BandSlim: A Novel Bandwidth and Space-Efficient KV-SSD with an Escape-from-Block Approach

Junhyeok Park
junttang@sogang.ac.kr
Sogang University

Chang-Gyu Lee
changgyu@u.sogang.ac.kr
Sogang University

Soon Hwang
soonhw@sogang.ac.kr
Sogang University

Soonyeal Yang
soonyeal.yang@sk.com
SK hynix Inc.

Jungki Noh
jungki.noh@sk.com
SK hynix Inc.

Woosuk Chung
woosuk.chung@sk.com
SK hynix Inc.

Junghee Lee
j_lee@korea.ac.kr
Korea University

Youngjae Kim*
youkim@sogang.ac.kr
Sogang University

ABSTRACT

The Key-Value Solid State Drive (KV-SSD) represents a significant evolution in storage device interfaces by accommodating non-page-aligned key-value pairs, a departure from conventional models. However, KV-SSDs encounter challenges as their specialized data transfer and packing requirements conflict with established storage protocols like NVMe, which are designed around fixed memory page units. This discord leads to inefficient data movement and increased NAND page write I/Os, which in turn escalates network traffic and degrades both performance and NAND efficiency. To tackle these challenges, this paper introduces *BandSlim*, a novel solution equipped with two methods to streamline bandwidth during I/O transmission: (i) a fine-grained inline value transfer utilizing NVMe commands for bandwidth-efficient value transfer, and (ii) a selective value packing strategy combined with a backfilling policy to reduce NAND page write I/Os. We integrated *BandSlim* on a state-of-the-art FPGA-based LSM-tree KV-SSD, utilizing the Cosmos+ OpenSSD platform. Our comprehensive evaluations illustrate that *BandSlim* achieves a remarkable reduction in PCIe traffic of up to 97.9% and NAND page write counts by up to 98.1% compared to the NVMe-based KV-SSD without employing *BandSlim*.

CCS CONCEPTS

• **Computer systems organization → Firmware**; • **Information systems → Flash memory**; **Storage management**.

KEYWORDS

Key-Value Solid State Drive, Non-Volatile Memory Express, Peripheral Component Interconnect Express.

*Y. Kim is the corresponding author.

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673064

ACM Reference Format:
Junhyeok Park, Chang-Gyu Lee, Soon Hwang, Soonyeal Yang, Jungki Noh, Woosuk Chung, Junghee Lee, and Youngjae Kim. 2024. *BandSlim*: A Novel Bandwidth and Space-Efficient KV-SSD with an Escape-from-Block Approach. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673064

1 INTRODUCTION

Designing an efficient storage system necessitates reducing data movement costs from the host's memory to the storage media. However, traditional Key-Value Stores (KVS) like RocksDB [10] and LevelDB [11] function as middleware on top of file systems. Consequently, user I/O requests must navigate through the kernel's file system and block layers to execute data reads and writes to storage. This multi-layer traversal incurs significant memory copying and kernel context switch overheads during I/O operations. In contrast, Key-Value Solid State Drives (KV-SSDs) [13, 21, 29] offer a substantial reduction in these overheads. By eliminating the file system and block layer within the kernel, KV-SSDs provide lower latency and higher throughput compared to traditional KVSs.

Unlike block-based SSDs, one of the noteworthy advantages of KV-SSDs is that they allow the design of I/O subsystems to handle users' variable size requests precisely and in the exact size. This design opportunity processing I/O operations in exact size can significantly enhance storage efficiency regarding space utilization and response time. Unfortunately, to date, commercially and academically released KV-SSDs such as Samsung KVSSD [29], PinK [13], and iLSM-SSD [21] utilize the NVMe protocol, which is specifically engineered for block-based storage devices. These KV-SSDs implement the Physical Region Page (PRP) list for conveying payload, essentially inducing I/O amplification originating from the size difference between the block and key-value pair.

Moreover, the misalignment of key-value pair and block size is not the only cause of I/O amplification. KV-SSDs conceal the same problem, which is coming from the NAND flash I/O unit. Because typical NAND flash memory requires writing in a NAND page unit (16 KB) larger than the block (4 KB), the I/O amplification problem is inherent in NAND flash-based drives. However, modern block-based SSDs effectively mitigate this issue by employing a

NAND page buffer [5, 16, 18] using DRAM within the device. By aligning the NAND page size to fit multiple blocks, a buffer inside can successfully amortize the NAND page write cost over multiple block writes with minimal cost, which essentially minimizes I/O amplification. KV-SSDs, however, face the additional challenge of managing byte-level offsets to accommodate multiple key-value pairs in NAND page-sized buffer entries due to the variable nature of the key and value sizes.

Limitations of Existing Approaches: There have been two approaches to mitigate the amplification occurring in the KV-SSD. One can be tackled from the host side by batching enough key-value entries to fit the NAND page I/O semantic within the device easily. Recently proposed KV-SSDs such as Dotori [9] and KV-CSD [27] took this approach by implementing bulk PUT operation, which is host-side batching. However, a fundamental issue with buffering the key-value entries on the host side is the risk of data loss on power failure. Additionally, because the host sends a chunk of payload packed with multiple key-value pairs, KV-SSDs that need to index and organize each key separately face extra overhead from unpacking them. KAML [15] tackles the problem within the device rather than the host side. The KV-SSD shown in KAML suggests building a log directly on the NAND page utilizing the buffer inside the device; consequently, NAND pages form a log consisting of a batch of multiple key-value pairs. With this approach, byte-level offset management can be alleviated via log design. However, considering a NAND page-sized buffer is filled with key-value pairs within the KV-SSD device, the amplification due to the misalignment of the transmission unit and key-value size still occurs. *To the best of our knowledge, this study is the first comprehensive research to identify and resolve both amplification occurring at transmission and NAND page write in KV-SSD.*

To tackle both amplifications occurring in small key-value transfer and storing NAND flash pages, we introduce *BandSlim*. *BandSlim* aims to transfer small key-value payloads with minimal PCIe traffic overhead, thereby enhancing data transfer efficiency between the host and KV-SSD. Additionally, *BandSlim* introduces the design to minimize the number of NAND page writes required to persist key-value pairs by packing small and large key-value records densely into NAND pages as possible in a backfilling manner. However, note that the design decisions *BandSlim* made to meet these goals are not against the NVMe standard. It is more of an NVMe-compatible proposal to keep its various utilities from device identification to device management. *BandSlim* consists of two methods: (i) *Fine-Grained Value Transfer* and (ii) *Efficient Fine-Grained Value Packing*.

Fine-Grained Value Transfer employs an inline value transfer mechanism that piggybacks values smaller than a memory page size to NVMe commands using the reserved fields. We observed that it substantially reduces data traffic over the PCIe interconnect. However, we also observed that the response time increases rapidly as the payload size approaches a memory page size. This is obvious because the available space in a single NVMe command is far less than a 4 KB page size; thus, multiple NVMe commands must be issued as payload size increases. Thus, *BandSlim* also incorporates an adaptive value transfer strategy that switches back and forth piggybacking and Direct Memory Access (DMA) to take balance between reduced PCIe traffic and response time.

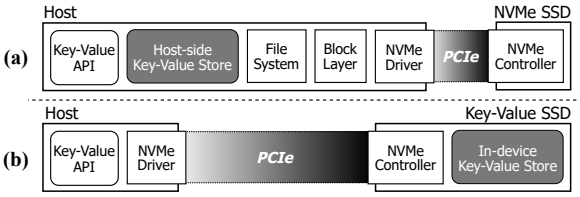


Figure 1: Comparison of software stacks for (a) Host-side Key-Value Store and (b) Key-Value Solid State Drive.

