Maximizing Interconnect Bandwidth and Efficiency in Nonvolatile Memory, Express-Based Key-Value Solid-State Devices With Fine-Grained Value Transfer

Junhyeok Park, Chang-Gyu Lee, and Soon Hwang, *Sogang University, Seoul, 04107, South Korea*

Seung-Jun Cha, *Electronics and Telecommunications Research Institute, Daejeon, 61012, South Korea*

Woosuk Chung, *Memory Systems Research, SK hynix, Seongnam, 13558, South Korea*

Youngjae Kim, *Sogang University, Seoul, 04107, South Korea*

The key-value solid-state drive (KV-SSD) redefines storage interfaces by integrating a key-value store directly within the device, offering native support for non-page-aligned key-value pairs. This architectural innovation enables KV-SSDs to offload storage management from the host system, positioning them as ideal candidates for resource disaggregation. However, KV-SSDs face significant challenges, notably, input–output amplification caused by conflicts with traditional storage protocols like Non-Volatile Memory Express (NVMe), which are designed around memory page units. Specifically, this results in a substantial increase in data traffic over interconnect between the host and SSD. This article introduces BandSlim, a novel solution addressing these challenges in data transfer by leveraging 1) NVMe command piggybacking for universally compatible, fine-grained transfers without requiring interconnect-level support and 2) the remote memory access capabilities of emerging Compute eXpress Link interconnects for higher-performance fine-grained transfers. Through extensive evaluations, BandSlim achieves up to a 97.9% reduction in Peripheral Component Interconnect Express traffic compared to conventional NVMe KV-SSDs.

Designing an efficient storage system necessitates reducing data movement costs from the host's memory to the storage media. However, traditional key-value stores (KVSs) like RocksDB[1] function as middleware on top of file systems. Consequently, user input–output (I/O) requests must navigate through the kernel's file system and block layers to execute data reads and writes to storage. This multilayer traversal incurs significant memory copying and kernel context switch overheads during I/O operations. In contrast, key-value solid-state drives (KV-SSDs) offer a substantial reduction in these overheads. KV-SSDs implement KVSs at the device controller level and provide native support for key-value operations. This allow users to bypass the file system and block layer when processing I/O requests, enabling lower latency and higher throughput compared to traditional SSD-based host-side KVSs. Such KV-SSDs are well suited for modern resource disaggregation architectures as they can effectively decouple storage management from the host system.[2]

KV-SSDs hold significant potential as next-generation storage devices but still face critical challenges,

Digital Object Identifier 10.1109/MM.2025.3572475
Date of publication 23 May 2025; date of current version 24 December 2025.

with the biggest issue being I/O amplification caused by limitations in existing protocols. Commercially and academically released KV-SSDs[3,4,5,6] utilize the Non-Volatile Memory Express (NVMe) protocol,[7] 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 that originates from the size difference between the block and the key value. This results in a substantial increase in data traffic over the interconnect between the host and the SSD.

One common solution to this problem has been host-side batching, where enough key-value pairs are grouped to align with the memory page size, as implemented in KV-SSDs like KV-CSD[4] through bulk PUT operations. However, buffering entries on the host side risks data loss during power failure, making this approach unsuitable for mission-critical applications where data integrity is essential. For those scenarios where data persistence is critical and I/O transactions occur at the key-value pair level as defined by NVMe standard, a more fundamental solution is required.

To address these issues, we introduce *BandSlim*, which minimizes data traffic over host–SSD interconnect when transferring small key-value pairs. *BandSlim* employs two approaches: the first utilizes NVMe commands to achieve an inline value transfer at the storage protocol level, while the second leverages the emerging Compute eXpress Link (CXL) interconnect to address the transfer at the interconnect protocol level. The first approach piggybacks values smaller than a memory page to NVMe commands using reserved fields. We observed that this method significantly reduces data traffic over the host–SSD interconnect. The second approach, which uses CXL's remote memory access capability, aims to improve this further by offering a more fine-grained, higher-performance value transfer solution, bypassing the limitations of piggybacking. It exposes the device-side NAND page buffer to the host's CPU and transfers values using CXL.mem operations. The first method can be implemented directly in existing NVMe protocols without requiring new interconnect support, while the second approach is optimized for systems that support CXL, allowing for more advanced and efficient fine-grained transfers.

In both methods, however, as the value size grows, the performance of fine-grained transfer becomes less favorable compared to the traditional PRP-based transfer. Therefore, *BandSlim* employs an adaptive transfer strategy, dynamically switching between fine-grained and PRP-based transfers to balance traffic reduction with optimal response times for varying value sizes.

