-KVS, implements a mechanism known as *slowdown* [9]. This slowdown mechanism anticipates potential write stalls and proactively reduces the write pressure on the LSM-KVS. While slowdowns can prevent write stalls, it may unnecessarily decrease the throughput of RocksDB by limiting the write pressure directed to the LSM-KVS. Additionally, the state-of-the-art solution ADOC [5] mitigates write stalls by dynamically increasing batch sizes and the number of

compaction threads during a write slowdown, thereby reducing compaction duration. However, ADOC increases host CPU utilization by employing multiple compaction threads.

Alternatively, hardware-based solutions have been investigated. Persistent Memory (PM)-based designs [6], [10], [11] buffer writes in PM before flushing them to the LSM-tree, while FPGA-based accelerators [12]–[14], GPU [15]–[17], and DPU [18]–[20] speed up merge sort to reduce compaction time. Key-Value SSD (KV-SSD) architectures [21]–[25] handle key-value operations directly within storage devices, bypassing the OS and file system overheads. Although these approaches enhance performance, they require additional hardware (e.g., PM, FPGA, GPU, DPU), raising costs and complexity.

The aforementioned software solutions suffer from unnecessary performance degradation due to inaccurate predictions or increased host CPU usage, while hardware solutions require additional hardware, raising costs. In this study, we propose a groundbreaking approach that avoids write stalls without compromising KVS performance, minimizes host CPU utilization, and requires no additional hardware costs. Our method represents a new paradigm that is fundamentally different from existing approaches, by actively leveraging idle resources in existing storage devices to avoid write stalls while minimizing host CPU involvement.

In this paper, we present KVAccel, a novel hybrid hardware-software co-design framework that leverages a new dual-interface SSD architecture to mitigate write stalls and optimize the utilization of storage bandwidth. KVAccel is built on the observation that during host-side write stalls, the underlying storage device's available I/O bandwidth remains underutilized, despite its potential to handle additional I/O operations. KVAccel then incorporates a dynamic I/O redirection mechanism that monitors the status of host-side LSM-KVS and, upon detecting a write stall, shifts writes from the LSM-KVS to the device-side key-value write buffer.

KVAccel presents a disaggregation of the SSD's logical NAND flash address space into two regions: one for the traditional block interface, which is managed by the host-side LSM-KVS, and another for the key-value interface inspired by the KV-SSD, which serves as a temporary write buffer to serve pending write requests by bypassing the traditional LSM-based data path during stalls.

To maintain consistency between the main LSM on the host and the write buffer on the device, KVAccel introduces

---

a range scan-based rollback mechanism. This mechanism structures the device-side write buffer as a separate LSM from the host-side main LSM and employs an iterator-based range scan over the buffer, enabling fast scan of buffered key-value pairs back to the host for merging. KVACCEL then merges them into the main LSM, maintaining the properties of the LSM and ensuring data consistency between the two interfaces.

KVACCEL offers detector, I/O redirection, and rollback modules on top of RocksDB [1]. The dual-interface SSD was implemented using the Cosmos+ OpenSSD platform [26], an FPGA-based NVMe SSD development board. RocksDB operates on the block interface provided by a single OpenSSD, while the key-value interface of the same device handles the redirected key-value pairs.

The key contributions of this paper are as follows:

- We identify a critical opportunity to mitigate the fundamental issue of write stalls in LSM-KVS by leveraging the under-utilized storage bandwidth during these stalls, transforming an inherent inefficiency into a performance optimization.
- We propose a hybrid SSD architecture that integrates a new key-value interface alongside the traditional block interface within a singular device, allowing us to address write stalls without significantly modifying the existing LSM-KVS or deploying additional hardware in the system.
- We develop efficient dynamic I/O redirection and rollback mechanisms to seamlessly manage data flow between the host-side LSM and device-side key-value interface, ensuring consistency and high performance.
- Our approach demonstrates that by introducing an additional storage interface, separate from the traditional block interface, on a singular storage device in the system, we can provide an architecturally beneficial solution to address the inherent limitations of LSM-KVS, which required intentionally lowering the quality of write service to mitigate write stalls.

Our extensive evaluation using *db_bench* [27] demonstrates that KVACCEL avoids the write stall penalty and achieves up to a 17% increase in throughput compared to the state-of-the-art solution by harnessing underutilized PCIe bandwidth during write stall periods, all while maintaining read performance. These results show that KVACCEL not only alleviates the performance bottlenecks of existing LSM-KVS systems but also introduces a novel, architecturally superior, cost-effective solution for optimizing write-intensive workloads in modern storage environments.

II. BACKGROUND AND RELATED WORK

This section reviews the compaction process in LSM-tree structures, where write stalls occur, and examines related research aimed at mitigating these stalls.

A. Log-Structured Merge Tree and Write Stall Issue

The Log-Structured Merge (LSM) tree [28] is a write-optimized data structure widely adopted in various NoSQL databases including LevelDB [2], RocksDB [1], and Cassandra [29]. The LSM-tree organizes data into memory and
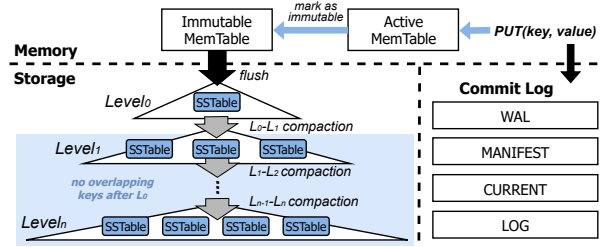

Fig. 1: An architecture of LSM-tree.

disk-based components with hierarchical levels increasing in size, as shown in Figure 1. The memory components include active MemTable (MT) which absorbs the incoming write requests from application. Once MT reaches a size threshold, a new active MT is allocated and old MT is converted into an Immutable MemTable (IMT). The flush operation picks the IMT and convert it into Sorted String Table (SST) file and write to storage device. The SSTs are organized in ascending levels, with each level having a size threshold. When a level reaches the size threshold, SSTs of current victim level n goes through a merge-sort operation, known as compaction, with SSTs of level $n + 1$. This process ensures key-value pairs within each SST to be sorted and unique.

Write Stall Problem: Despite being write-optimized, LSM-KVSs suffer from the write stall problem. We define the write stall problem as blocking of incoming write requests by the internals of LSM-tree. SILK [3] and ADOC [5] categorize these write stalls into three different events. ❶ Flush-based write stalls: when the flush operation is not able to keep up with the rate of incoming write requests resulting in exhaustion of memory. ❷ Level 0 to Level 1 (L_0 to L_1) compaction-based write stalls: the SSTs in level 0 can hold overlapping key range, which necessitate the compaction operation to be serialized between L_0 to L_1 compaction. This serialization of L_0 to L_1 compaction can lead to blocking of flush operation when L_0 reaches its size threshold, resulting in L_0 to L_1 compaction-based write stall event. ❸ Pending compaction bytes-based write stall: when the lower levels of LSM-KVS delays the compaction operation leading to high space amplification, resulting in blocking of incoming write requests.

B. Existing Optimizations for Addressing Write Stall Issue

To optimize LSM-KVS, there have been extensive research conducted by academia and industry which can be classified into two categories: (i) software-level and (ii) hardware-level.