Efficient Fine-Grained Value Packing implements a *Selective Packing with Backfilling Policy* specifically designed for better NAND page utilization. *BandSlim* specifically locates small payloads to fill the gap formed by the page-aligned payload, which is transferred by DMA prior to the small payload. This design choice reflects restrictions in some real-world in-device DMA engines, which require that the transfer size and destination address within the device be aligned with a 4 KB page size. To provide compatibility across various DMA engines as other kernel or device drivers does, *BandSlim* is built on the assumption that some payloads are needed to be placed 4 KB aligned offset with 4 KB aligned size [8, 26, 28, 31].

For evaluation, we implemented *BandSlim* in a state-of-the-art FPGA-based KV-SSD [22] built on the Cosmos+ OpenSSD platform [19]. We demonstrated that *BandSlim* reduces PCIe traffic by up to 97.9% and issues up to 98.1% fewer NAND page writes compared to the KV-SSD without *BandSlim*.

The contributions of this paper are summarized as follows:

- Identified traffic amplification in host-device interconnect, specifically in KV-SSDs, and proposed NVMe-compatible design to minimize bandwidth waste during small key-value pair transfer.
- Demonstrated the trade-off between PCIe traffic and response time in fine-grained value transfer and effectively resolved it using an adaptive value transfer method.
- Proposed small payload packing design with a backfilling approach to increase NAND page write efficiency while considering the DMA engine's page-alignment restriction.

2 BACKGROUND AND MOTIVATION

2.1 LSM-tree-based Key-Value Solid State Drive

Key-Value Solid State Drives (KV-SSDs) have renovated the storage interface by changing the unit of I/O transactions from the traditional block to key-value [15, 29]. By providing Key-Value Store (KVS) functionalities at the device level, KV-SSDs enabled a thinner storage software stack, which excludes traditional file systems and block layers, thereby significantly reducing overhead caused by multiple layers of space management.

One of the dominant KVS designs employed in recent KV-SSD studies is the Log-Structured Merge-tree (LSM-tree) with the key-value separation [7, 13, 21, 22, 27]. The LSM-tree is a data structure already used in many mature host-side KVSs, such as RocksDB [10] and LevelDB [11] due to its strength in processing write-intensive workloads. In LSM-tree's original form, a key-value pair is stored together in the same file called SSTable. However, as LSM-tree's maintenance operation called a compaction job is pointed out to be

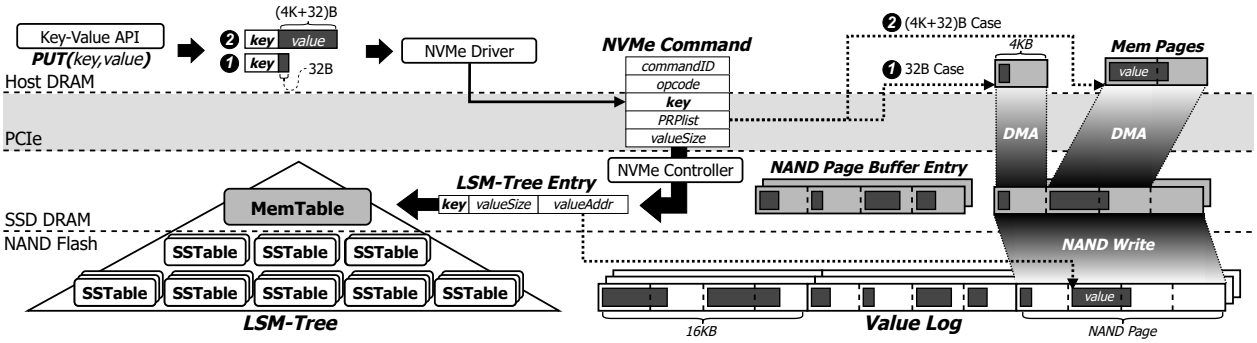


Figure 2: Data flow of two cases of key-value pair transfers with sub-4 KB payload and over-4 KB payload regarding NAND page buffer management and PCIe traffic bloating in LSM-tree-based Key-Value Solid State Drives.

a cause of significant write amplification, which degrades performance and storage devices endurance, a key-value separation has been proposed [23]. The key-value separation design summarizes separating values from LSM-tree and storing to Value Log (vLog), as its name suggests. With this design, the compaction, which is essentially responsible for reading SSTables, merging them, and writing only live key-value pairs to new SSTable files, no longer repeatedly rewrites live values, thus reducing the write amplification.

In comparison with traditional host-side KVS depicted in Figure 1(a), Figure 1(b) shows the storage stack that is using KV-SSDs with LSM-tree and key-value separation [13, 21, 22, 27] (hereafter referred to simply as KV-SSD). Since the KV-SSD can bypass the file system and block layers, it consists of user-level key-value APIs, a key-value device driver, and controller based on protocols like NVMe, and an in-storage LSM-tree-based KVS. The user-level key-value API offers point and range queries, namely PUT, GET, SEEK, and NEXT. The size of the key and value in these APIs is handled as arbitrary length, not in block units (*key-value interface*). A pair of key and value address is stored in the LSM-tree, and a value is stored in the vLog. The vLog of KV-SSD is a linear, logical NAND flash address space. This space is further divided into multiple logical NAND pages. Each value is appended to the vLog sequentially, filling logical NAND pages. Note that it fills logical NAND pages which are mapped to physical NAND pages by the FTL (Flash Translation Layer). The entries of the LSM-tree point to corresponding values inside the vLog.

2.2 NVMe Key-Value Storage Device Interface

Key-Value Pair Transfer Mechanism: The NVMe protocol has introduced a key-value command set for KV-SSDs [25]. Within the NVMe key-value interface, when writing key-value pairs, the NVMe driver stores a key and metadata in the reserved fields of the NVMe command. The payload, which is the value in this context, is transferred via the Physical Region Page (PRP) as the block interface part of NVMe specification does. The PRP is a linked list whose entry describes the addresses of physical memory pages of the host memory. One or more memory pages where the value is stored are specified to be transferred. Subsequently, the driver inserts the NVMe command into the submission queue and rings the doorbell to notify the device of the write request. The NVMe controller fetches the command from the queue, interprets it, and identifies the pages for copying from the received PRP list. To initiate the value transfer, the controller triggers a Direct Memory Access (DMA) transaction, which copies pages from the host memory to the device memory. The controller later inserts the received key to the memory component of LSM-tree, MemTable, and writes the value to the vLog (see Figure 2). The reverse operation, involving the transfer of values from the KV-SSD to the host, follows a similar process.

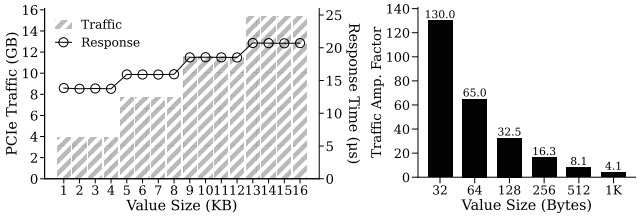
NAND Page Buffer Management: In alignment with NVMe SSDs, NVMe KV-SSDs incorporate a NAND page buffer [16] within the SSD's battery or capacitor-backed DRAM (see Figure 2). Each buffer entry serves as a persistent write buffer for NAND pages. When inserting values into the vLog, multiple values can be packed into a single buffer entry if the size of each value is smaller than the NAND page. When the buffer entry can no longer hold values, its contents are written to the physical NAND page which is FTL-mapped to the current logical NAND page of the vLog region.