For evaluation, we implemented *BandSlim* on a state-of-the-art field-programmable gate array (FPGA)-based NVMe KV-SSD[5] using the Cosmos+ OpenSSD platform[8] and estimated CXL.mem-based value transfer's benefits using real CXL.mem measurements. We demonstrated that *BandSlim* achieves up to 97.9% traffic reduction compared to an NVMe KV-SSD without *BandSlim*.

The contributions of this article are as follows:

- We Identify traffic amplification in the host–SSD interconnect, specifically in KV-SSDs, and propose NVMe and CXL-based solutions to minimize waste.
- We propose the design of a CXL-based storage interface and demonstrate use cases where the interface shows clear advantages, particularly in KV-SSDs.
- We demonstrate the tradeoff between data traffic and response time in fine-grained value transfer and effectively resolve it by using an adaptive approach.

BACKGROUND AND MOTIVATION

Storage Stack of KV-SSDs

The storage stack of KV-SSDs consists of user-level key-value APIs, a key-value device driver and controller based on protocols like NVMe, and an in-storage KVS. The key-value application programming interface (API) offers point and range queries (e.g., PUT and SEEK). The size of the key and the value in these APIs is handled as arbitrary lengths, not in block units (*key-value interfaces*). In case of log-structured merge (LSM)-based KV-SSDs with a key-value separation,[3,4,5,6] a pair of key and value address is stored in the LSM tree, and a value is stored in the Value Log (vLog). The vLog is a linear logical NAND flash address space that can be divided into multiple logical NAND pages. Each value is appended to the vLog sequentially, filling logical NAND pages, which are mapped to physical NAND pages by the Flash Translation Layer (FTL). The entries of the LSM tree point to corresponding values inside the vLog.

NVMe-Based Key-Value Pair Transfer

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 PRP as the block interface part of the NVMe specification does.[7] 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. Later, the controller inserts the received key to the memory component of the LSM tree and MemTable and writes the value to the vLog (see Figure 1). The reverse operation, involving the transfer of values from the KV-SSD to the host, follows a similar process.

Peripheral Component Interconnect Express Traffic Amplification in NVMe KV-SSDs

As in typical KVSs, the key and the value size are variable and not necessarily aligned to a memory page. According to Meta, RocksDB in a production environment experiences a size of values that nearly do not reach 100 bytes on average,[9] 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, the current NVMe key-value interface causes significant inefficiencies because it adheres to the same procedure as the block-based NVMe protocol when transferring values to or from the device. *Specifically, its data transfer method, the PRP, restricts DMA transfers to occur in units of 4 KB, the size of a memory page*. As shown in Figure 1, 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 (4 K + 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 amplified data traffic significantly raises energy consumption of the system, possibly increasing the total cost of ownership.

Experimental Analysis of Traffic Amplification

We measured Peripheral Component Interconnect Express (PCIe) traffic from a host to a KV-SSD[5] by issuing 1 million key-value writes with variable-sized values by using the Intel PCM[10] to track PCIe traffic.

Figure 2(a) shows the total PCIe traffic during the experiments, with traffic increasing stepwise at 4-KB value-size boundaries. For example, total PCIe traffic for 1 and 4-KB values is approximately 4 GB, indicating constant transfer volumes up to 4 KB. This pattern repeats for value-size ranges like 5–8 KB. Average transfer response times display similar cascading patterns.

The issue becomes more severe with smaller values. The traffic amplification factor, defined as *the ratio of PCIe traffic to data size*, surges for values under 1 KB. As shown in Figure 2(b), transferring a value of 32 bytes generates approximately 4 KB of PCIe data traffic, roughly 130 times the requested data size.

Limitations of Existing Methods

The NVMe protocol currently offers another transfer method besides the PRP, called the *Scatter-Gather List* (*SGL*).[7] Unlike the PRP, the SGL supports variable-sized DMAs across scattered memory segments. However, it has been reported that the cost of enabling the SGL

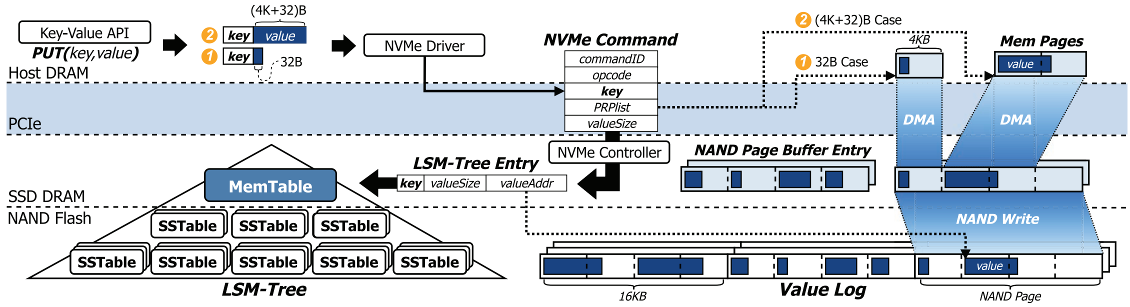


FIGURE 1. Data flow of two cases of key-value transfers with a sub-4-KB payload [32 bytes (B)] and a more than 4-KB payload [(4K + 32)B] regarding Peripheral Component Interconnect Express (PCIe) interconnect traffic amplification in LSM-based NVMe KV-SSDs.

outweighs the benefit for I/Os smaller than 32 KB,[11] indicating that using the SGL for small-value transfers is not advisable and realistic.