Software-Level Optimization: SILK [3] introduces an I/O scheduler that mitigates write stalls by delaying flush and compaction operations to low-load periods, prioritizing flushes and lower-level compactions, and preempting compactions. Despite these strategies, SILK offers minimal performance improvement and exhibits ordinary tail latency under sustained write-intensive and long peak workloads. RocksDB [1] indeed employs a slowdown mechanism [9] that predicts potential write stalls and intentionally lowers the write throughput to

prevent sudden performance drops, but this comes at the cost of increased latency and degraded service quality during heavy workloads. Blsm [30] proposes a merge scheduler to coordinate compactions across multiple levels, but the L_0 to L_1 compaction still severely stalls foreground requests. The state-of-the-art solution, ADOC [5], also reduces and restores the write ratio as needed, and introduces a new mechanism to dynamically adjust write buffer size and background threads during write-intensive workloads, demonstrating superior write stall mitigation compared to existing solutions.

Despite these efforts, *existing approaches aim to minimize compaction time to mitigate the write stall issue, but they ultimately rely on intentionally lowering the write request rate.* This trade-off negatively impacts service quality for users, highlighting the inherent limitation of ensuring uninterrupted write operations at the cost of degraded performance.

Hardware-Level Optimization: To eliminate the high storage stack overhead during key-value writes and compaction, some studies have implemented LSM-KVS directly on SSDs, referred to as Key-Value SSDs (KV-SSDs) [21]–[25]. iLSM [22] bypasses the file system and block layer within the kernel, thereby improving the I/O latency and throughput of key-value clients. PinK [23] proposed a resource-efficient LSM-KVS within the KV-SSD and demonstrated that KV-SSDs can reduce CPU and DRAM resource usage on both the host and device side. In contrast, there are studies that optimize LSM-KVS by leveraging Persistent Memory (PM). MatrixKV [6] observes the shortcoming of the original SST format and points to slow L_0 to L_1 compaction as the root cause of write stalls when deploying PMs. It redesigns the format of SST for PM and proposes a new compaction scheme between the first two levels, which they call column-compaction. Zhang et. al. [13] proposed an FPGA-based acceleration engine for LSM-KVS to speed up the compaction process via hardware-software collaboration, improving throughput and averting resource contention. However, these studies face significant limitations in terms of applicability, as they rely on new devices that *either require completely bypassing host-side LSM-KVS stacks or adding new hardware components.*

III. PROBLEM DEFINITION

In this section, we point out that both industry standard and state-of-the-art software-level solutions both rely on the write slowdown, an inefficient write stall prevention method. Furthermore, we highlight that during write stalls in LSM-KVS, the storage device is underutilized, even though it still has the capacity to process I/O requests.

A. Slowdowns: The Inefficient Write Stall Solution

To prevent write stalls, the basic and most primitive solution is to slow down the writes itself before a write stall occurs. This is done by putting the write thread to sleep for a short duration of time, such as 1 ms [9]. Industry standard LSM-KVS such as RocksDB make liberal use of slowdowns during heavy write workloads to prevent write stalls. Meanwhile, ADOC [5], the state of the art solution, still falls back to slowdowns as
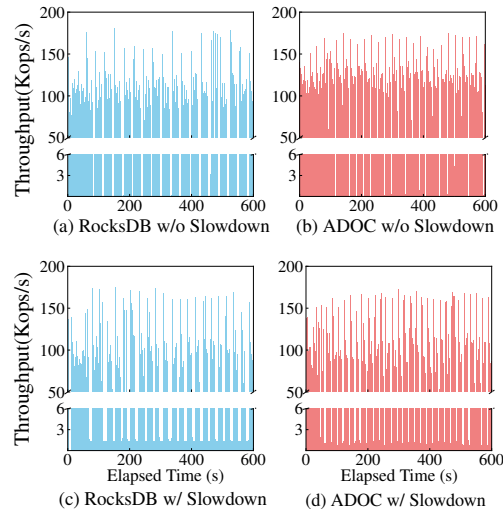

Fig. 2: Per-second throughput time-series for RocksDB and ADOC, based on write slowdown usage.

a last resort despite software optimizations such as dynamic allocation of compaction threads and batch size.

To measure the effectiveness and frequency of write slowdowns, we used RocksDB's benchmark tool, *db_bench* [27], and executed the `fillrandom` workload for 600 seconds on RocksDB and ADOC. The experiments were conducted using an OpenSSD-based SSD prototype mounted with a traditional block based interface with the ext4 file-system. The SSD supports a peak bandwidth of approximately 630 MB/s, and is connected to the host via a PCIe Gen2.0 x8 interface, yielding a theoretical maximum PCIe bandwidth of 4 GB/s. Details about our experimental environment are provided in Section VI-A.

Figure 2 (a)-(d) show RocksDB and ADOC's time-series throughput. Here, we present two variants of RocksDB and ADOC, first where the slowdown feature of RocksDB and ADOC is disabled while in the second, the slowdown feature is enabled, respectively. Comparing Figures 2 (a) and (c) with (b) and (d), respectively, we observe that when the slowdown feature is enabled for both RocksDB and ADOC, the issue of write throughput dropping to zero—i.e., write service halting momentarily—disappears. Instead, although the throughput is slightly lower, it remains stable at a base level, providing consistent service at up to 2 Kops/s. This demonstrates that the slowdown feature effectively mitigates write stalls, ensuring stable and uninterrupted service. However, it also highlights that the extent of throughput mitigation achieved through the slowdown mechanism is not particularly significant.

Surprisingly, looking at Figure 3 reveals that these slowdowns actively harm performance in both throughput and P99 latency. While slowdown is in effect, the overall throughput of RocksDB and ADOC dropped by 34% and 47% respectively. Tail latency values were also elongated by 48% and 28% for RocksDB and ADOC respectively as well. Taking a more microscopic look into the slowdowns, we find that during the workload execution, RocksDB and ADOC experienced a total of 258 and
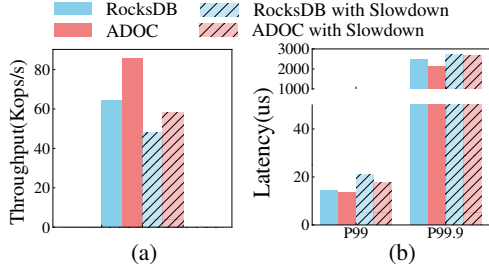
Fig. 3: Throughput (a) and tail latency (b) results of RocksDB and ADOC, based on write slowdown usage.

433 instances of write slowdowns, respectively. Additionally, ADOC also makes use of more CPU resources over RocksDB while still suffering write slowdowns, as seen in Figure 12 in the evaluation section. As slowdowns ultimately throttle write operations over the course of the workload, the performance results inevitably suffer in comparison to results that do not employ slowdown. In addition, the hit in latency performance can be traced to each slowdown causing the write thread to sleep for a short period, worsening write response times. In other words, while the slowdown mechanism alleviates the occurrence of write stalls, it ultimately degrades the overall write performance of LSM-KVS, causing users to experience this performance drop.

B. Underutilized PCIe Bandwidth and Device Resources

In LSM-KVS during a write stall, all user write operations are blocked to allocate system resources for the compaction process. Once compaction is initiated, SSTables (SSTs) are loaded from the storage device/SSD to memory, where a merge-sort operation is performed. Newly created SSTs are then written back to the storage device. Importantly, during the merge-sort phase, no data transfer occurs between host's memory and the storage device. This leaves an interval of time during a write stall where potential transfer bandwidth is being unused, yet write operations are not proceeding.