2.3 PCIe Traffic Amplification and NAND Write Amplification in KV-SSD

As in typical KVSs, the key and value size are variable and not necessarily aligned to a memory page. According to Meta, RocksDB in a production environment experiences the size of values nearly not reaching a hundred bytes on average [3], which is far less than the 4 KB memory page size. Consequently, a KV-SSD must be capable of effectively handling requests for such variable-sized, small values. However, a naive adoption of the current NVMe standard to support the key-value interface causes inefficiencies in value transfer and NAND I/O utilization as follows:

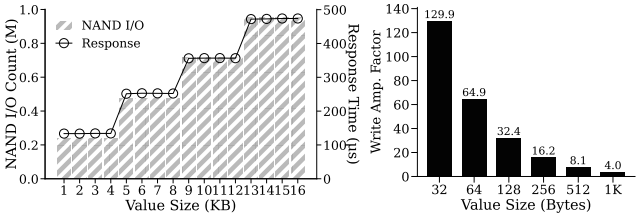
Problem#1–Bloated PCIe Traffic: This problem occurs because the NVMe key-value interface adheres to the same procedure as the original block-interfaced NVMe protocol when transferring values to or from the device [24, 25]. Specifically, *the NVMe's data transfer method, PRP, restricts DMA transfers to occur in units of 4 KB, a size of memory page* [6]. This restriction aligns with the historical evolution of the block storage stack, which has evolved to align with memory page units [8, 26, 28, 31].

As shown in Figure 2, consider a scenario where the value size is 32 bytes (❶). In this case, one 4 KB memory page that temporarily holds the value is specified by the PRP, and a DMA copy of 4 KB occurs. On the other hand, if the value size slightly exceeds the memory page size, such as (4K+32) bytes (❷), two memory pages are required to accommodate the value. Consequently, the DMA facilitated by the PRP transfers 8 KB of data. This bloated data traffic



(a) Total PCIe Traffic & Avg. Resp. Time (b) Traffic Amp. Factor

Figure 3: Total PCIe Traffic with Average Response Time and PCIe Traffic Amplification Factor for varying value sizes.



(a) Total NAND I/O & Avg. Resp. Time (b) Write Amp. Factor

Figure 4: Total NAND Page Writes with Average Response Time and Write Amplification Factor for varying value sizes.

on PCIe can lead to a significant increase in energy consumption and power usage of the system, ultimately elevating the total cost of ownership [2]. This is why modern data centers prioritize reducing data movement costs as a primary mission [4, 30].

Problem#2–NAND Write I/O Amplification: The second issue is that *the packing of received objects into NAND pages within NVMe SSDs also occurs in units of 4 KB memory pages* [5, 18]. The NAND page buffer in NVMe SSDs packs data along 4 KB boundaries as depicted in Figure 2. If the size of a NAND page is 16 KB, data from the host can fill one NAND page buffer entry with a maximum of four write requests (for example, with a value size of 32 bytes). If the value size is marginally larger than one memory page, such as (4K+32) bytes, only two write requests are sufficient to fill one buffer entry, beyond which it cannot be further filled. This in-device page-unit packing clearly clashes with KV-SSDs, leading to severe NAND write amplification.

The NVMe protocol still enforces memory page unit payload transfers tailored to and originated from the traditional block interface for the new key-value interface of KV-SSDs, and further executes memory page unit payload packing over the NAND page buffer within the device, just like block-based SSDs.

2.4 Experimental Analysis of Amplification

Bloated PCIe Traffic: We measured PCIe traffic from a host to a device by issuing 1 million write requests with variable-sized values using a state-of-the-art NVMe KV-SSD [22]. Experimental setup details are in Section 4.1. We utilized the Intel Performance Counter Monitor (PCM) [14] to track PCIe traffic.

Figure 3(a) illustrates the total data transferred over PCIe during the experiments. Notably, PCIe data transfer exhibits a doubling in traffic volume at precise 4 KB value size boundaries. For instance, the total data transferred for 1 KB and 4 KB value sizes is about 4 GB, showing that transfer volumes remain constant for value sizes up to 4 KB. This pattern repeats for value size ranges of 5-8 KB, 9-12 KB, and 13-16 KB, aligning the total PCIe traffic with that of the smallest multiple of 4 KB covering the value size. We also recorded the average transfer response times, which revealed a similar cascading pattern as seen in PCIe traffic shown in Figure 3(a).

The issue intensifies with smaller values. We assessed the Traffic Amplification Factor (TAF), the ratio of PCIe traffic to the size of the requested data, with value sizes set at 32, 64, 128, 256, 512 bytes, and 1 KB. As depicted in Figure 3(b), each transfer consistently sends a 4 KB memory page, significantly amplifying the data transfer volume. For instance, transferring a 32-byte value generates around 4 KB of traffic, roughly 130 times the size of the data requested.

NAND Write I/O Amplification: We measured write response times and the number of written NAND pages using the same workloads as the previous experiments, with results shown in Figure 4(a). Write response times were over 10 times longer than transfer response times, mainly due to NAND flash I/O times. Surprisingly, these extended NAND flash I/O durations did not diminish the spikes at 4 KB boundaries observed in prior experiments but rather exacerbated them, leading to more severe page-unit cascades. The count of written NAND pages, also depicted in Figure 4(a), shows a significant increase at each memory page boundary.

We assessed the Write Amplification Factor (WAF), defined as the ratio of the data written to NAND flash to the size of the received value. The results, shown in Figure 4(b), reveal that WAF closely mirrors TAF, despite including non-value-related NAND writes like in-device LSM-tree compaction. This suggests that the amplification due to memory page alignment during transfers also extends to writing values from device memory to NAND flash.

2.5 Exploring Escape-from-Block Strategies

The NVMe protocol currently offers two data transfer methods: the PRP and Scatter-Gather List (SGL) [24]. As the PRP list describes the payload in host memory in a list of physical pages, it has an inherent limitation in describing byte granular key and value pairs. In addition to this, the assumption that the payload is multiple blocks guided the NVMe storage stack to be optimized for block-size transfer from memory allocations for DMA in the host to the DMA engine within the device. On the other hand, SGL can support multiple variable-sized DMAs across scattered memory segments. However, it has been reported that the cost of enabling the SGL outweighs the benefit for I/O smaller than 32 KB [1]. Consequently, the Linux kernel establishes a minimum threshold for data transferred via SGL at 32 KB [32], indicating that using SGL for small value transfers is not advisable.

Given these circumstances, we have focused on utilizing NVMe commands to escape the key-value transfer from the traditional block-based payload transfer mechanism. Considering that the typical size of values from real-world KVS workloads rarely exceeds a hundred bytes and is often below 64 bytes [3], NVMe command (64 bytes [24]) has an opportunity to cover a significant portion of value transfer in real-world workload. In other words, we can employ NVMe commands to make fine-grained and bandwidth-efficient key-value pair transfers.

