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

Junhyeok Park, Chang-Gyu Lee, Soon Hwang, Jungki Noh, Soonyeal Yang,
Woosuk Chung, Junghee Lee, and Youngjae Kim

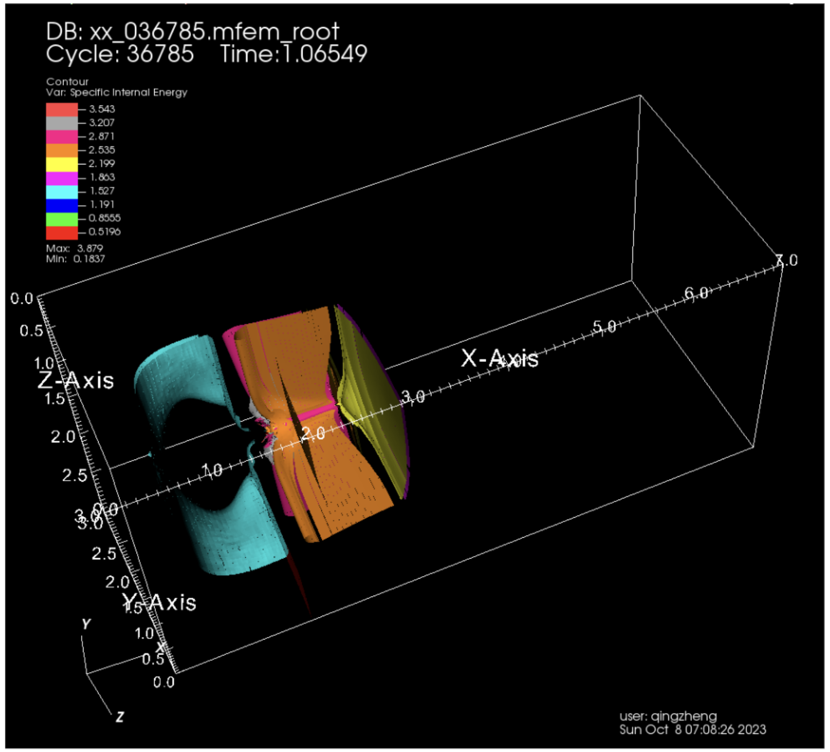
ICPP 2024

Presenter: Junhyeok Park

Background

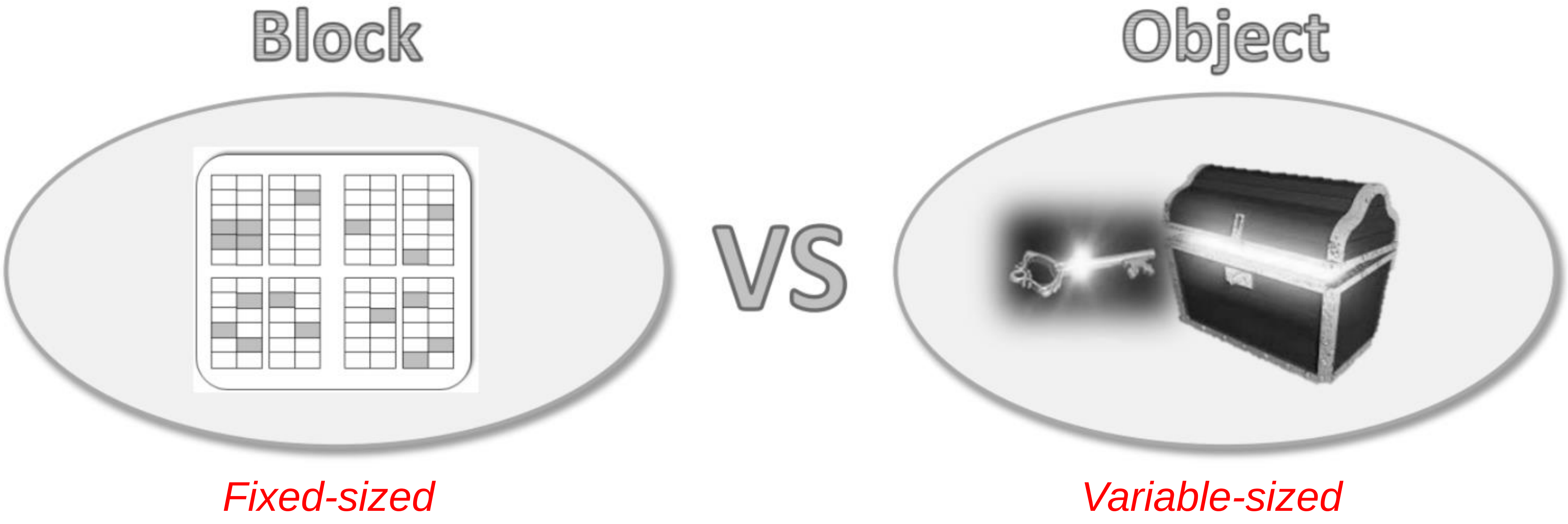
Big Data Era

- A rapid adoption of Artificial Intelligence (AI), High-Performance Com- -puting (HPC), Data Analytics, and Cloud Service in these days.
  - They handle **“Big Data”**.