To empirically verify this behavior, the used PCIe bandwidth of the previous `fillrandom` experiments were measured while measuring PCIe bandwidth at 1-second intervals using Intel PCM [31]. Note that since ADOC's work depends on write slowdowns for its performance optimizations, they were excluded from these experiments. Figure 4 illustrates the time-series PCIe bandwidth measurements for RocksDB without slowdown, focusing on a 100-second segment of the total experiment duration.

Figure 4 (a)-(b) show the results when using one compaction thread (RocksDB(1)) and four compaction threads (RocksDB(4)), respectively. The red dotted lines in the figure indicate the maximum bandwidth of the SSD (630 MB/s), while the green dotted boxes mark periods of write stalls. From the figure, significant unused bandwidth can be observed during the write stall periods from both configurations of RocksDB.

To further analyze the above, we conducted a statistical analysis of the PCIe bandwidth observed during the write
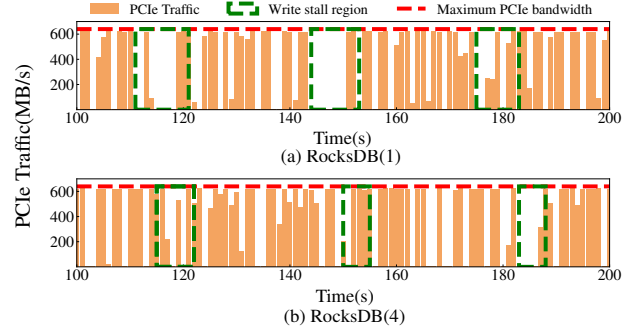

Fig. 4: Measurements of PCIe bandwidth utilization in the 100–200 seconds of execution range in RocksDB without applying slowdown techniques.
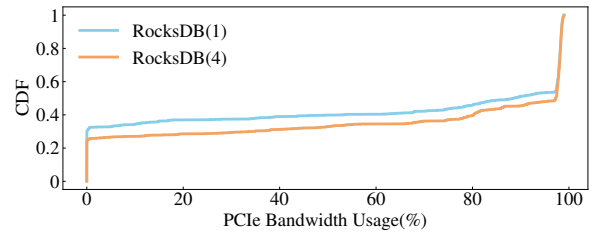

Fig. 5: A Cumulative Distribution Function (CDF) of PCIe bandwidth during a period of write stall on RocksDB. The numbers on the legend denote compaction thread count.

stall periods over the entire 600-second experiment. Figure 5 presents the CDF of PCIe bandwidth utilization during these periods. In RocksDB, with one compaction thread, 30% of the write stall periods exhibit no PCIe bandwidth usage, while 49% use over 90% of available PCIe bandwidth. Four compaction threads improve usage somewhat, where 21% of the write stall periods exhibit no PCIe bandwidth usage and 55% use over 90% of available PCIe bandwidth. While one compaction thread does leave more periods of completely no PCIe bandwidth usage during a write stall, both configurations leave up to 90% of available PCIe bandwidth around 50% of the time during a write stall. Therefore, these results demonstrate that RocksDB in both configurations leaves a significant portion of the device's available PCIe bandwidth underutilized during write stalls.

C. Exploring Available I/O Processing Capacity of the SSD

From the previous experimental results, the following observations can be made.

> **Observation 1.** Both state of the art and industry standard solutions make use of write slowdowns to prevent write stalls, which cause a sharp drop in overall throughput and tail latency.
> **Observation 2.** PCIe bandwidth is under-utilized during write stalls in industry standard LSM-KVS due to the compaction operation blocking device I/O.

These observations lead to a dilemma between two currently possible paths. One can choose to keep slowdowns on and maintain I/O operation service, while coming at the great cost of throttling throughput and deteriorating tail latency. On the other hand, one can disable slowdowns and run the LSM-KVS at maximum capacity, keeping throughput and tail latency alive. However, this leads to enlonged write stalls to occur unpredictably.

The experiments also show a third potential path with discovery of underutilized PCIe and device bandwidth during write stalls. This under-utilization is due to the key-value store halting I/O operations while compaction is in progress. If this underutilized bandwidth can be leveraged in times when the SSD still has available I/O processing capacity, the potential to mitigate write stalls and increase performance without sacrificing system resources can be realized.

IV. OPPORTUNITIES IN KEY-VALUE INTERFACES

Currently, KV-SSDs leverage NVMe extensions [32] to support its key-value interface. The NVMe-based key-value interface API typically supports point and range queries [24], such as PUT, GET, SEEK, and NEXT, and additionally offers buffered I/O capabilities like compound commands [33]. As described in Figure 6, the key-value interface enables efficient I/O processing by eliminating the need for file systems and block layers, effectively simplifying the storage software stack and reducing the overhead associated with multi-layer space management during processing writes and compaction.

The KV-SSDs share the same NAND flash address space and use the same Flash Translation Layer (FTL) mechanisms as traditional block-based NVMe SSDs but internally implement a LSM-KVS at the controller level [22], [23], [25], [34]. The controller abstracts logical addresses for point and range query executions, enabling direct key-value service within the device. Aside from executing point and range queries internally, *the rest of the storage infrastructure, such as the NVMe interface and FTL-managed logical-to-physical address mapping, remains identical to that of conventional SSDs*, ensuring compatibility while offering enhanced functionality. Based on this, we propose designing a hybrid dual-interface SSD that supports both block and key-value interfaces. This approach allows the SSD to leverage the available bandwidth and processing capacity during write stalls in LSM-KVS systems. By temporarily redirecting pending write requests through the key-value interface, the SSD can reduce the impact of write stalls and improve overall performance without disrupting ongoing operations in the LSM-KVS.

V. DESIGN OF KVACCEL

This section introduces the design objectives of KVACCEL, details the hardware and software components involved in its implementation, explains their operation, and discusses how crash consistency is ensured.
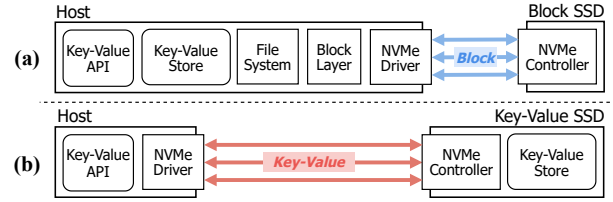

Fig. 6: A comparison of software stacks for (a) NVMe Block Interface SSD and (b) NVMe Key-Value Interface SSD.

A. Design Goals

To address the aforementioned issues, we propose KVACCEL, a novel hybrid hardware-software co-design framework that leverages a dual-interface SSD architecture to eliminate write stalls and optimize the utilization of storage bandwidth. The design goals of KVACCEL are as follows:

G1. Mitigating Write Stalls Effectively: Leverage the key-value interface of the hybrid SSD to serve as a temporary in-device write buffer during host-side write stalls. By redirecting writes to the key-value interface, KVACCEL can prevent the host-side LSM from becoming overloaded during compaction.

G2. Maximizing I/O Bandwidth Utilization: Ensure that the SSD's available bandwidth and I/O processing capacity are fully utilized during write stalls by dynamically switching between the block and key-value interfaces.

G3. Seamless Integration for Consistency and Performance: Achieve seamless integration between the hybrid SSD and host LSM-KVS by employing efficient metadata management and a rollback mechanism. This ensures data consistency between the host's LSM and the device's key-value write buffer, even when switching between interfaces.