To escape from the block-based NAND page buffer management, which amplifies NAND write I/Os when handling variable-sized values, KAML [15] proposed the batching for multiple values and

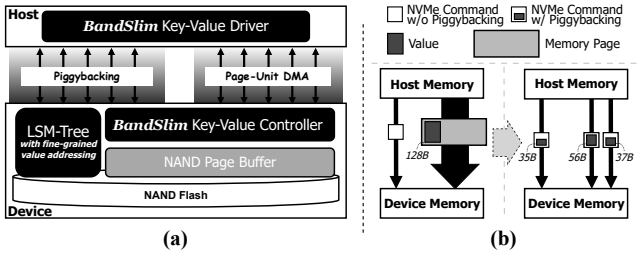
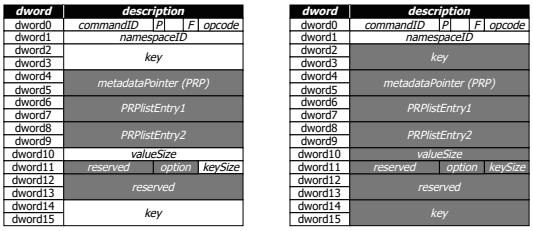


Figure 5: (a) Software architecture overview of *BandSlim*, and (b) Piggybacking values onto NVMe commands.



dword	description			
dword0	commandID	P	F	opcode
dword1	namespaceID			
dword2	key			
dword3				
dword4	metadataPointer (PRP)			
dword5				
dword6	PRPlistEntry1			
dword7				
dword8	PRPlistEntry2			
dword9				
dword10	valueSize			
dword11	reserved	option	keySize	
dword12	reserved			
dword13				
dword14	key			
dword15				

(a) Write Command

dword	description			
dword0	commandID	P	F	opcode
dword1	namespaceID			
dword2	key			
dword3				
dword4	metadataPointer (PRP)			
dword5				
dword6	PRPlistEntry1			
dword7				
dword8	PRPlistEntry2			
dword9				
dword10	valueSize			
dword11	reserved	option	keySize	
dword12	reserved			
dword13				
dword14	key			
dword15				

(b) Transfer Command

Figure 6: NVMe key-value write and transfer command in *BandSlim* features values piggybacked in gray-colored fields.

stored them at the NAND page level in a log-fashion. However, the design for efficiently packing values that are smaller than the block size was not detailed enough to adapt LSM-tree-based KV-SSDs. Moreover, some DMA engines in storage devices, including our testbed, require that the transfer size and destination addresses be page-aligned [20]. Consequently, device drivers are typically designed to accommodate this requirement [26, 31]. Thus, the simple buffering approach from KAML can necessitate excessive memory copies when handling large values. To address this, we have devised a strategy where large, infrequent values are sent to the device using a traditional method, while the smaller, more common values are interspersed between them within NAND pages.

3 DESIGN OF *BANDSLIM*

3.1 Software Architecture Overview

Figure 5(a) illustrates the software architecture of *BandSlim*. The modules in black are key components of *BandSlim*.

- *BandSlim Key-Value Driver:* This component piggybacks values onto NVMe commands, achieving PCIe traffic generated close to the requested value size (§3.2).
- *BandSlim Key-Value Controller:* This component packs received values into the NAND page buffer entry in a fine-grained manner. It features the *Selective Packing with Back-filling Policy* to address the memory copy overheads that occur when packing relatively large values transferred via PRP-based DMA (§3.3).
- *LSM-tree with Fine-Grained Value Addressing:* A key-value separated LSM-tree storage engine featuring fine-grained value addressing over the vLog (§3.4).

3.2 Fine-grained Value Transfer over NVMe

To enable the fine-grained value transfer over NVMe via NVMe-command-piggybacking, we reviewed the structure of the NVMe command. As shown in Figure 6(a), if traditional transfer methods are not used, `dword4-9` (24 bytes), which were originally designated for specifying memory pages for DMA transfers, can be repurposed for piggybacking values. Additionally, the 8 bytes of reserved `dword12-13` can also be used for piggybacking. Plus, the reserved 2 bytes and 1-byte for specifying vendor-specific options in `dword11` can be used as well. Thus, a maximum of 35 bytes can be repurposed. *BandSlim* utilizes these fields for piggybacking and inline shipping of values by embedding them into these fields.

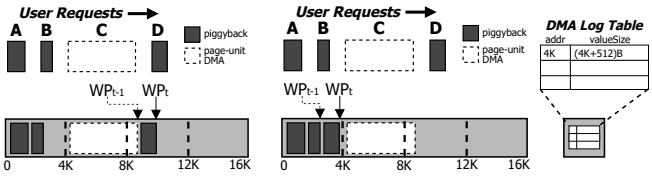
To handle values that cannot be covered by a single NVMe command submission, we classify piggybacking commands into two types using separate opcodes: (i) `write command` and (ii) `transfer command`. The former serves as the initial command. It contains key and metadata like key size, value size, and more, similar to the original key-value write command. It allows piggybacking of values of up to 35 bytes. The latter is a command solely intended for transferring the remaining bytes of the value. As depicted in Figure 6(b), since the key and value size are already sent to the device through the first command, `write command`, we can utilize all fields of `transfer command` for piggybacking, except for essential fields such as opcode. Consequently, all the remaining `dwords` (56 bytes) are utilized for piggybacking in `transfer command`.

Therefore, when *BandSlim* transfers a key-value pair and cannot send the entire value in a single command submission, it uses the `transfer command` as trailing commands to deliver the remainders to the device in 56-byte increments. Figure 5(b) illustrates the transfer of value with a size of 128 bytes using the piggybacking transfer. It requires 3 NVMe commands (total 192 bytes) to transmit the value. With the need to transmit at least one command as well, compared to the traditional approach that generated 4 KB traffic for value transfer, we can reduce approximately 78.4% of the traffic.

Adaptive Value Transfer Method. As the value size increases, NVMe-command-piggybacking-based value transfer can involve a significant number of `transfer commands`. In such cases, due to the accumulation of overheads in generating NVMe commands and synchronously handling them within the device, the transfer time becomes inferior compared to the conventional PRP-based value transfer (§4.2). Furthermore, when the value size marginally exceeds the memory page size, for example, (4K+32) bytes, employing a hybrid approach can be more efficient than processing it solely through piggybacking or PRP-based page-unit DMAs. We can transfer the first 4 KB via page-unit DMA and the remaining 32 bytes via piggybacking on `transfer command`. Even though *BandSlim* primarily targets workloads with small value sizes, it must also be capable of effectively handling values of exceptional, large sizes.

To tackle these issues, *BandSlim* utilizes a threshold-based reactive method that selects the most suitable transfer method from NVMe-command-piggybacking-based, PRP-based, and hybrid transfers based on the size of the value. This decision-making process is supported by exploratory runs conducted using synthetic benchmarks, where users identify thresholds and configure *BandSlim* accordingly. To facilitate this, *BandSlim* provides benchmarks for determining these thresholds (§4.1). During the benchmark runs, various value sizes ranging from 4 bytes to 8 KB are tested through millions of PUT commands to compare transfer times.

We establish two key performance thresholds from these tests: (i) $\alpha \times threshold_1$ marks the size at which piggybacking becomes



(a) w/o Backfilling **(b) w/ Backfilling**

Figure 7: Description of *Selective Packing Policy* and *Selective Packing with Back filling Policy*.

less efficient than PRP-based transfers. (ii) $\beta \times threshold_2$ identifies where PRP-based transfers outperform hybrid transfers for values sizes slightly greater than multiples of 4 KB. These thresholds are dynamically adjusted using coefficients α and β, which are scaled based on user preferences for reducing PCIe traffic. For users prioritizing response time, both α and β can be set to 1. For those valuing traffic reduction, α and β can be increased to favor piggybacking and hybrid methods. Based on this approach, *BandSlim* ensures efficient handling of value sizes ranging from sub-page to large.