PROPOSED DESIGN: *BANDSLIM*

Fine-Grained Value Transfer Over NVMe

Due to space constraints, a detailed discussion of *BandSlim*'s first proposed technique, NVMe command piggybacking, is provided in the conference version of this article.[12] The summary is as follows: the method leverages the NVMe command, which is 64 bytes in size, to enable fine-grained transfer of small values. Considering most values in the real-world are less than 64 bytes,[9] it repurposes up to 35 bytes of unused fields in the NVMe key-value write command for value transfer. For values that exceed 35 bytes, the piggybacking method introduces a `transfer command`, which sends the remaining bytes in 56 bytes per command after the initial write command. This approach significantly reduces PCIe traffic under real-world KVS workloads.

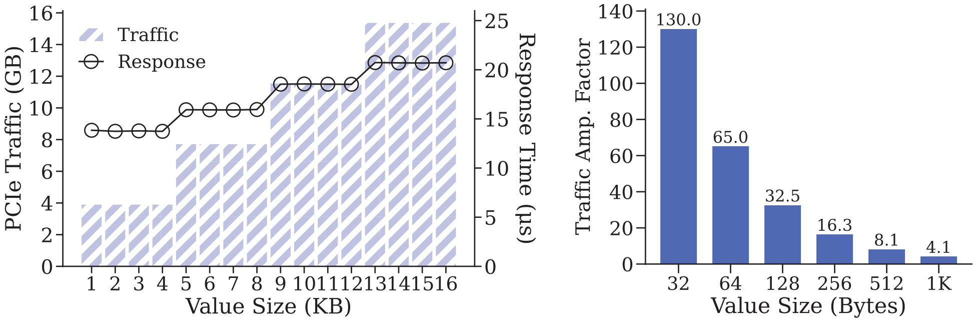
Fine-Grained Value Transfer Using CXL

Although the piggybacking method resolves the traffic amplification issue, it is still constrained by several key limitations. First, the overhead associated with creating, submitting, processing, and releasing NVMe commands is significant, particularly when dealing with a large value that demands issuing multiple commands, which makes this method effective only for transfers of tiny values, typically in the range of tens of bytes. Second, it cannot completely eliminate traffic bloating as it still depends on NVMe commands, which requires extra traffic for signaling doorbells and completions.

To overcome the limitations of repetitive NVMe command overheads in the piggybacking method, we also propose a design that fundamentally supports fine-grained transfer at the interconnect level using CXL. For this, *BandSlim* defines the NAND page buffer as host-managed device memory (HDM), making it directly accessible to the host through CXL.mem operations. During a CXL device enumeration, the driver queries the Base Address Register (BAR) and the size of the HDM to map the BAR and HDM within the host's system memory. The host CPU's system bus includes CXL root ports (RPs), establishing connections with KV-SSDs as endpoints. After that, the driver can transmit values via CXL.mem write operations.

Figure 3 illustrates the value transmission process in this method. The driver manages a CXL.mem write pointer (CMWP) to track the current position in the NAND page buffer. When a user initiates a key-value write, the NVMe command includes only the key, unlike the previous piggybacking method. ❶ The driver initiates a memory copy request to the CMWP. ❷ The host CPU then sends the value to the NAND page buffer at the CMWP location via CXL.mem. ❸ The CXL RP receives this request, converting it into *flits*, that is, CXL transaction units (*BandSlim* operates based on the default 68-byte *flit* mode). ❹ The CXL controller decompresses the *flits*, adjusts target addresses by subtracting the HDM base address, and forwards the request to the dynamic random-access memory (DRAM) controller, ❺ which sends the payload to the DRAM location indicated by the CMWP. ❻ The controller notifies the driver to update the CMWP via NVMe command completion.

As the FTL operates independently, the host cannot determine when each buffer entry is flushed. Thus, *BandSlim* always calls *clflush()* for CXL.mem writes to ensure that the payload is sent to device memory.



(a) (b)

FIGURE 2. Measurements of total PCIe traffic with average response time and PCIe traffic amplification (amp.) factor for varying value sizes, with NAND I/O disabled. (a) Total PCIe traffic and average response time. (b) Traffic amplification time.