B. Overall Architecture

Hardware and Software. KVACCEL is system that offers dynamic redirection and rollback techniques to a LSM-KVS to both mitigate write stalls and fully utilize available I/O bandwidth. This is achieved through the close co-design of software and hardware components. The *Software* components assign I/O commands to the correct interface depending on real-time information of the database. Maintaining the consistency of the database between the two interfaces during database operations is also paramount in the software design. The *Hardware* components implements the disaggregation of separate block and key-value interfaces to allow for the hybrid interface of the SSD. The hardware also implements support for bulk range scan operations over its write buffer to perform the rollback operation for consistency of our system.

Disaggregation and Aggregation. The design of KVACCEL is based on two key factors: disaggregation and aggregation. *Disaggregation* facilitates the division of the SSD into the hybrid interface, as well as the software required for the I/O pathways for each interface. KVACCEL disaggregates a SSD into a hybrid interface with separate block and key-value interfaces, each with its own separate LSM-tree that each interface manages. *Aggregation* focuses on managing the data
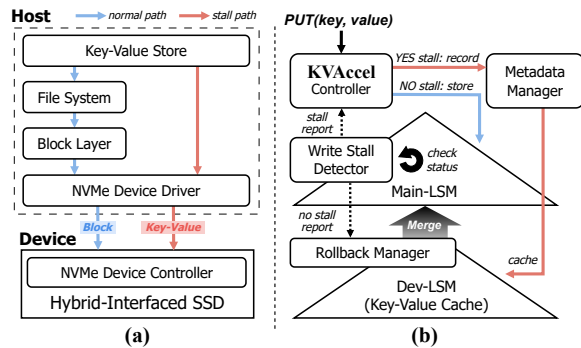
670

Fig. 7: (a) A software stack of KVACCEL and (b) The write path of KVACCEL shown using its software modules.

stored in the hybrid interface SSD as if it were a single database instance. This includes unifying the host-side and device-side LSM-trees during rollback operations by efficiently merging cached key-value pairs from the device back into the host's LSM structure. Additionally, KVACCEL maintains a global metadata manager to track the locations of key-value pairs across both interfaces, ensuring transparent access to data regardless of its physical placement in the SSD.

Figure 7(a) shows the potential of writing using the key-value interface during periods of write stall. Through the key-value interface, I/O operations can bypass the file system and block layer and drill a path straight to the NVMe controller via the driver. This path offers a method to service I/O requests uninterrupted through the key-value interface, even while write stalls are occurring on the database running on the block interface. A key point in disaggregation is that the hybrid interface of both block and key-value interfaces are implemented in a singular device. This is significant in that to see the benefits of KVACCEL, only the one storage device programmed to run is required. A single device solution enables KVACCEL to bypass the burdens of additional hardware deployment (e.g., PM, FPGA) that previous hardware-level solutions to the write stall issue introduce.

C. Interface Pathing via Software Modules

To make use of the hybrid interface, the decision to use which interface needs to be made every time a operation is requested by the database. To do this, KVACCEL makes use of the following four software components shown in Figure 7(b) to make this decision to make full use of unused device bandwidth. The LSM-tree residing on the block interface is labeled *Main-LSM*, while the LSM-tree on the key-value interface labeled *Dev-LSM*. Main-LSM is used by the LSM-KVS running on the host machine, and uses the block interface to serve write operations during periods when write stall is not present. On the other hand, Dev-LSM runs entirely within the hybrid SSD, and uses the key-value interface to serve write operations when Main-LSM is facing a write stall as secondary cache storage.

- **Detector:** The Detector periodically checks three components of Main-LSM that are associated with the character-

istics of a write stall: the number of SSTs in L_0, MT size, and pending compaction size. The Detector then reports this information to the Controller to use for path determination.
- **Controller:** The Controller uses the information reported by the Detector to issue I/O operations to the correct interface. If the Detector reports that no write stalls are occurring, the Controller directs the operation to Main-LSM. If the Detector reports a write stall, the Controller performs the operation to the Dev-LSM.
- **Metadata Manager:** As the SSD has been disaggregated into a hybrid interface, the data written can be in either Main-LSM or Dev-LSM. To keep track of which interface the database needs to use for future read operations, the key-value pairs that are redirected to the Dev-LSM are kept track of. This metadata of a key-value pair's location is captured in a hash table in memory, and is used for membership testing for future operations that need to know the location of a certain key-value pair. In the case of a system failure and data loss of the metadata manager were to happen, the data can be recovered by a range scan covering every key-value pair in the key-value interface.
- **Rollback Manager:** To aggregate the two LSM-trees into one, returning the cachced key-value pairs from Dev-LSM to Main-LSM is required. To facilitate this, the Rollback Manager is tasked to initiate the rollback operation depending on the contention status of Main-LSM. The Rollback Manager receives information of the presence of a write stall from the Detector. Further details on the rollback mechanism can be found in Section V-E.

With these modules, the read and write paths of KVACCEL, depending on the status of the Metadata Manager and the presence of a write stall, can be seen as follows.

- **Read Path:** *(1)* The Metadata Manager checks the location of the queried key. *(2)* If the key-value pair is in the Main-LSM or if the Dev-LSM is empty, the Controller directs the read operation to the Main-LSM. *(3)* If the key-value pair is found in the Dev-LSM, the Controller redirects the read operation to the Dev-LSM.
- **Write Path:** *(1)* The Detector checks for the presence of a write stall. *(2)* If a write stall is detected, the Controller, through the Metadata Manager, updates the record to indicate that the key-value pair is now in the Dev-LSM, and the pair is written to the Dev-LSM. *(3)* If no write stall is detected, the Controller directs the key-value pair to be written to the Main-LSM. *(3-1)* If the Metadata Manager indicates that an overlapping key-value pair already exists in the Dev-LSM, it updates the record to indicate that the latest key-value pair is now in the Main-LSM.

Note that these paths only refer to the point queries of Put() and Get(). For range queries, refer to Section V-F.

D. Hybrid Dual-Interface SSD

To support a storage device with a dual-interface, the SSD's logical NAND flash address space range is disaggregated into two address ranges, as shown in Figure 8. One address range
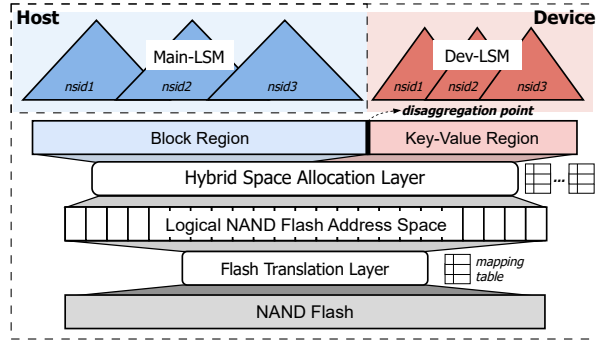
Fig. 8: A dynamic, namespace-aware hybrid NAND flash space allocation of disaggregated NAND flash address space.

Fig. 9: An in-device iterator-based range scan to accelerate host-device co-managed rollback mechanism in KVACCEL.

is used for the block interface, and the other for the key-value interface. The address ranges are defined by the disaggregation point, which is a logical address that defines the end of one interface and the start of the next. The SSD's controller issues different commands for each respective interfaced based on the given opcode of the NVMe command. Block interface commands perform FTL mapping over the logical address space allocated for the block interface. Key-value interface commands allocates NAND pages from the logical address space allocated for the key-value interface. Both interfaces make use of existing NVMe command set specifications made for each respective interface type [32], [35].