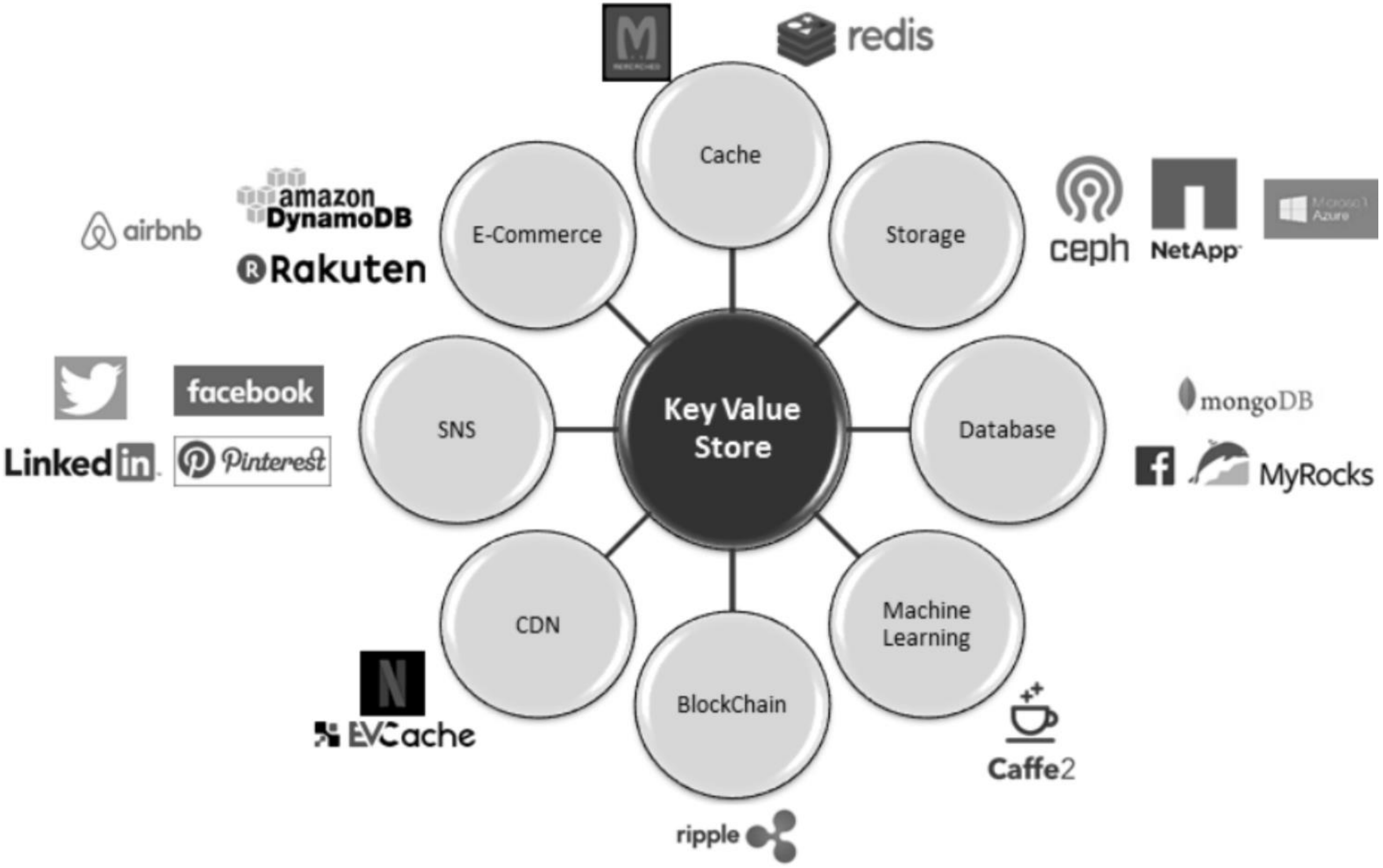
What does Data look like?

- These Big Data applications do not merely handle Blocks; they manage variable-sized **Key-Value Pairs** or **Objects**.