3.3 In-device Fine-grained Value Packing

3.3.1 ***All Packing Approach and Its Limitations.*** To design an efficient and fine-grained packing policy for the NAND page buffer management of KV-SSDs, we reviewed the simple buffering approach mentioned in KAML [15], which we will refer to as an *All Packing Policy*. We introduced a Write Pointer (WP) to track the current write offset within the NAND page buffer and implemented the *All Packing Policy*. For values transferred via NVMe-command-piggybacking, the controller fetches commands and extracts the value from the piggybacking fields. Then, it performs a memory copy of the extracted value to the address pointed by the WP, and updates the WP. Trailing `transfer commands` are processed in the same manner, sequentially updating the WP and packing the piggybacked value fragment in the buffer. On the host side, the driver submits `transfer commands` to the submission queue where the `write command` for that value was inserted, ensuring that the piggybacked value fragments are processed in FIFO order.

For values transferred via page-unit DMA or a hybrid approach[1], packing them into the NAND page buffer requires a memory copy due to the aforementioned page alignment restriction of DMA destination addresses (§2.5). When the controller executes page-unit DMA, the destination address should be set, for instance, to the closest page-aligned address following the current WP. If the WP and the destination address coincide, we can skip the memory copy. If not, these values are memory copied to the WP in the same manner as piggybacked values are processed.

3.3.2 ***Selective Packing Policy.*** The *All Packing Policy* aims to minimize NAND page writes as much as possible. However, since values transmitted via page-unit DMA often exceed average size due to adaptive methods, excessive memory copying can lead to overheads. Given the resource constraints of storage devices, large memory copies can significantly slow down operations and delay other requests. To tackle these challenges, we present a *Selective Packing Policy*. When the controller receives a value transmitted via page-unit DMA, it updates the WP to the end of the value, similar to the traditional approach, without performing packing. It is particularly predicated on the assumption that, under real-world KVS workloads, requests to write large values are rare. Even at the expense of some spatial loss, this method can be valid if the overhead of copying large values exceeds that loss. For requests transmitted via piggybacking, values are still packed. In other words, the *Selective Packing Policy* only packs values transmitted via piggybacking.

Figure 7(a) illustrates this policy. The figure assumes a scenario with four mixed key-value write requests of small and large values, labeled *A*, *B*, *C*, and *D*. Requests *A*, *B*, and *D* transfer values using the piggybacking method. In contrast, request *C* transfers its value via page-unit DMA. Under the *Selective Packing Policy*, values of requests *A* and *B* are compactly packed following the WP, while the value of *C* is positioned at the next page-unit boundary closest to the WP. The WP is then updated to the address following the end of *C*'s value, and *D*'s value is packed at this updated location.

3.3.3 ***Selective Packing with Backfilling Policy.*** In workloads where small values are dominant, the *Selective Packing Policy* can exhibit equivalent performance to the *All Packing Policy*. However, if the workload occasionally involves values transferred via page-unit DMA, this can lead to internal fragmentation within NAND pages. Therefore, we present the *Selective Packing with Backfilling Policy*, aiming to address both the issue of NAND page internal fragmentation and the memory copy issue for packing large values.

Figure 7(b) depicts this policy. The process up to handling request *C* is the same as the previous one. However, in here, the WP is not updated after receiving the DMA-transferred value. Instead, it introduces backfilling, where the value of request *D* continues to be packed at the original WP, filling up the empty space. To do this, an additional data structure is needed to track values transferred via page-unit DMA, to ensure the current WP avoids these values. Thus, we introduce a DMA Log Table (DLT) in a separate space of device memory from the NAND page buffer, where it records the destination address and value size upon each page-unit DMA operation. The DLT is a circular queue, the head of which always points to the oldest unconsumed entry, moving to the next oldest once consumed. Whenever the controller packs a value transferred via piggybacking, it checks if the current WP plus the current value size exceeds the oldest DLT entry's destination address among the unconsumed entries. This reference process has a time complexity of O(1). If the WP exceeds it, the controller sets a new address by adding the DMA-transferred value size to the destination address, updates the WP to that address, and repeats the process.

However, the DLT necessitates extra memory space, which could increase the design costs. To save space, *BandSlim* records only the logical NAND page number and memory page offset instead of the full address, reducing the bit count needed. For instance, a 1 TB NAND space with 16 KB page size requires only (26+2) bits instead of 40. Plus, given that NAND page buffer entries are limited, we capped the maximum number of entries to match (e.g., 512 entries). If we allocate 4 bytes for specifying value size in each DLT entry, the upper bound of the size of memory space for DLT is only 4 KB.

3.4 Fine-grained Value Addressing over vLog

The fine-grained value packing necessitates a byte-level addressing over the vLog, increasing the bit size required for addressing fields

[1]Hereafter, we will refer to both transfer modes collectively as page-unit DMA.

Table 1: HW/SW 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 2: HW/SW specifications of the host node.

CPU	Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz (32 cores)
Memory	384GB DDR4
OS	Ubuntu 22.04

in the LSM-tree. This increase in bit size for vLog addressing fields seems significant, especially since SSDs typically strive to minimize space usage due to their limited memory capacity, which often leads to reduced bit sizes in addressing fields [17]. However, even though the size of MemTable increases, it remains constant due to LSM-tree flushes and resets. Despite the slight increase in memory requirements, we believe that this is a reasonable compromise for the advantages that fine-grained value packing offers, such as more efficient space utilization and improved performance of KV-SSDs.

4 EVALUATION

4.1 Evaluation Setup

We implemented *BandSlim* by extending the state-of-the-art NVMe KV-SSD [22] based on Cosmos+ OpenSSD platform [19]. The partial source code for the *BandSlim* key-value driver and controller is made publicly accessible[2], with the intention of promoting active development within the field of KV-SSDs. The SoC of the platform operates the *BandSlim* controller, the PCIe interface controller, the DRAM controller, and the NAND flash controller. The host system runs the *BandSlim* driver. Table 1 and Table 2 present the HW/SW specifications of our setup.

For performance evaluations, we slightly modified db_bench [12], a widely recognized benchmarking tool used in RocksDB [10]. We enabled db_bench to send NVMe key-value commands to the Cosmos+ OpenSSD platform through the NVMe passthrough.

We conducted various patterns of the workloads to verify our proposed design. The description of the workloads is as follows.

- ***Workload A*:** This is db_bench's *fillseq* for a write pattern where the key is sequential. The value size does not change.
- ***Workload B* ($W(B)$):** This writes 1 million random key-value pairs with value sizes of 8 bytes or 2 KB at a 9:1 ratio.
- ***Workload C* ($W(C)$):** This is similar to $W(B)$, but with the value size ratio reversed to 1:9 for 8 bytes and 2 KB values.
- ***Workload D* ($W(D)$):** This workload writes values of sizes (8, 16, 32, 64, 128, 256, 512 bytes, 1 KB, and 2 KB) in random order, totaling 1 million, with each size having an equal ratio.
- ***Workload M* ($W(M)$):** db_bench's *mixgraph All_random* [3]. It reflects real-world characteristics with a maximum value size of 1 KB and almost 70% of values being under 35 bytes. We have modified *mixgraph* to issue only 1 million PUTs.

In all experiments, we used 4-byte unique keys generated by a hash function with a random seed. The keys were inserted into the corresponding fields of the NVMe command (see Figure 6).

We conducted evaluations for the following designs.

- *Baseline:* the state-of-the-art NVMe KV-SSD [22]. It employs the PRP-based page-unit value transfer and NAND page buffer management from NVMe SSDs. We clarify that other state-of-the-art systems like KV-CSD [27], also follow this.
- *Piggyback:* transfers values only through piggybacking.
- *Hybrid:* transfers values using only hybrid transfer method.
- *Adaptive:* transfers values using the adaptive method.
- *Packing:* maintains transfers using only PRP-based page-unit DMAs, while performing fine-grained value packing.
- *Piggy+Pack:* transfers values using only piggybacking method and performs fine-grained value packing.
- *Block, All, Select, and Back II:* To compare different packing policies, they represent the baseline, *All Packing*, *Selective Packing* and *Selective Packing with Backfilling* in order.