Adaptive Value Transfer Method

As the value size increases, the performance of both fine-grained value transfer methods starts to degrade. The first method, in particular, suffers due to the accumulation of overheads in generating NVMe commands and synchronously handling them within the device, resulting in much longer transfer times compared to a conventional PRP-based transfer. To tackle these issues, *BandSlim* utilizes a threshold-based reactive method that selects the most suitable transfer method from both the fine-grained and PRP-based transfers based on the size of the value. The threshold is identified by exploratory runs using benchmarks depicted in the evaluation. 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.

The threshold is denoted as $\tau_\alpha = \alpha \cdot \tau$, where τ represents the value size at which the fine-grained transfer becomes less efficient than PRP-based transfers. *BandSlim* employs the following strategy:

$$\text{Transfer method} = \begin{cases} \text{Fine-grained transfer,} & \text{if value size} < \tau_\alpha, \\ \text{PRP-based transfer,} & \text{if value size} \geq \tau_\alpha. \end{cases}$$

The scaling factor α allows users to adjust the threshold: increasing α raises the threshold, while setting α to one retains the default threshold. For users who prioritize response time, α can be set to one. For those who focus on traffic reduction, α can be increased to delay the transition point where PRP-based transfers become more efficient. This approach ensures efficient handling of value sizes ranging from subpage to large.

In-Device Value Packing Mechanism

Our conference paper proposed the *selective packing with backfilling policy* to address the NAND page write amplification problem in NVMe KV-SSDs.[12] This policy aims to reduce the overhead of copying large values during the packing process within the device under adaptive value transfer by selectively packing only small values transmitted via fine-grained transfer, while large values are placed to page-aligned addresses via PRP-based DMA without packing. To minimize internal fragmentation, the policy allows small values to fill gaps between large values at page-aligned addresses and also avoid them with a DMA log table (DLT).

Figure 4 illustrates the process of packing under CXL.mem-based transfer. ❶ When a user requests a large value write that exceeds the threshold τ_α, the value is transferred via PRP DMA over NVMe (CXL.io). ❷ The controller completes the DMA and creates a DLT entry based on the DMA destination address and value size. ❸ This entry is sent to the host in the NVMe completion queue entry (CQE), ❹ where the *BandSlim* driver caches it. For a small value below the threshold τ_α, the driver checks if adding the value size to the CMWP exceeds the specified address in the cached DLT head entry. ❺ and ❻ If it does not, the value is transferred to the CMWP address over CXL.mem with *clflush()* enabled, while only the key is sent in an NVMe command. ❼ and ❽ If the next small-value size exceeds the address in the cached DLT head entry, the CMWP updates to the DLT entry's address plus the value size, and the value is transferred to the updated CMWP address via CXL.mem. ❾ For another large value, the transfer occurs via PRP DMA, with the destination address aligned to the nearest page boundary, tracked by the controller. ❿ The new address and value size are added to the DLT and cached on the host, as shown in Figure 4.

As only the NAND page buffer is exposed as HDM, while CMWP and DLT prevent conflicts between CXL.mem and PRP DMA, *BandSlim* preserves data consistency and metadata integrity. Plus, because *BandSlim* does not modify the core components of KV-SSDs,

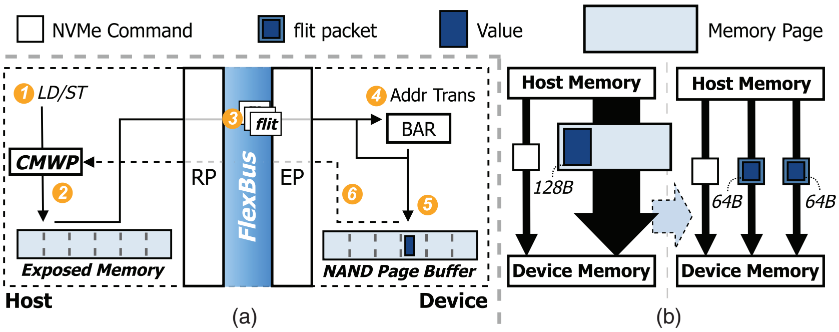


FIGURE 3. (a) CXL.mem memory write mechanism and (b) fine-grained value transfer via CXL.mem. CMWP: CXL.mem write pointer. EP: endpoint; load/store (LD/ST).

including key-value indexing and FTL, the existing read and query performance remains unaffected.