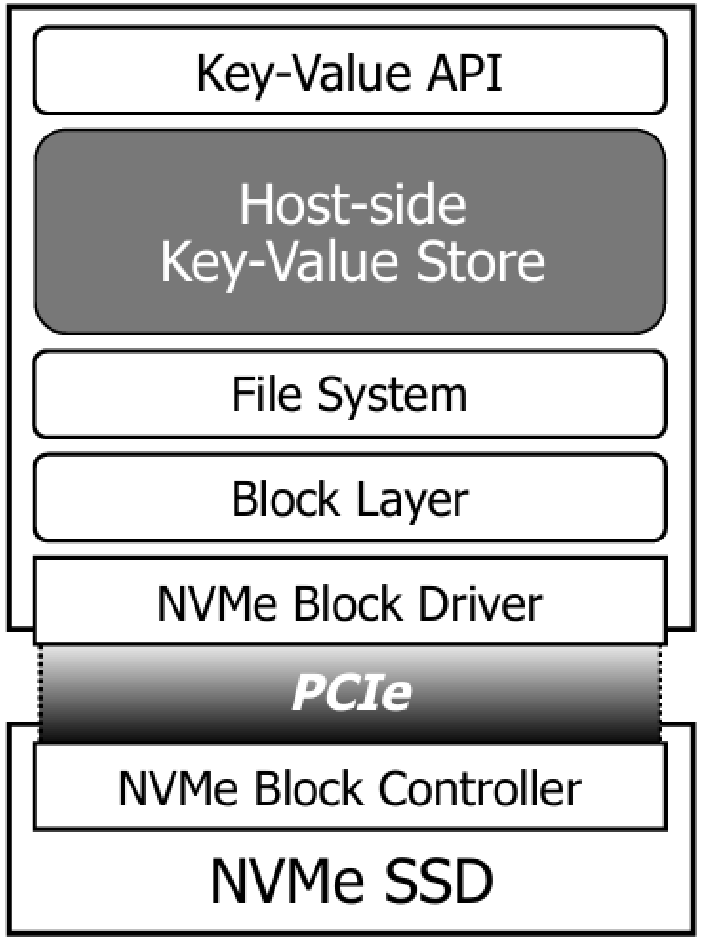
Key-Value Store

- Therefore, these Big Data applications typically operate by employing Key-Value Stores (e.g., RocksDB, Cassandra).



Software Stack Issue

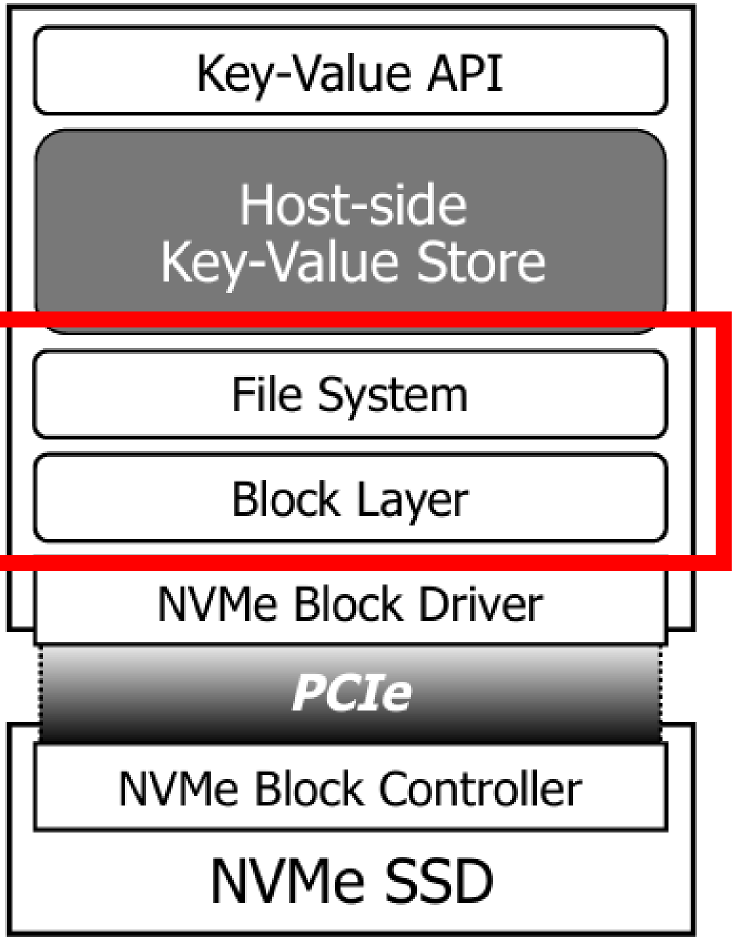
- Key-Value Stores run on top of file system & block layer, device driver and device controller.



Software Stack Issue

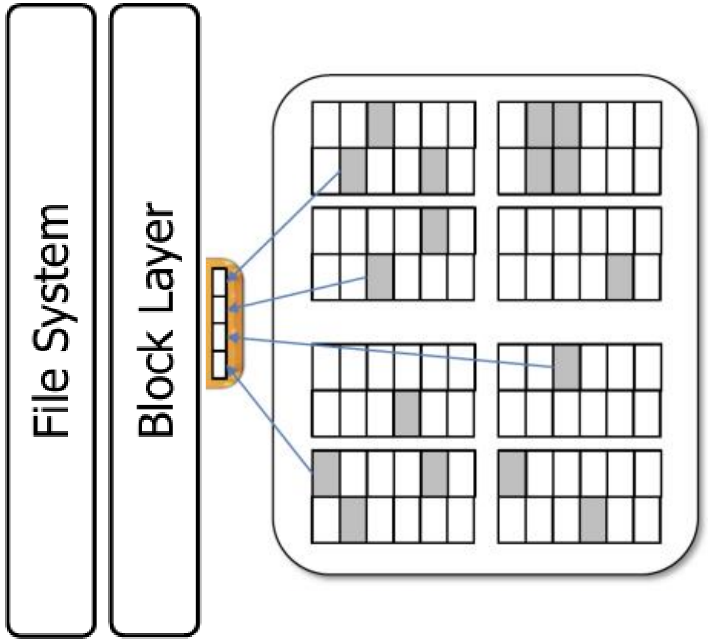
- Key-Value Stores run on top of file system & block layer, device driver and device controller.