[2]https://github.com/lass-lab/bandslim

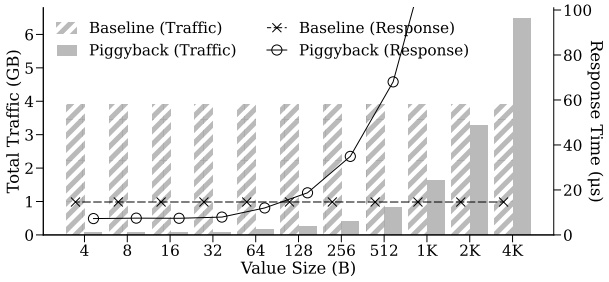


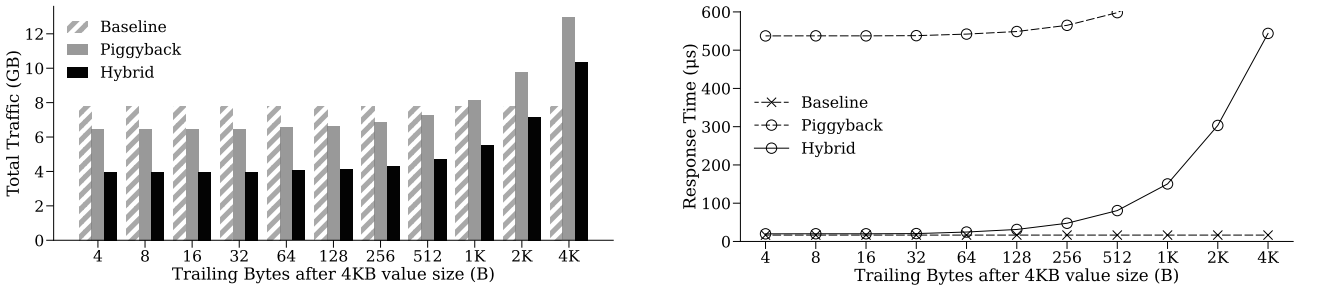
Figure 8: Measurement of Total PCIe Traffic with Average Response Time for varying value sizes.

4.2 Effects of Fine-grained Value Transfer

We disabled NAND I/O to demonstrate the effects of fine-grained value trasnfer. We ran the *Workload A* for 1 million unique key-value pairs across various value sizes. Additionally, to see the impact of fine-grained value transfer on performance, we also measured the average transfer response time.

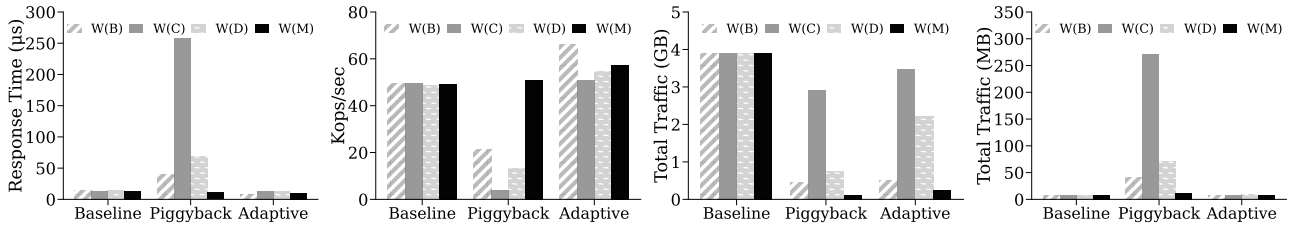
Figure 8 shows the results of traffic measurements. As expected, the *Baseline*, which performs page-unit DMA, shows the same PCIe traffic for all value sizes below one memory page size (4 KB). The *Piggyback*, on the other hand, dramatically reduces the PCIe traffic imposed on the interconnect for small values under 1 KB in size. For example, in the 4 bytes to 32 bytes cases, we can observe that *Piggyback* reduces traffic by up to 97.9% compared to the *Baseline*. Meanwhile, as the value size increases with piggybacking applied, the network traffic begins to increase due to the addition of trailing commands. NVMe command submission involves not only a 64-byte entry but also doorbell ringing by the driver for notification, and tail pointer read by the controller for submission queue fetch [24]. On the other hand, PRP-based DMA involves only one submission and tail pointer read, and doorbell ringing, with the rest being data transfer traffic via page-unit DMA. Therefore, while the *Piggyback* shows significantly less traffic for smaller value sizes, as the value size increases, the addition of multiple trailing commands leads to an increase in traffic. This increase continues until the total traffic approaches the *Baseline* at 2 KB and then exceeds it at 4 KB.

Figure 8 also shows the response time results. The *Piggyback* shows a response time that is approximately a half of the *Baseline* for cases of 32 bytes and below. However, as observed in the PCIe traffic, for the 64 bytes case, the transfer response time becomes almost identical due to the addition of trailing commands. For cases of 128 bytes and above, where more trailing commands are attached,



(a) Total PCIe Traffic (b) Average Response Time

Figure 9: Measurement and analysis of PCIe traffic and response times for *Workload A* of varying sizes of 1 million key-value pairs from host memory to device memory, where the value size is 4 KB with additional trailing bytes ranging from 4B to 4 KB.



(a) Average Response Time (b) Average Throughput (c) Total PCIe Traffic (d) Total Host MMIO Traffic

Figure 10: Performance analysis of transfer methods including the adaptive value transfer through *Workloads B, C, D*, and *M*

a significant performance degradation can be observed. Despite the PCIe traffic being much reduced compared to the *Baseline*, the reason for such results starting from 128 bytes is because the transmission of NVMe commands in our implementation is synchronous and serialized. In our environment, when a command is submitted by the driver, an NVMe passthrough is used, which mandatorily handles only one command at any given time. Thus, no subsequent commands can be sent until the controller signals completion. This results in round-trip overhead, significantly reducing performance.

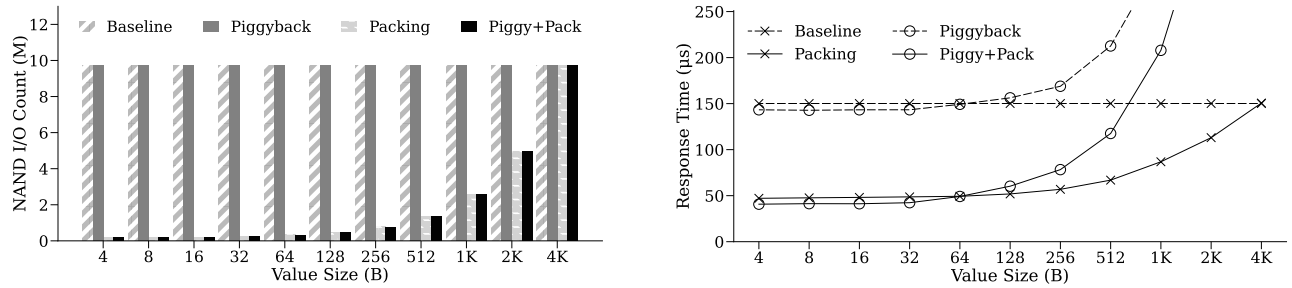
Effects of Hybrid Value Transfer. To examine the effectiveness of hybrid transfer for adaptive approach, we set the value sizes from (4K+4) bytes to (4K+4K) bytes, doubling the trailing bytes from 4 to 4 KB after 4 KB, and conducted the same experiments with *Workload A*. Figure 9(a) shows the PCIe traffic. *Hybrid*, which transfers the first memory page using page-unit DMA and the rest via piggybacking, proves to be the optimal choice in terms of traffic up to 6 KB among the three. In the case of *Piggyback*, it uses less traffic than *Baseline*, which transfers two memory pages, up to 1 KB, but shows a sharp increase thereafter, as observed in Figure 8.