As the FTL maps logical address spaces for each interface separately, there are no issues of overlapping logical NAND pages between the two interfaces. When a file system is created for the block interface, the file system only sees the address range that was allocated for the block interface, and reports storage capacity to reflect said address range. Likewise, the key-value interface will only store key-value pairs up to the limits of its allocated address range.

Multi-Tenancy and Multi-Device Support: The ability to create multiple isolated regions on a singular device is a key requirement in multi-tenancy environments. To offer multi-tenancy in KVACCEL, both the block and key-value interface needs to support such features of isolated divisions. Multi-tenancy on the block interface is supported by namespaces as specified in the NVMe standard [36], while previous works on supporting namespaces [37] and multi-tenancy [38] on the key-value interface are compatible with KVACCEL's key-value interface implementation. By utilizing both namespace implementations for each interface and matching namespaces in both interfaces for each tenant, KVACCEL can fully support the requirements of multi-tenancy with both interfaces.

Additionally, the two interfaces can be used as separate devices, where one storage device utilizes the block region, while another the key-value interface. By assigning different NVMe devices as targets for each interface, KVACCEL can be run in a multi-device setup.
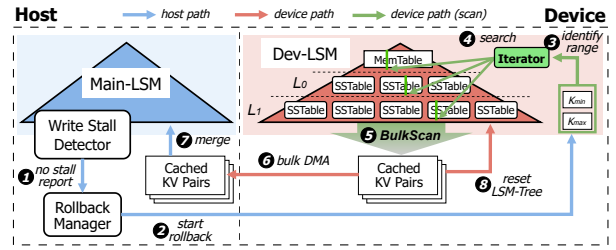
E. Rollback Operation

To return the two separated LSM-KVSs back into a singular database, the cached key-value pairs in Dev-LSM needs to be returned back to Main-LSM. This is done in a process called rollback. Figure 9 displays an overview of the rollback operation, and the interactions between the host and the device during said operation. A rollback operation starts with the Detector and Rollback Manager. ❶ As rollback is only performed during periods when write stall is not present in Main-LSM, the Detector needs to notify the Rollback Manager the appropriate moment to start rollback. ❷ When no write stalls are detected and there are key-value pairs in Dev-LSM, the rollback operation is initiated.

Rollback Scheduling: The rollback manager can schedule a rollback *eagerly* or *lazily* depending on the characteristics of a workload. An eager rollback scheme will trigger rollback as soon as the rollback manager detects that there are enough leftover resources in the LSM-KVS. Such a scheme is better suited for read oriented workloads, as point read query on the Dev-LSM are much slower than its counterpart in the Main-LSM, as such a read operation requires querying the slower device storage every time for a read operation. On the other hand, a lazy rollback scheme will trigger rollback when it is certain no other workload will interfere or be interfered by the rollback. This scheme is designed for write intensive workloads, as there is little penalty to keep the key-value pairs in Dev-LSM in this workload and therefore less urgency to perform rollback.

Iterator-Based Bulky Range Scan: Regardless of the chosen rollback scheduling scheme, rollback needs to be performed as fast as the system allows. This is mainly due to the possibility of I/O operation conflicts. Performance can especially be crippled in cases where read and write operations happen simultaneously, where a time-consuming read operations can impact write operations. Such a conflict can occur with the aforementioned slower point read query on the Dev-LSM. To accelerate the rollback operation, ❸ an iterator first identifies the range of the entire Dev-LSM to perform a range query by using the start and end keys of Dev-LSM. ❹ The iterator will search over the entire Dev-LSM, and ❺ cache key-value pairs are serialized in bulk and transferred to host via device memory using NAND I/O. ❻ key-value pairs are then saved to system memory in chunks of 512 KB, so that the host
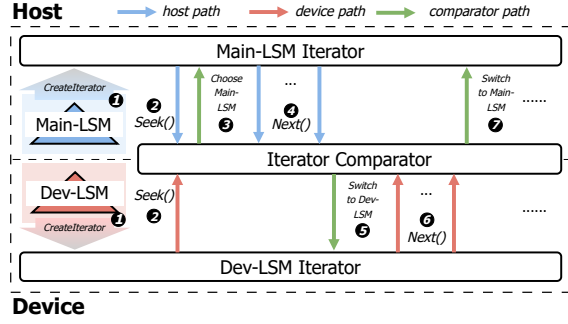
Fig. 10: A range query operation in KVACCEL.

TABLE I: Specifications of the OpenSSD platform.

SoC	Xilinx Zynq-7000 with ARM Cortex-A9 Core
NAND Module	1TB, 4 Channel & 8 Way
Interconnect	PCIe Gen2 ×8 End-Points

TABLE II: Specifications of the host system.

CPU	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz (32 cores), CPU usage limited to 8 cores.
Memory	384GB DDR4
OS	Ubuntu 22.04.4, Linux Kernel 6.6.31

can access the key-value pairs using Direct Memory Access (DMA). This size was chosen as 512 KB is the maximum size unit that DMA supports for data transfer on our platform. ❼ Finally, the host can retrieve and unpacks the key-value pairs to merge back in Main-LSM. ❽ After one rollback operation is finished, a reset is performed on the Dev-LSM to prevent consistency issues in the next rollback operation. By resetting Dev-LSM, the key-value pairs redirected to be involved in the next rollback can be the most up to date data. The reset also ensures the rollback of all key-value pairs in Dev-LSM to be completely written back to Main-LSM.

An important point to keep mind of is as the duration of a write stall is relatively short, Dev-LSM does not have a large amount of SSTs that needs to be rolled back. This fact, along with the aforementioned iterator-based range scan method, can ensure that every rollback operation can be finished in between periods of write stall.

F. Range Query Support

Range queries work with the combination of iterator implementations of each respective interface in KVACCEL. Main-LSM can use the chosen LSM-KVS's implementation of iterator and range scan. Meanwhile, Dev-LSM's key-value interface has support for iterators and range scan functionality [24], and KVACCEL utilizes the same bulky range scan mechanism from the rollback operation. Each interface will have its own iterator to perform Seek() and Next() operations over its LSM-trees. The two iterators will be aggregated to work in tandem to perform a range query over the entire LSM-KVS. An example range query operation is shown in Figure 10. ❶ An iterator for both Main-LSM and Dev-LSM are created, and ❷ a Seek() operation is performed for both LSM-trees. ❸ The values returned from the Seek() operation are sent to the iterator comparator to be compared and saved. The iterator that returned the desired start key, or the smaller key if the desired start key was not found, is selected. ❹ The selected iterator than procedes to perform Next() operations, until the iterator returns a key larger than the key saved from the opposing iterator's first Seek() operation. ❺ The used iterator is then switched, and ❻ Next() operations are continued on the switched iterator. ❼ This process of switching iterators when

necessary continues until the desired end point is reached or the final key-value pair is reached.