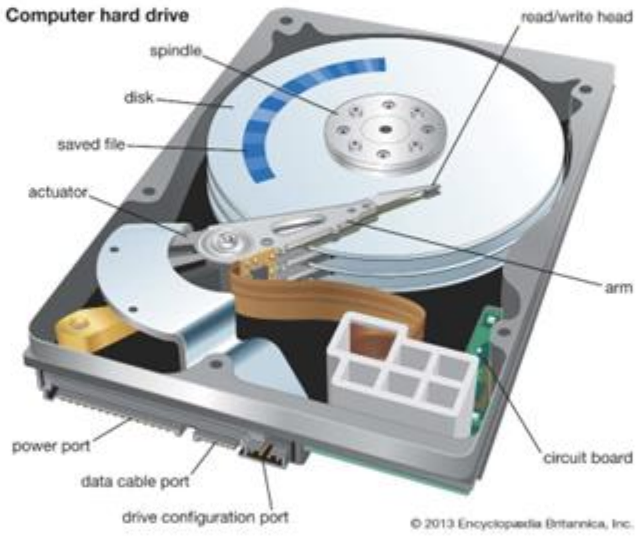
Do we really need these layers?



Software Stack Issue

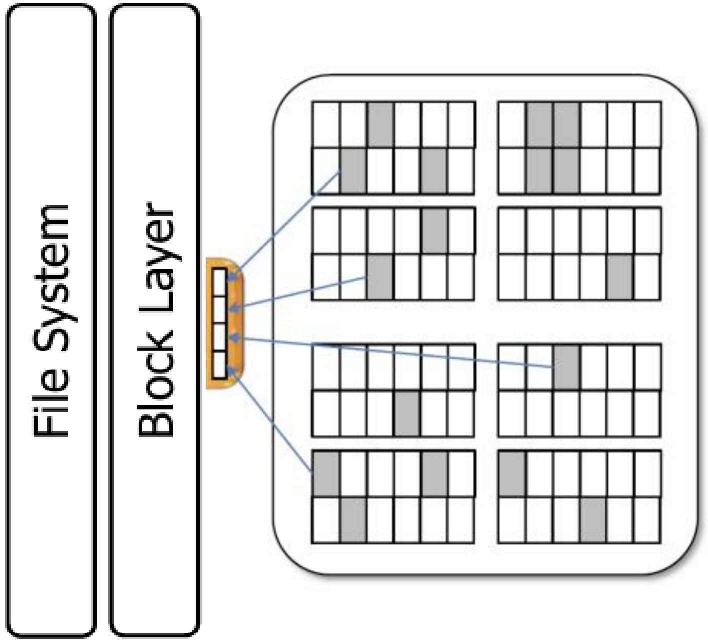
- These layers are in place to follow the **block interface**, which originated from the hard disk drives.