Figure 9(b) shows the transfer response times. As expected, the response time for *Piggyback* is significantly worse. The response time of *Hybrid* shows only a slight decrease compared to *Baseline* from (4K+4) bytes to (4K+64) bytes, but it is still lagging behind (1.4% lower at maximum). Thus, while *Hybrid* significantly reduces PCIe traffic compared to *Baseline*, it does not improve performance. However, if reducing traffic is a priority for the user, this can be considered highly effective. This could be reflected by setting the coefficient β greater than 1. However, we set all coefficients for adaptive value transfer used in the subsequent experiments to 1.

Effects of Adaptive Value Transfer. We analyzed the performance effects from enabling adaptive value transfer, using *Workload B, C, D* and *M*. The $W(B)$ represents situations with a dominance of small values, and the $W(C)$ represents a dominance of large values. The $W(D)$, on the other hand, represents a balanced presence of varying sizes of values. The $W(M)$, as explained, represents a real-world pattern of KVS workload. We compared the *Baseline* with *Piggyback* and *Adaptive*. In *Adaptive*, the transfer method shifts from piggybacking to page-unit DMA at 128 bytes. This threshold is set based on the transfer response time results shown in Figure 8.

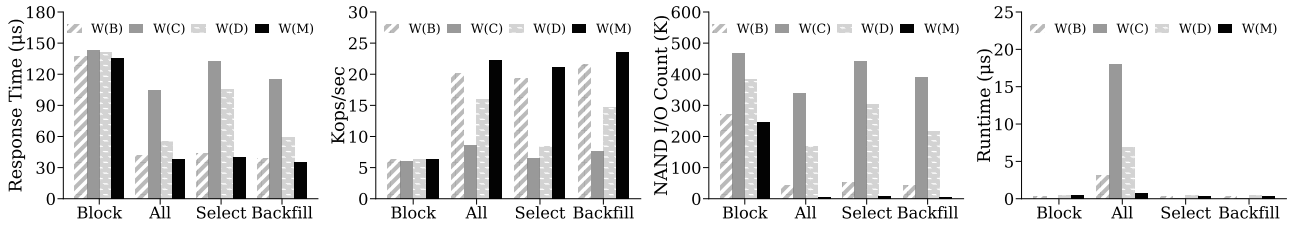
Figure 10(a) and Figure 10(b) show the performance comparison of the three transfer methods for each workload. *Piggyback* shows the worst performance across workloads *B*, *C* and *D*. Especially in the large-value-dominant $W(C)$, performance of *Piggyback* drastically deteriorates. The *Baseline* shows better performance under these workloads. This can be explained by the overhead created by trailing commands for transferring exceptional, large values. Surprisingly, however, *Piggyback* improved response time by about 22% compared to *Baseline* for $W(M)$, and improved throughput as well. This demonstrates that *Piggyback* alone can achieve improved performance compared to *Baseline* in real-world scenarios. In any situation, however, *Adaptive* proves to be the best in all workloads. In $W(D)$, where small and large values are evenly mixed, or $W(B)$, where small values are dominant, *Adaptive* shows a significant performance improvement over the other two methods.

Figure 10(c) shows the results of PCIe traffic measurements. As expected, *Piggyback* reduces PCIe traffic the most. In the case of $W(M)$, it achieves a dramatic 97.9% reduction in PCIe traffic compared to *Baseline*. In contrast, *Adaptive* slightly sacrifices PCIe traffic reduction in favor of performance improvement. In the case of $W(M)$, for example, it achieves a 93.3% reduction in traffic but shows a 12% improvement in throughput compared to *Piggyback*. For $W(C)$, *Adaptive* generates 18% more traffic than *Piggyback*, but increases the throughput by nearly 13 times, and improves the processing rate by about 2% compared to *Baseline*.



(a) Total Number of NAND I/Os (b) Average Response Time

Figure 11: Measurement and analysis of NAND I/O page counts and response times for *Workload A* of varying sizes of 10 million key-value pairs from host memory to NAND flash. The *All Packing Policy* was applied in this case.



(a) Average Response Time (b) Average Throughput (c) Total Number of NAND I/O (d) Average Memcpy Time

Figure 12: Performance analysis of in-device packing policies. The driver transfers values using the adaptive value transfer.

Figure 10(d) shows only the Memory-Mapped I/O (MMIO) traffic generated out of the PCIe traffic presented in Figure 10(c). The MMIO traffic represents the total number of bytes transmitted to the device each time the host driver rings the doorbell for NVMe command submission. An increase in the host's MMIO traffic implies more engagement of the host CPU. The host CPU keeps accessing the PCIe address space while transferring values to the device by piggybacking. As a result, the *Piggyback* shows a significant amount of MMIO traffic as the value size increases ($W(C)$). On the other hand, the *Baseline* maintains a constant amount of MMIO regardless of the workload, as it always exchanges data with a single command. This again demonstrates the need for an adaptive approach in fine-grained value transfer if the target workload can frequently involve values larger than the $threshold_1$.

4.3 Effects of Fine-grained Value Packing

To evaluate the effects of fine-grained value packing, we enabled NAND I/O this time. We ran *Workload A* for 10 million unique key-value pairs across various value sizes. In this experiment, we used the *All Packing Policy*. We measured the NAND page I/O count. Figure 11(a) shows the results. Compared to the *Baseline*, *Packing* and *Piggy+Pack* showed dramatically reduced I/O counts for small values. In the case of 4 bytes to 32 bytes, packing reduced NAND writes by 98.1%. Note that *Selective Packing* can achieve the same effects in these cases. This reduction in NAND write counts has a significant impact on performance. Figure 11(b) shows the write response time. Regardless of which transfer mode was used, the application of fine-grained packing significantly reduced the response time. For example, at 32 bytes, the response time was reduced by 67.6%. In cases where the piggybacking transfer was also applied, an additional reduction of about 4.2% in response time was observed at 32 bytes case. However, similarly, due to the serialization, the response time of *Piggy+Pack* increases sharply from 128 bytes onwards since it transfers values using only piggybacking.

Next, we conducted a comparative analysis of write performance according to the packing policy. The host driver used the adaptive value transfer method. We used *Workloads B, C, D* and *M*. Figure 12(a) and Figure 12(b) compare the average response time and throughput when applying each packing policy for the workloads. The baseline, *Block*, shows the worst performance regardless of the workload. Meanwhile, the *Selective Packing Policy* performs as poorly as *Block* in large-value-dominant situations ($W(C)$). This drop is due to this policy's adherence to page alignments to avoid memory copy. Similarly, the *Selective Packing with Backfilling Policy* also experiences diminished performance in $W(C)$ due to the constrained size of the in-device NAND page buffer. When large values arrive via page-unit DMA, as in $W(C)$, and these values are smaller than 4 KB (2 KB of $W(C)$ for example), fragmentation occurs.