EVALUATION

Evaluation Setup

We used the state-of-the-art NVMe KV-SSD[5] on the Cosmos+ OpenSSD platform[8] as the baseline to verify our solutions. The platform consists of an ARM-based Xilinx Zynq-7000 system on chip (SoC), 1 TB of NAND (four channels, eight ways), and a PCIe Gen2 × 8 interconnect, paired with a host node featuring 64 cores of Intel Xeon Gold 6226 R CPU, 384 GB of DDR4 memory, and Ubuntu 22.04 operating system. The SoC operates the *BandSlim* key-value, PCIe interface, DRAM, and NAND flash controllers, while the host node runs the *BandSlim* key-value driver and benchmarks.

As publicly accessible SSDs and FPGA boards that support CXL remain extremely scarce (including ours) we validated our designs by implementing the piggybacking mechanism on the OpenSSD-based setup, while the CXL.mem-based transfer was evaluated through simulations using measured CXL.mem response times. The traditional PRP-based transfer and the NVMe piggybacking method underwent thorough system-level validation on the OpenSSD setup, providing reliable performance measurements. We then scaled the measured payload transfer times of baseline and piggybacking to PCIe Gen5 and incorporated the NVMe overheads into the CXL.mem response times to simulate the CXL.mem-based transfers. This approach allowed us to directly compare the performance of PRP-based transfers, NVMe piggybacking, and CXL.mem-based transfers, demonstrating the potential benefits of CXL.mem optimizations.

For performance evaluations, we used db_bench, a benchmarking tool used in RocksDB.[1] We enabled db_bench to send NVMe key-value commands to the OpenSSD platform through the NVMe pass through.

We conducted various workload patterns to verify our proposed design, which we describe as follows:

- *WorkloadA* [*W(A)*]: A db_bench's *fillseq* pattern where keys are sequential and value sizes are fixed. It serves as a baseline for evaluating performance under uniform and predictable write patterns.
- *WorkloadB* [*W(B)*]: This writes 1 million random key-value pairs with value sizes of 8 bytes or 2 KB at a 9:1 ratio. This tests the transfer method's efficiency in optimizing for frequent small-value writes.
- *WorkloadC* [*W(C)*]: Similar to *W(B)* but reverses the value-size ratio to 1:9, emphasizing scenarios with large-value dominance. The goal is to reveal the tradeoffs in handling small versus large values.
- *WorkloadD* [*W(D)*]: This workload writes values of sizes (eight; 16; 32; 64; 128; 256; and 512 bytes; 1 KB; and 2 KB) in random order, totaling 1 million, with each size having an equal ratio. This evaluates the transfer method's adaptability to diverse data sizes.
- *WorkloadM* [*W(M)*]: This is a db_bench's *mixgraph All_random*9 workload that reflects real-world characteristics with a maximum value size of 1 KB and almost 70% of values being less than 35 bytes. We have modified *mixgraph* to issue only 1 million PUTs.

In all the experiments, we used unique keys of 4 bytes generated by a hash function with a random seed.

We conducted evaluations for the following designs:

- *Baseline:* The state-of-the-art NVMe KV-SSD.5 It employs the PRP-based page-unit value transfer.

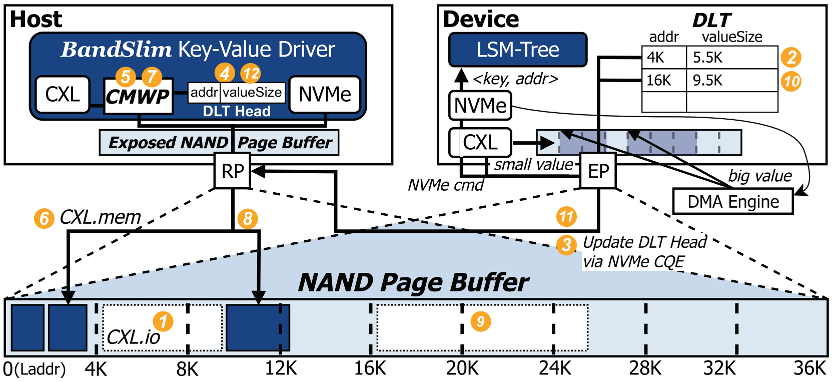


FIGURE 4. The process of *selective packing with backfilling* over the NAND page buffer with CMWP and DLT. CQE: completion queue entry.

- *Piggyback:* Transfers values only via piggybacking.
- *CXL.mem-Based Fine-Grained Value Transfer (CXL-FGT):* Transfers values only via CXL.mem.
- *Adaptive:* Transfers values via the adaptive method.

Effects of Fine-Grained Value Transfer

Piggybacking-Based Value Transfer

The evaluation of PCIe traffic and response times for *Piggyback* on a real platform is detailed in our conference paper[12] as space constraints prevent an in-depth discussion here. Briefly, *Piggyback* achieves up to 97.9% traffic reduction for values of 4–32 bytes but becomes less efficient as the value size increases due to the overheads of trailing commands, approaching *Baseline* at approximately 2 KB and surpassing it for larger sizes. Response times are halved for values up to 32 bytes but degrade for larger sizes due to the trailing commands' overheads.