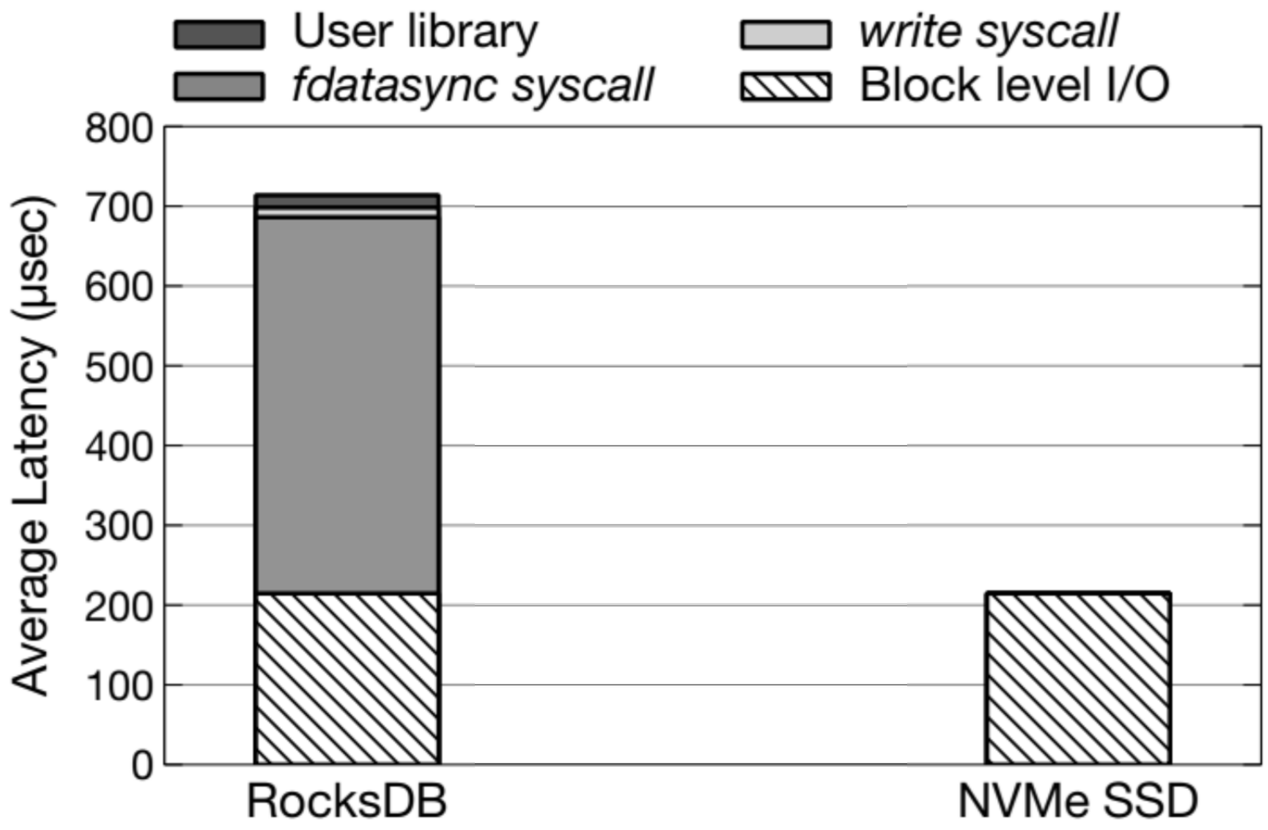
Software Stack Issue

- These layers are in place to follow the **block interface**, which originated from the hard disk drives.



Software Stack Issue

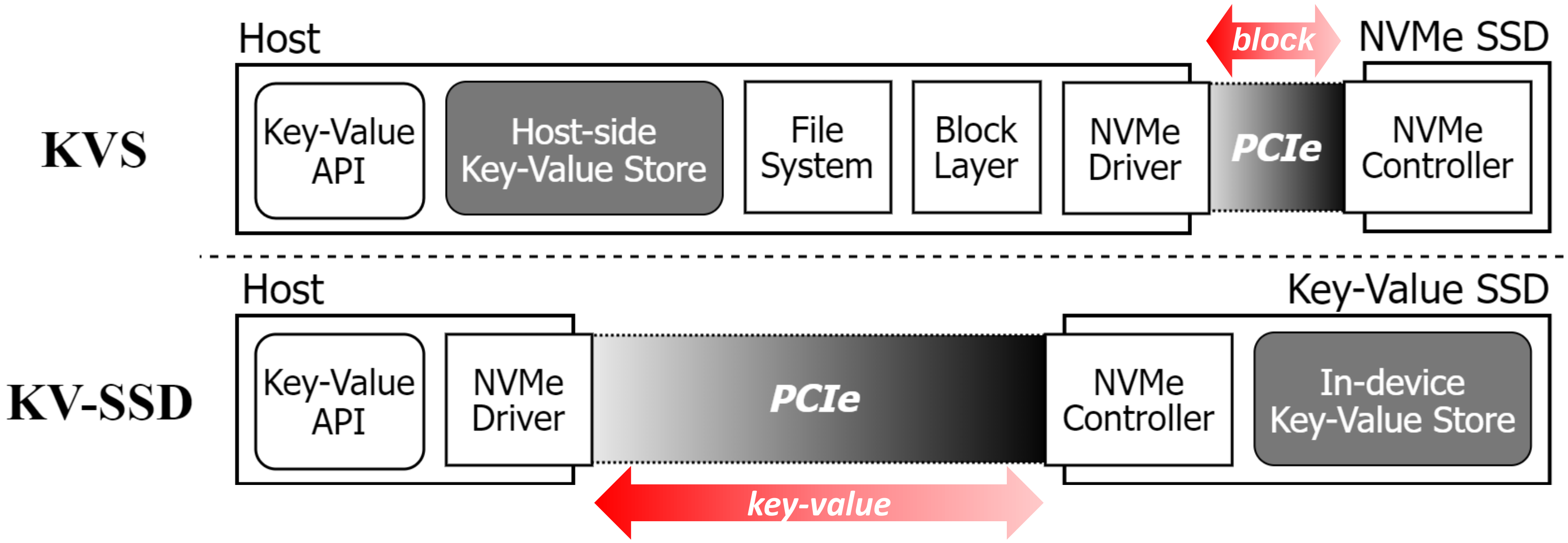
- The problem is that these layers account for a **significant portion** of the total response time in Key-Value Stores [1].



[1] Lee, C. G., Kang, H., Park, D., Park, S., Kim, Y., Noh, J., Chung, W., & Park, K. (2019). iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics. In Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).