G. ACID Property Management

KVACCEL maintains the ACID properties of database transactions by leveraging its dual-interface SSD design. First of all, for atomicity, the disaggregation of NAND flash address space within the dual-interface SSD handles operations between the Main-LSM and Dev-LSM in a completely independent manner. The rollback manager then monitors and reverts any changes made during incomplete transactions, ensuring that any partial or failed transactions during write redirection or rollback are consistently cleaned up by a rollback manager. Consistency is upheld through real-time metadata tracking and validation across both interfaces, with a dynamic consistency checker enforcing strict rollback protocols during high-pressure situations to maintain data accuracy in Main-LSM. The Metadata Manager directs all read and write operations to the appropriate structure, ensuring a seamless transition from Dev-LSM to Main-LSM. To achieve isolation, KVACCEL segregates concurrent I/O operations between the two LSM structures through the Controller Module, isolating Dev-LSM as a temporary cache during write stalls and preventing interference between the interfaces. Each range query is executed independently with separate iterators for each LSM, thereby ensuring query consistency even during ongoing write operations. Durability is guaranteed through a two-stage commit protocol that writes data first to Dev-LSM's non-volatile NAND space before committing it to Main-LSM. This method secures committed transactions even during unexpected power failures or system crashes. In the event of a failure during rollback, the data remains in Dev-LSM until the system is restored, ensuring no loss of committed transactions. This robust architecture makes KVACCEL capable of maintaining database integrity and performance under various system conditions.

VI. EVALUATION

A. Experimental Setup

We implemented KVACCEL's hardware components by extending the state-of-the-art NVMe KV-SSD [24] based on the Cosmos+ OpenSSD platform [26], which has been relied upon by various works for its hardware-level accuracy and reliability [39]–[41]. The SoC of the platform operates the KVACCEL's hybrid-interface SSD controller, the PCIe interface controller, the DRAM controller, and the NAND flash controller. A single ARM core of the Cosmos+ is used to run Dev-LSM's

673

TABLE III: LSM-KVS configurations. For all figures, the numbers next to each LSM-KVS refer to compaction thread count. For KVACCEL, the settings refer to the Main-LSM.

LSM-KVS	Compaction Threads (n)	MT Size
KVACCEL(n)	1	
	2	
	4	
RocksDB(n)	1	128 MB
	2	
	4	
ADOC(n)	1	
	2	
	4	

TABLE IV: *db_bench* workload configurations. Each benchmark was run with a 4 B key and 4 KB value size. Workload A,B,C were run for 600 seconds, and Workload D performed 60K read operations.

Name	Type	Characteristics	Notes (write/read ratio)
A	fillrandom	1 write thread	No write limit
B	readwhilewriting	1 write thread	9:1
C		+ 1 read thread	8:2
D	seekrandom	1 range query thread (Seek + 1024 Next)	Run after initial 20GB fillrandom

I/O operations, as well as other required operations such as flush and compaction operations. The host system runs a modified version of the Linux kernel to facilitate the hybrid-interface SSD, as well as the NVMe block and key-value interface drivers. Table I and Table II present the hardware and software specifications of our setup.

KVACCEL's software components were implemented on RocksDB v8.3.2. The Detector, Controller, Metadata Manager, and Rollback Manager software modules are all implemented on top of RocksDB. The Detector and Rollback Manager in particular run a thread detached from the RocksDB thread, refreshing the status of Main-LSM and checking for conditions of rollback every 0.1 seconds.

For performance evaluations, we slightly modified *db_bench* [27], a widely recognized benchmarking tool used in RocksDB. We enabled *db_bench* to send NVMe key-value commands to the Cosmos+ OpenSSD platform through the NVMe passthrough. The LSM-KVSs and the configurations used for the evaluations are detailed in Table III. The various patterns of the workloads to verify our proposed design are described in Table IV.

From Tables I and II, a mismatch in our environment can be seen with the CPU and the interconnect being used. The mismatch of a modern, high-performance CPU with the deprecated and slower PCIe 2.0 interconnect leads to a higher I/O request and compaction processing rate from the CPU in relation to the capabilities of the interconnect. This leads to premature saturation of PCIe bandwidth during compaction in Main-LSM with even a slight increase in compaction thread count. As KVACCEL's effectiveness depends on remaining PCIe bandwidth during compaction, this mismatch caused issues in demonstrating KVACCEL's effectiveness. Therefore,
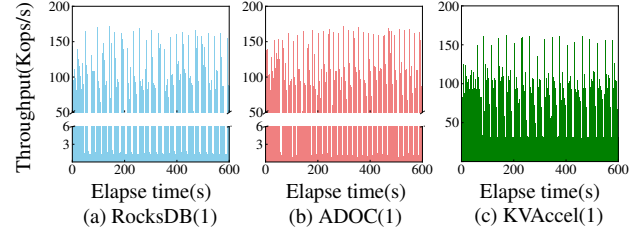

Fig. 11: Per-second throughput for each LSM-KVS while running workload A.
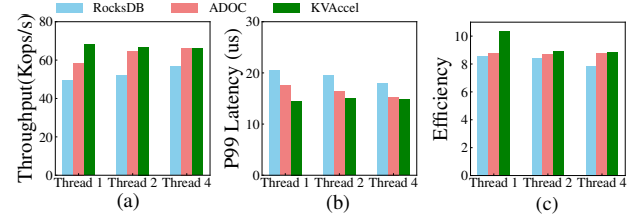

Fig. 12: (a) Throughput, (b) P99 Latency, and (c) Efficiency scores of all evaluated LSM-KVS for workload A. Thread counts here denote compaction thread count.

compaction thread count was limited to maximum of four to account for this mismatch. With a more modern PCIe version, this mismatch is expected to be alleviated, making KVACCEL's effectiveness evident regardless of the compaction thread count.

B. Write Stall Mitigation Evaluation

This section demonstrates KVACCEL's ability to mitigate write stalls via I/O redirection. Figure 11 displays the per-second throughput of all three LSM-KVS (RocksDB, ADOC, and KVACCEL) during the entirety of workload A. Figure 11 (a) and (b) focus on the periods of lower throughput in order to examine the decrease in throughput that occurred during the slowdown phase. ADOC and RocksDB can be both seen suffering from slowdowns to 2 Kop/s in order to prevent a write stall. In similar periods, KVACCEL proceeds to write upwards of 30 Kop/s, showing I/O redirection response of KVACCEL allowing for the avoidance of write stalls.

A point to emphasize here is KVACCEL does not employ any slowdown mechanisms to avoid a write stall. This is because KVACCEL is inherently designed to accept writes in its full capacity during a write stall via redirection instead of intentionally throttling write flow to attempt to avoid a write stall. This different approach to the write stall problem allows KVACCEL to maintain write operations while greatly lowering performance compromises, while other LSM-KVSs suffer from slowdowns or face a write stall depending on workload settings.

C. Performance Evaluation

In this section, the read/write performance, and efficiency of KVACCEL will be demonstrated with the workloads of Table IV. Here, we introduce a scoring metric of the ratio between throughput and CPU resources as a form of an efficiency measurement.

$$\text{Efficiency} = \frac{\text{Avg. Throughput(MB/s)}}{\text{Avg. CPU usage(\%)}} \quad (1)$$

This efficiency metric aims to elucidate the required resource utilization of the CPU to produce the throughput results of each LSM-KVS for a workload. A LSM-KVS that scores higher in the efficiency metric demonstrates that it requires less CPU resources to produce the same throughput results than a lower scoring LSM-KVS.

Figure 12 shows the average throughput, P99 latency, and efficiency respectively of all LSM-KVS configurations performing workload A. To demonstrate the full potential of KVACCEL in a write-only operation, rollback and compaction operations in Dev-LSM were disabled for workload A. This is because for a write-only workload phase, a lazy rollback scheme that performs rollback after the workload completes is the most sensible option.