CXL.mem-Based Value Transfer

To evaluate *CXL-FGT*, we first measured write response times for various value sizes using a 256-GB CXL memory device connected via PCIe Gen5, consistently invoking *clflush()*. These results, obtained on a dedicated server with a CXL-supported AMD EPYC 9754 configured as a CPU-less nonuniform memory access node, served as the foundation for a performance simulation model of *CXL-FGT*.

Next, we analyzed the response times[12] for *Baseline* and *Piggyback*, measured from our OpenSSD (PCIe Gen2) setup. We extracted the payload transfer time for various payload sizes from the total response times by separating the time spent on generating, submitting, processing, and completing NVMe commands. We then scaled the extracted payload transfer times by a factor of 6.4, reflecting the higher giga transfer rate of Gen5 (32 GT/s) compared to Gen2 (5 GT/s). Although actual performance gains may be smaller due to system overheads, this adjustment assumes an ideal scenario focused purely on bandwidth improvements. We added the NVMe overhead back to the scaled payload transfer times of *Baseline* and *Piggyback* to estimate their response times on PCIe Gen5. Finally, we created a performance simulation model for *CXL-FGT* by combining the measured CXL.mem write response times with the same NVMe overheads used for *Baseline* (*CXL-FGT* always require a single NVMe command). This approach enables a direct comparison of *Baseline*, *Piggyback*, and *CXL-FGT* designs. Figure 5(a) illustrates the time breakdown and adjustment used to estimate the response times of *Baseline* and *Piggyback* on PCIe Gen5 for values of 32 bytes.

Using the *CXL-FGT* simulation model and the estimated response times of *Baseline* and *Piggyback*, we evaluated the performance of each transfer mode

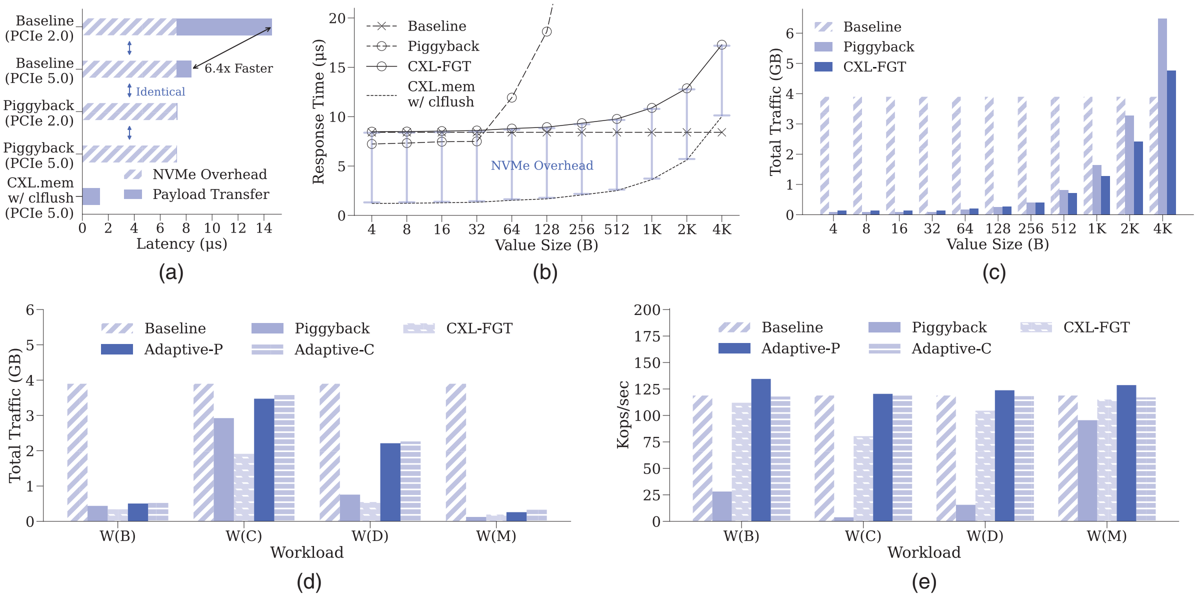


FIGURE 5. (a) Breakdown of estimated response times for values of 32 bytes on PCIe Gen5 using *Baseline* and *Piggyback*. (b) Estimated response times [*W*(*A*)] and (c) total PCIe traffic[*W*(*A*)] for *Piggyback* and *CXL-FGT* with 1 million key-value pairs [*W*(*A*)] of varying value sizes transferred from host to device memory. (d) Estimated PCIe traffic and (e) average throughput for various workloads [*W*(*B*), *W*(*C*), *W*(*D*), and *W*(*M*)] on PCIe Gen5. B: bytes.