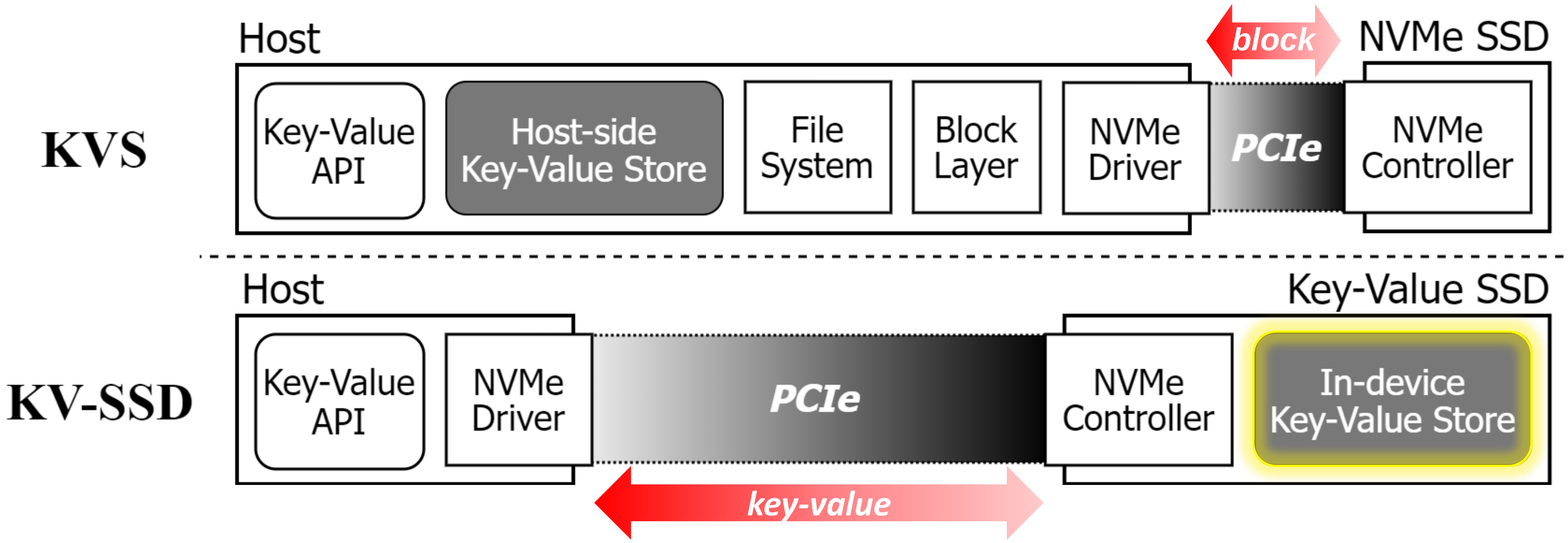
Key-Value Solid State Drive (KV-SSD)

- What about streamlining these layers from the storage stack?
  - By making a key-value pair as the unit of data communication interface
- KV-SSDs have renovated the storage interface by changing the unit of I/O transactions from the traditional block to key-value.



Key-Value Solid State Drive (KV-SSD)

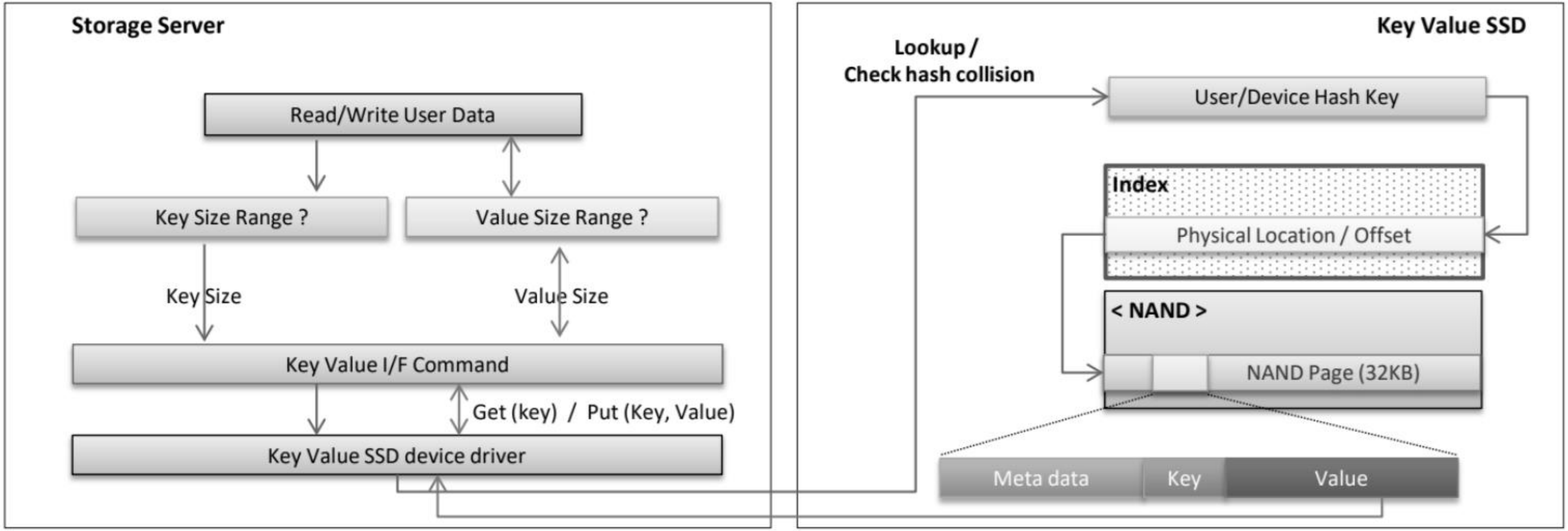
- What about streamlining these layers from the storage stack?
  - By making a key-value pair as the unit of data communication interface
- KV-SSDs have renovated the storage interface by changing the unit of I/O transactions from the traditional block to key-value.