KVACCEL ensures continuous user service by redirecting I/O requests to alternative interface when write stalls occur. This proactive handling of write stalls by KVACCEL results in improvements in both throughput and tail latency. Specifically, when utilizing a single compaction thread, KVACCEL achieved throughput improvements of 37% and 17% compared to RocksDB and ADOC, respectively. Additionally, KVACCEL exhibited decreases in P99 latency of 30% and 20%, respectively. A noteworthy observation is that KVACCEL, operating with only one compaction thread, achieves write throughput comparable to ADOC using four compaction threads. This occurs because KVACCEL significantly contributes to throughput when write stalls are longer and more frequent. However, increasing the compaction thread count generally reduces the length and frequency of write stalls, thereby diminishing KVACCEL's relative effectiveness. Regardless, KVACCEL's redirection mechanism guarantees a consistent level of user service with higher throughput over that of a LSM-KVS under slowdown.

Referring to the efficiency metric for the results, KVACCEL also maintains the better efficiencies in host machine's resources between all LSM-KVS compared, with KVACCEL(1) shows the best efficiency over all configurations. This is because KVACCEL is able to achieve the higher throughput results while maintaining the same CPU utilization.

Additionally, to show KVACCEL's performance in more diverse scenarios, KVACCEL was evaluated under different rollback schemes running workloads A to C. The results of these workloads of all LSM-KVS configurations can be seen in Figure 13, where comparisons of rollback schemes based on workload type are also made. Here, KVACCEL-L and KVACCEL-E refer to KVACCEL with lazy and eager rollback schemes respectively. For workload A, KVACCEL-L shows superior write performance over KVACCEL-E, as it is a write only workload, leading rollback operations to take away bandwidth from actual write operations. However, both configurations show lower performance in comparison to the write optimized KVACCEL as shown in Figure 12.

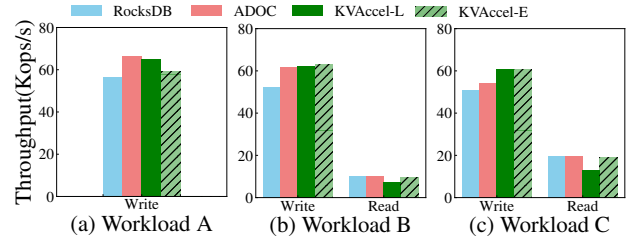Workload B and C present a read-write mix workload, where

Fig. 13: Read and write throughput comparison of different workloads based on rollback schemes choice. KVACCEL-L uses a lazy rollback scheme, and KVACCEL-E uses an eager rollback scheme. All LSM-KVS configurations in this figure use 4 compaction threads. Read throughput is non-applicable for workload A, as workload A is a 100% write workload, and is thus excluded.

both rollback schemes achieve similar write throughput, both holding a lead of 36% and 51% over ADOC respectively. However, KVACCEL-E shows an increase in read performance, due to rollback allowing more read operations to be performed from Main-LSM, showing that a eager rollback scheme can be more effective for a write/read mixed workload.

TABLE V: Throughput of range queries for RocksDB, ADOC, and KVACCEL performing workload D.

LSM-KVS	Range Query Throughput (Kops/s)
RocksDB	302
ADOC	351
KVACCEL	100

Table V shows the results of range query workloads from workload D. These results prove that KVACCEL is able to fully support the range query operation across the hybrid interfaces. However, KVACCEL still suffers a significant performance hit in comparison to other LSM-KVS. This is in large part due to a lack of read caching mechanism for iterator operations on the Dev-LSM. Without a read cache located in fast memory for Dev-LSM's iterator, its range query performance lags behind significantly in contrast to the Main-LSM's iterator. This predicament acts as a bottleneck, resulting in KVACCEL's range query throughput to be bound to Dev-LSM's range query.

D. Recovery Process

In the event of a system failure, the hashtable managed by the Metadata Manager is lost, as it resides in volatile memory. To recover from this, all KV pairs stored in Dev-LSM are rolled back to Main-LSM, effectively mitigating the impact of the hashtable loss. Since all KV pairs are successfully restored to Main-LSM, the absence of the hashtable does not affect system integrity. Notably, restoring 10,000 KV pairs from Dev-LSM to Main-LSM required 1.1 seconds, demonstrating that the recovery process incurs minimal overhead.

E. Overhead Analysis

Through the additional software modules that KVACCEL implements, there are unavoidable overhead processes on top

TABLE VI: Detailed breakdown of time overheads for KVAC-CEL's operations.

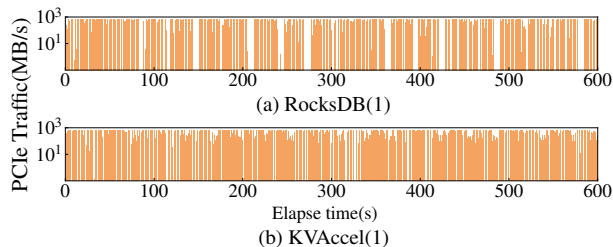| Operation	Average Elapsed Time (us)
Detector	1.37
Key Insert	0.45
Key Check	0.20
Key Delete	0.28

Fig. 14: Overview of PCIe bandwidth usage for (a) RocksDB(1) (b) KVACCEL(1) in logarithmic scale.

of the core LSM-KVS operations. A breakdown of all the potential overheads of KVACCEL are covered in Table VI.

The Detector module has the largest overhead impact, with an average of 1.37 microseconds every 0.1 seconds it is used. The Metadata Module is also a required overhead, due to the requirement of maintaining consistency between the dual interfaces. For this there are the key insert, check and delete operations, which on average, takes 0.45, 0.2 and 0.28 microseconds respectively. In practice, during workloads, the largest overhead observed related to the Metadata Manager was the combination of a key check and delete operation, which took 0.48 microseconds.

F. Microscopic Analysis of PCIe Usage

To verify the usage of PCIe bandwidth of KVACCEL, we conducted experiments with Workload A and measured the bandwidth utilization by using Intel PCM [31]. Figure 14 shows the results in time series in comparison to baseline RocksDB. In Figure 14, KVACCEL achieved a 45% reduction in zero-traffic intervals during write stall periods compared to RocksDB. It can be observed that KVACCEL takes advantage of its dual interface and demonstrate high PCIe utilization which aligns with the results presented in Figure 11.

VII. CONCLUSION

There has been extensive research on mitigating write stalls in LSM-tree-based key-value stores. However, these existing studies fall short in overcoming the write stalls and limits the performance gain. This study introduces KVACCEL, the first hardware-software co-design that revitalizes the underutilized computational power of SSDs during compaction to avoid write stalls. KVACCEL integrates a dual-interface SSD architecture, dynamically redirecting writes to a key-value interface during host-side write stalls, eliminating the need for complex host-side optimizations, high CPU usage, or additional hardware. We implemented KVACCEL by extending RocksDB to support

I/O redirection during write stalls. Our evaluation shows that KVACCEL outperforms ADOC in throughput and CPU efficiency for write-heavy workloads, while both systems perform comparably in mixed read-write scenarios.

REFERENCES

[1] Facebook, "RocksDB." https://rocksdb.org, 2012. Last Accessed: 2024-09-01.

[2] Google, "LevelDB." https://github.com/google/leveldb, 2017. Last Accessed: 2024-09-01.

[3] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores," in *Proceedings of the USENIX Annual Technical Conference*, ATC '19.