The same rationale applies to the results observed in $W(D)$. In situations where values of considerable size, yet way smaller than the closest multiple of memory pages (2 KB for example), occur frequently or abundantly, the *All Packing Policy* emerges as the most optimal among the four. However, in scenarios where small values predominate, such as in $W(B)$ or $W(M)$, the throughput of the *Selective Packing* dips by at most 4.5% compared to the *All Packing* which instead requires higher in-device CPU engagements (will be discussed). Furthermore, the *Selective Packing with Backfilling* showcases the most optimal performance across both $W(B)$ and $W(M)$. It achieved a processing rate up to approximately 7% higher than *All Packing*. The backfilling policy performs less efficiently than the *All Packing* in workloads containing a significant number of large-sized values that need to be transmitted via page-unit DMA, as it incurs performance degradation due to the lower NAND space utilization (see Figure 12(c)). However, in real-world workloads

dominated by significantly small values, such as $W(M)$, it efficiently stores occasional large values without memory copying, thus being the most optimal approach. The *Selective Packing* can also achieve performance equivalent to the *All Packing* in such workloads.

Figure 12(d) shows the average memory copy time. It is true that despite all other packing policies inherently involving memory copying when handling piggybacked values, the *All Packing Policy*, which copies memory for all large values, shows a significantly higher memory copy time. Especially, as the proportion of large values increases, i.e., in the order of $W(M)$, $W(B)$, $W(D)$, $W(C)$, the memory copy time also increases. In situations where there are numerous large-sized values that need to be transmitted via page-unit DMA, it is indeed a rational choice to opt for the *All Packing Policy*, as the memory copy overhead becomes inevitable without employing DMA engines that can support non-page-aligned access to the device memory. However, in workloads where such large values occur exceptionally rarely, we can optimize without incurring this memory copy overhead by employing a selective approach. It is important to note that we can design a controller that effectively adapts to any workload by integrating the strengths of both.

5 CONCLUSION

In this paper, we introduce *BandSlim*, a solution designed to address the incompatibilities between traditional block-interfaced storage protocols (e.g., NVMe) and the new key-value interface of KV-SSDs. This mismatch leads to excessive traffic on the PCIe interconnect and amplified NAND write I/Os, significantly degrading performance. *BandSlim* effectively resolves these issues by enabling a fine-grained value transfer and efficient in-device value packing. Our evaluations demonstrate that *BandSlim* significantly reduces network traffic during value transfers by up to 97.9% and reduces NAND page write counts by up to 98.1%, compared to the state-of-the-art NVMe-based KV-SSD without employing *BandSlim*.

ACKNOWLEDGMENTS

This work was funded in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT) (No. NRF-2021R1A2C2014386), in part by the Institute of Information & Communications Technology Planning Evaluation (IITP) grant funded by the Korea Government (MSIT) under Grant 2021-0-00528, and in part by an SK hynix research grant.

REFERENCES

[1] 2017. nvme : add Scatter-Gather List (SGL) support in NVMe driver. https://lore.kernel.org/all/04aaed5c-1a8a-f601-6c9c-88bf1cf66e8a@mellanox.com/T/.
[2] Byrne, John and Chang, Jichuan and Lim, Kevin T. and Ramirez, Laura and Ranganathan, Parthasarathy. 2011. Power-efficient networking for balanced system designs: early experiences with PCI. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems (HotPower)*. 1–5.
[3] Hichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 209–224.
[4] Cao, Wei and Liu, Yang and Cheng, Zhushi and Zheng, Ning and Li, Wei and Wu, Wenjie and Ouyang, Linqiang and Wang, Peng and Wang, Yijing and Kuan, Ray and Liu, Zhenjun and Zhu, Feng and Zhang, Tong. 2014. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. 29–41.
[5] Zhiguang Chen, Fang Liu, and Yimo Du. 2011. Reorder the Write Sequence by Virtual Write Buffer to Extend SSD's Lifespan. In *Proceedings of the 8th Network and Parallel Computing (NPC)*. 263–276.
[6] Wonil Choi, Jie Zhang, Shuwen Gao, Jaesoo Lee, Myoungsoo Jung, and Mahmut Kandemir. 2016. An in-depth study of next generation interface for emerging non-volatile memories. In *Proceedings of the 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*.
[7] Chun-Ying Chung, Jaehoon Koo, Jaeho Im, Seungseob Lee, and et al. 2019. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 939–953.
[8] Dan Helmick. 2022. Optimal Performance Parameters for NVMe SSDs. https://www.snia.org/educational-library/optimal-performance-parameters-nvme-ssds-2022
[9] Carl Duffy, Jaehoon Shim, Sang Hoon Kim, and Jin Soo Kim. 2023. Dotori: A Key-Value SSD Based KV Store. In *Proceedings of VLDB Endowment*, Vol. 16.
[10] Facebook. 2014. RocksDB. http://rocksdb.org.
[11] Facebook. 2017. LevelDB. https://github.com/google/leveldb.
[12] Facebook. 2021. *RocksDB Database Benchmark Tool.* https://github.com/facebook/rocksdb/wiki/Benchmarking-tools
[13] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.
[14] Intel. 2010. Performance Counter Monitor. https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html.
[15] Yanqin Jin, Hung Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 373–384.
[16] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. 2006. FAB: Flash-aware Buffer Management Policy for Portable Media Players. *IEEE Transactions on Consumer Electronics (TCE)* 22, 2 (Dec 2006).
[17] Dawoon Jung, Jeong-Uk Kang, Heeseung Jo, and Jin-Soo Kim. 2010. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation. *ACM Transactions on Embedded Computing Systems* 9 (2010), 1–41. Issue 4.
[18] Hyojun Kim and Seongjun Ahn. 2008. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*.
[19] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage*, Article 15 (July 2020), 35 pages.
[20] Won-ok Kwon, Song-Woo Sok, Chan-ho Park, Myeong-Hoon Oh, and Seokbin Hong. 2022. Gen-Z memory pool system implementation and performance measurement. *ETRI Journal* 44 (2022), 450–461. Issue 3.
[21] Chang Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. 2019. iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics. In *Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.
[22] Seungjin Lee, Chang Gyu Lee, Donghyun Min, Inhyuk Park, Woosuk Chung, Anand Sivasubramaniam, and Youngjae Kim. 2023. Iterator Interface Extended LSM-tree-based KVSSD for Range Queries. In *Proceedings of the 16th ACM International Systems and Storage Conference (SYSTOR)*.
[23] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Wisckey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the File and Storage Technologies (FAST)*. USENIX, 133–148.
[24] NVM Express Inc. 2011. NVM Express Specification. https://nvmexpress.org/developers/nvme-specification/.
[25] NVM Express Inc. 2021. NVM Express Key Value Command Set Specification. https://nvmexpress.org/developers/nvme-specification/
[26] Oracle. 2010. Designing Device Drivers for the Oracle Solaris Platform. https://docs.oracle.com/cd/E19120-01/open.solaris/819-3196/dma-29901/index.html
[27] Inhyuk Park, Qing Zheng, Dominic Manno, Soonyeal Yang, Jason Lee, David Bonnie, Bradley Settlemyer, Youngjae Kim, Woosuk Chung, and Gary Grider. 2023. KV-CSD: A Hardware-Accelerated Key-Value Store for Data-Intensive Applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 132–144.
[28] Joyce Russell. 2018. Efficient and predictable high-speed storage access for real-time embedded systems. *EngD thesis, University of York*. (2018).
[29] Samsung. 2017. Samsung Key-Value SSD Enables High Performance Scaling.
[30] SK hynix. 2023. Accelerating Data Analytics Using Object Based Computational Storage in a HPC.
[31] The Linux Kernel documentation. 2020. Dynamic DMA mapping Guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt
[32] The Linux Kernel source code. 2024. sgl_threshold. https://github.com/torvalds/linux/blob/master/drivers/nvme/host/pci.c.