➔ **lower latency & higher throughput**

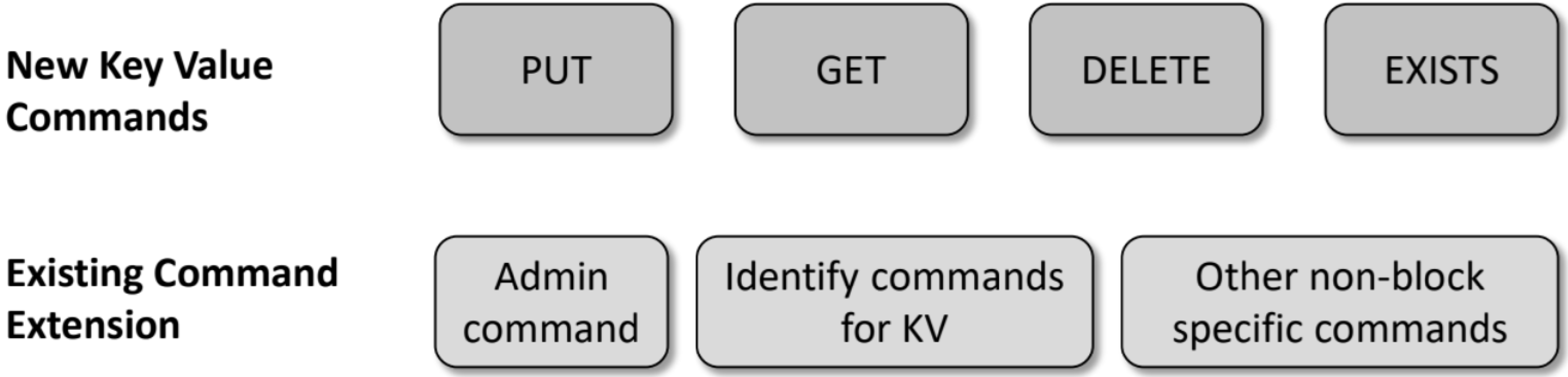
Key-Value Solid State Drive (KV-SSD)

- KV-SSD supports key-value store operations like PUT and GET.
- KV-SSD maintains Key-to-Page mapping info by deploying index structures like Hash Table or LSM-tree.



NVMe Key-Value Command Set

- The NVMe protocol has introduced a key-value command set.

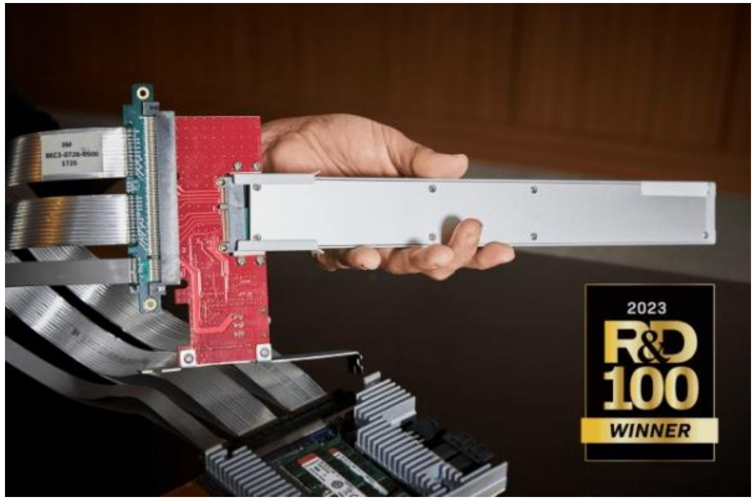


NVMe Key-Value Command Set

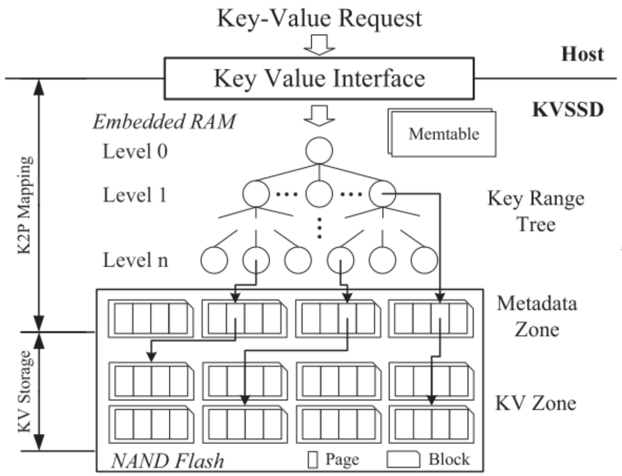
- The NVMe protocol has introduced a key-value command set.