across different value sizes using *WorkloadA*. The results, presented in Figure 5(b), reveal key insights. First, *CXL-FGT* performs worse than *Piggyback* for value sizes of 4–32 bytes due to the mandatory NVMe command overhead separated from CXL transfers. This limitation could be mitigated by integrating piggybacking with *CXL-FGT*. Second, *CXL-FGT* does not outperform *Baseline* on PCIe Gen5, again, highlighting a limitation due to the mandatory NVMe command overhead. However, as expected, *CXL-FGT* significantly outperforms *Piggyback* starting at the 64-bytes case, demonstrating how CXL.mem mitigates the performance degradation of *Piggyback* for larger data sizes. For instance, in the 25-bytes case, *CXL-FGT* achieves nearly 4× faster response times compared to *Piggyback*, with an even greater performance gap as value sizes increase.

Figure 5(c) shows that *CXL-FGT* generates less traffic compared to *Piggyback*. Using CXL *flit* units of 68 bytes, we estimated *CXL-FGT*'s traffic, while *Baseline* and *Piggyback* reflect measured values.[12] CXL enables finer granularity because *Piggyback* incurs additional traffic with each `transfer command` submission, such as doorbell rings and tail pointer reads. CXL avoids this overhead, allowing more efficient data movement. However, note that piggybacking's primary strength lies in resolving KV-SSD bottlenecks without requiring new interconnect technologies.

Effects of Adaptive Value Transfer

The evaluation results of the adaptive value transfer using *Workloads B*, *C*, *D*, and *M* on a real platform is detailed in our conference paper.[12] Briefly, *Adaptive* consistently outperforms *Piggyback* and *Baseline* by transitioning from piggybacking to PRP at 128 bytes [identified as the threshold based on experiments with [*W*(*A*)], achieving the best performance in all workloads. It reduces traffic by 93.3% for *W*(*M*) while improving throughput by 12% over *Piggyback*, and in *W*(*C*), it increases throughput nearly 13-fold despite generating 18% more traffic than *Piggyback*.

To assess the impact of CXL.mem-based transfer under various workloads, we simulated *Workloads B*, *C*, *D*, and *M* using the results from Figure 5(b) and (c). Figure 5(d) shows the traffic results, where *CXL-FGT* achieved lower overall traffic than *Piggyback* in all workloads except *W*(*M*) despite the additional traffic overhead compared to *Piggyback* for transferring values of 4–32 bytes. This reduction stems from avoiding frequent NVMe command overheads by using CXL.mem, with the benefit most evident in *W*(*C*) due to its larger values. For the *Adaptive* method, we evaluated scenarios employing piggybacking (*Adaptive-P*) and CXL.mem (*Adaptive-C*). The threshold for switching the transfer mode in *Adaptive-P* was set to 64 bytes, based on the results in Figure 5(b). The same threshold was applied to *Adaptive-C* to ensure a fair comparison between piggybacking and CXL.mem. As expected, both methods exhibited higher traffic than fine-grained transfers. Moreover, as *Piggyback* inherently resulted in lower traffic compared to *CXL-FGT* for value sizes between 4 and 32 bytes, *Adaptive-C* exhibited slightly higher traffic than *Adaptive-P*.

Figure 5(e) presents the average throughput derived by the simulated response times. Across all workloads, *CXL-FGT* consistently outperformed *Piggyback*, further demonstrating the effectiveness of CXL.mem-based transfers. Meanwhile, *Adaptive-P* achieved the best performance in all workloads, while *Adaptive-C* generally showed performance comparable to *Baseline*. Again, this result is attributed to the inherent limitations of *CXL-FGT*, which retains the mandatory NVMe command overhead. In particular, workloads with a high proportion of value sizes between 4 and 32 bytes [e.g., *W*(*B*) and *W*(*M*)] highlighted the limitations of *CXL-FGT*, which adheres to NVMe routines, resulting in *Adaptive-C* performing worse than *Adaptive-P*.

In summary, *CXL-FGT* achieves finer-grained traffic and significantly better performance than *Piggyback* as value sizes increase. However, as the method maintains NVMe transactions, the performance advantage of fine-grained transfer over *Baseline* diminishes with the increased bandwidth of PCIe Gen5. Furthermore, *CXL-FGT* does not fully utilize the cache coherency of CXL as it actively invokes *clflush()* for each write. We plan to address these limitations by developing new storage protocols that leverage the full potential of CXL.

CONCLUSION

BandSlim tackled the PCIe traffic amplification challenges in KV-SSDs, which arose from conflicts between non-page-aligned key-value pairs and NVMe protocols designed for memory page units. *BandSlim* leveraged NVMe command piggybacking and CXL.mem to transfer values in a fine-grained and efficient manner. Extensive evaluations showcased that *BandSlim* significantly reduced traffic and optimized data transfer efficiency.