[4] S. Jamil, A. Khan, K. Kim, J.-K. Lee, D. An, T. Hong, S. Oral, and Y. Kim, "DenKV: Addressing Design Trade-offs of Key-value Stores for Scientific Applications," in *Proceedings of the IEEE/ACM International Parallel Data Systems Workshop*, PDSW '22.

[5] J. Yu, S. H. Noh, Y.-r. Choi, and C. J. Xue, "ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '23.

[6] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM," in *Proceedings of the USENIX Annual Technical Conference*, ATC '20.

[7] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating Synergies between Memory, Disk and Log in Log Structured Key-Value Stores," in *Proceedings of the USENIX Annual Technical Conference*, ATC '17.

[8] C. Ding, T. Yao, H. Jiang, Q. Cui, L. Tang, Y. Zhang, J. Wan, and Z. Tan, "TriangleKV: Reducing Write Stalls and Write Amplification in LSM-Tree Based KV Stores With Triangle Container in NVM," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, p. 4339–4352, Dec. 2022.

[9] Facebook, "Write Stalls." https://github.com/facebook/rocksdb/wiki/Write-Stalls, 2021. Last Accessed: 2024-08-29.

[10] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning LSMs for Nonvolatile Memory with NoveLSM," in *Proceedings of the USENIX Annual Technical Conference*, ATC '18.

[11] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "SLM-DB: Single-Level Key-Value store with persistent memory," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '19.

[12] X. Sun, J. Yu, Z. Zhou, and C. J. Xue, "FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores," in *Proceedings of the IEEE 36th International Conference on Data Engineering*, ICDE '20.

[13] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun, "FPGA-Accelerated Compactions for LSM-Based Key-Value Store," in *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST '20.

[14] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-Engine: An Optimized Storage Engine for Large-scale E-Commerce Transaction Processing," in *Proceedings of the International Conference on Management of Data*, SIGMOD '19.

[15] P. Xu, J. Wan, P. Huang, X. Yang, C. Tang, F. Wu, and C. Xie, "LUDA: Boost LSM Key Value Store Compactions with GPUs," *arXiv preprint arXiv:2004.03054*, 2020.

[16] H. Zhou, Y. Chen, L. Cui, G. Wang, and X. Liu, "A GPU-Accelerated Compaction Strategy for LSM-based Key-Value Store System," in *Proceedings of the 38th International Conference on Massive Storage Systems and Technology*, MSST '24.

[17] H. Sun, J. Xu, X. Jiang, G. Chen, Y. Yue, and X. Qin, "gLSM: Using GPGPU to Accelerate Compactions in LSM-tree-based Key-value Stores," *ACM Transactions on Storage*, vol. 20, Jan. 2024.

[18] C. Ding, J. Zhou, J. Wan, Y. Xiong, S. Li, S. Chen, H. Liu, L. Tang, L. Zhan, K. Lu, *et al.*, "DComp: Efficient Offload of LSM-tree Compaction with Data Processing Units," in *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23.

[19] I. Park, Q. Zheng, D. Manno, S. Yang, J. Lee, D. Bonnie, B. Settlemyer, Y. Kim, W. Chung, and G. Grider, "KV-CSD: A Hardware-Accelerated Key-Value Store for Data-Intensive Applications," in *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '23.

[20] C. Ding, J. Zhou, K. Lu, S. Li, Y. Xiong, J. Wan, and L. Zhan, "D2Comp: Efficient Offload of LSM-tree Compaction with Data Processing Units on Disaggregated Storage," *ACM Transactions on Architecture and Code Optimization*, vol. 21, Sept. 2024.

[21] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A Flexible, High-Performance Key-Value SSD," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, HPCA '17.

[22] C.-G. Lee, H. Kang, D. Park, S. Park, Y. Kim, J. Noh, W. Chung, and K. Park, "iLSM-SSD: An Intelligent LSM-tree based Key-value SSD for Data Analytics," in *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '19.

[23] J. Im, J. Bae, C. Chung, Arvind, and S. Lee, "PinK: High-speed In-storage Key-value Store with Bounded Tails," in *Proceedings of the USENIX Annual Technical Conference*, ATC '20.

[24] S. Lee, C.-G. Lee, D. Min, I. Park, W. Chung, A. Sivasubramaniam, and Y. Kim, "Iterator Interface Extended LSM-tree-based KVSSD for Range Queries," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, SYSTOR '23.

[25] J. Park, C.-G. Lee, S. Hwang, S. Yang, J. Noh, W. Chung, J. Lee, and Y. Kim, "BandSlim: A Novel Bandwidth and Space-Efficient KV-SSD with an Escape-from-Block Approach," in *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP '24.

[26] "Cosmos+ OpenSSD Platform." http://www.openssd-project.org/platforms/cosmospl, 2017. Last Accessed: 2024-09-22.

[27] Facebook, "DB Bench." https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2017. Last Accessed: 2024-10-01.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-Structured Merge-Tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[29] "Cassandra." https://cassandra.apache.org/, 2008. Last Accessed: 2024-02-14.

[30] R. Sears and R. Ramakrishnan, "bLSM: A General Purpose Log Structured Merge Tree," in *Proceedings of the International Conference on Management of Data*, SIGMOD '12.

[31] Intel, "Intel® performance counter monitor - a better way to measure cpu utilization." https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html, 2022. Last Accessed: 2024-10-01.

[32] NVM Express Inc., "NVM Express Key Value Command Set Specification." https://nvmexpress.org/developers/nvme-specification/, 2021. Last Accessed: 2024-09-12.

[33] S.-H. Kim, J. Kim, K. Jeong, and J.-S. Kim, "Transaction Support using Compound Commands in Key-value SSDs," in *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage '19.

[34] Y. Park, J. Park, A. Khan, J. Park, C.-G. Lee, W. Chung, and Y. Kim, "OCTOKV: An Agile Network-Based Key-Value Storage System with Robust Load Orchestration," in *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '23.

[35] NVM Express Inc., "NVM Express Specification." https://nvmexpress.org/developers/nvme-specification, 2011. Last Accessed: 2024-09-12.

[36] NVM Express Inc., "NVMe Namespaces." https://nvmexpress.org/resource/nvme-namespaces, 2022. Last Accessed: 2024-09-12.

[37] D. Min and Y. Kim, "Isolating Namespace and Performance in Key-Value SSDs for Multi-Tenant Environments," in *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21.

[38] D. Min, K. Kim, C. Moon, A. Khan, S. Lee, C. Yun, W. Chung, and Y. Kim, "A Multi-tenant Key-value SSD with Secondary Index for Search Query Processing and Analysis," *ACM Transactions on Embedded Computing Systems*, vol. 22, July 2023.

[39] D. Min, Y. Ko, R. Walker, J. Lee, and Y. Kim, "A Content-Based Ransomware Detection and Backup Solid-State Drive for Ransomware Defense," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, pp. 2038–2051, July 2022.

[40] D. Min, D. Park, J. Ahn, R. Walker, J. Lee, S. Park, and Y. Kim, "Amoeba: An Autonomous Backup and Recovery SSD for Ransomware Attack Defense," *IEEE Computer Architecture Letters*, vol. 17, p. 245–248, July 2018.

[41] J. Ahn, J. Lee, Y. Ko, D. Min, J. Park, S. Park, and Y. Kim, "DISKSHIELD: A Data Tamper-Resistant Storage for Intel SGX," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, ASIACCS '20.