ACKNOWLEDGMENTS

This work was funded in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT) (Grant Nos. RS-2024-00453929 and RS-2025-00564249), in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government (MSIT) under Grant RS-2021-II210528, and in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (Grant No. 2018-0-00503, Researches on next generation memory-centric computing system architecture). Youngjae Kim is the corresponding author.

REFERENCES

1. "A persistent key-value store," *RocksDB*, 2014. Accessed: Nov. 30, 2024. [Online]. Available: http://rocksdb.org
2. A. Tomlin. "Why KV SSD will replace ZNS." SNIA, 2022. Accessed: Nov. 30, 2024. [Online]. Available: https://www.snia.org/educational-library/why-kv-ssd-will-replace-zns-2022
3. C. G. Lee et al., "iLSM-SSD: An intelligent LSM-tree based key-value SSD for data analytics," in *Proc. IEEE 27th Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS)*, 2019, pp. 384–395, doi: 10.1109/MASCOTS.2019.00048.
4. I. Park et al., "KV-CSD: A hardware-accelerated key-value store for data-intensive applications," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2023, pp. 132–144, doi: 10.1109/CLUSTER52292.2023.00019.
5. S. Lee et al., "Iterator interface extended LSM-tree-based KVSSD for range queries," in *Proc. 16th ACM Int. Syst. Storage Conf. (SYSTOR)*, 2023, pp. 60–70.
6. D. Min et al., "A multi-tenant key-value SSD with secondary index for search query processing and analysis," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 4, pp. 1–27, Jul. 2023, doi: 10.1145/3590153.
7. "Key value command set specification," *NVM Express Inc.*, 2024. [Online]. Available: https://nvmexpress.org/specifications/
8. J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ OpenSSD: Rapid prototype for flash storage systems," *ACM Trans. Storage*, vol. 16, no. 3, pp. 1–35, Jul. 2020, doi: 10.1145/3385073.
9. H. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 209–224.
10. "Intel: PCM," GitHub, 2010. Accessed: Dec. 1, 2024. [Online]. Available: https://github.com/intel/pcm
11. "NVMe: Add scatter-gather list (SGL) support in NVMe driver," OpenWRT, 2017. [Online]. Available: https://merlin.infradead.org/pipermail/linux-nvme/2017-July/011956.html pcm
12. J. Park et al., "BandSlim: A novel bandwidth and space-efficient KV-SSD with an escape-from-block approach," in *Proc. 53rd Int. Conf. Parallel Process. (ICPP)*, 2024, pp. 1187–1196.

JUNHYEOK PARK is pursuing his M.S. degree at Sogang University, Seoul, 04107, South Korea. His research interests include next-generation NAND flash drives and file and storage systems. Park received his B.S. degree in computer science and engineering Sogang University. Contact him at junttang@sogang.ac.kr.

CHANG-GYU LEE is pursuing his Ph.D. degree in computer science and engineering at Sogang University, Seoul, 04107, South Korea. His research interests include operating systems and file and storage systems. Lee received his B.S. degree in software and computer engineering from Ajou University. Contact him at changgyu@u.sogang.ac.kr.

SOON HWANG is pursuing his Ph.D. degree at Sogang University, Seoul, 04107, South Korea. His research interests include file and storage systems for high-performance computing and artificial intelligence. Hwang received his B.S. and M.S. degrees in computer science and engineering from Sogang University, in 2021 and 2023, respectively. Contact him at soonhw@sogang.ac.kr.

SEUNG-JUN CHA is a senior researcher at the Electronics and Telecommunications Research Institute, Daejeon, South Korea. His research interests include enhancing performance in system software, such as operating systems and database management systems, and he is currently focused on advancing memory interconnect technologies. Cha received his B.E., M.E., and Ph.D. degrees in computer engineering from Chungnam National University, in 2006, 2008, and 2013, respectively. Contact him at seungjunn@etri.re.kr.

WOOSUK CHUNG is a vice president at SK hynix, where he leads the Software Solution Group in the Memory Systems Research organization. Chung received his B.S. degree in electronic engineering from Hanyang University in 1997 and his M.S. degree in engineering from Hanyang University in 1999. His research interests include AI and big data analytics optimized memory and storage systems. Contact him at woosuk.chung@sk.com.

YOUNGJAE KIM is a professor with the Department of Computer Science and Engineering, Sogang University, Seoul, 04107, South Korea. His research interests include operating systems, file and storage systems, and parallel and distributed systems. Kim received his Ph.D. degree in computer science and engineering from The Pennsylvania State University. Contact him at youkim@sogang.ac.kr.