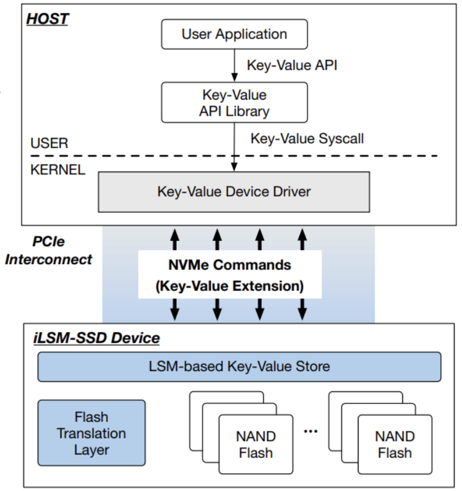
- Most of commercially and academically released KV-SSDs have utilized the NVMe key-value command set to offer key-value interface.

***SK hynix KV-CSD* [2]**



Academia

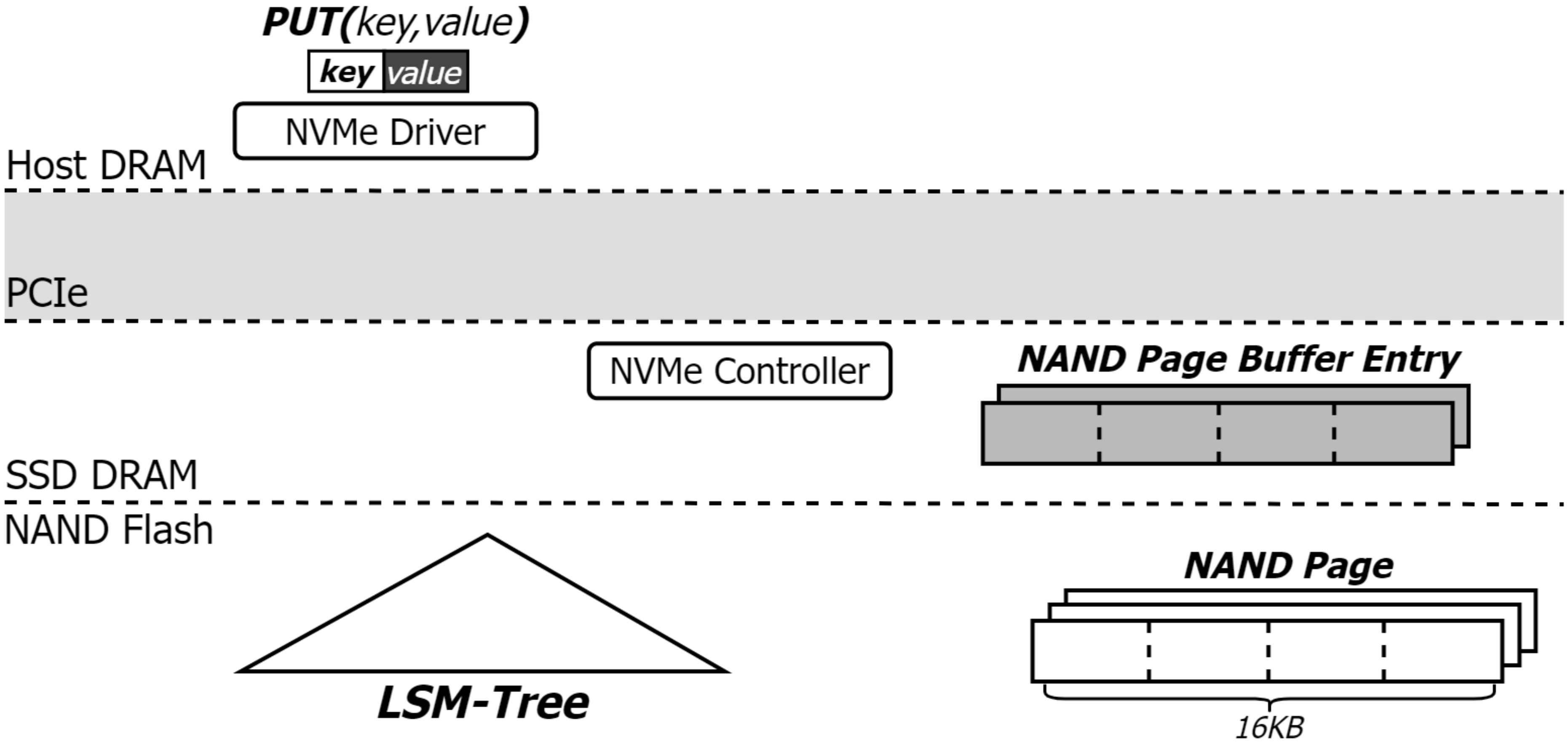




[2] Park, I., Zheng, Q., Manno, D., Yang, S., Lee, J., Bonnie, D., Settlemyer, B., Kim, Y., Chung, W., & Grider, G. (2023). KV-CSD: A Hardware-Accelerated Key-Value Store for Data-Intensive Applications. In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER), 132–144.

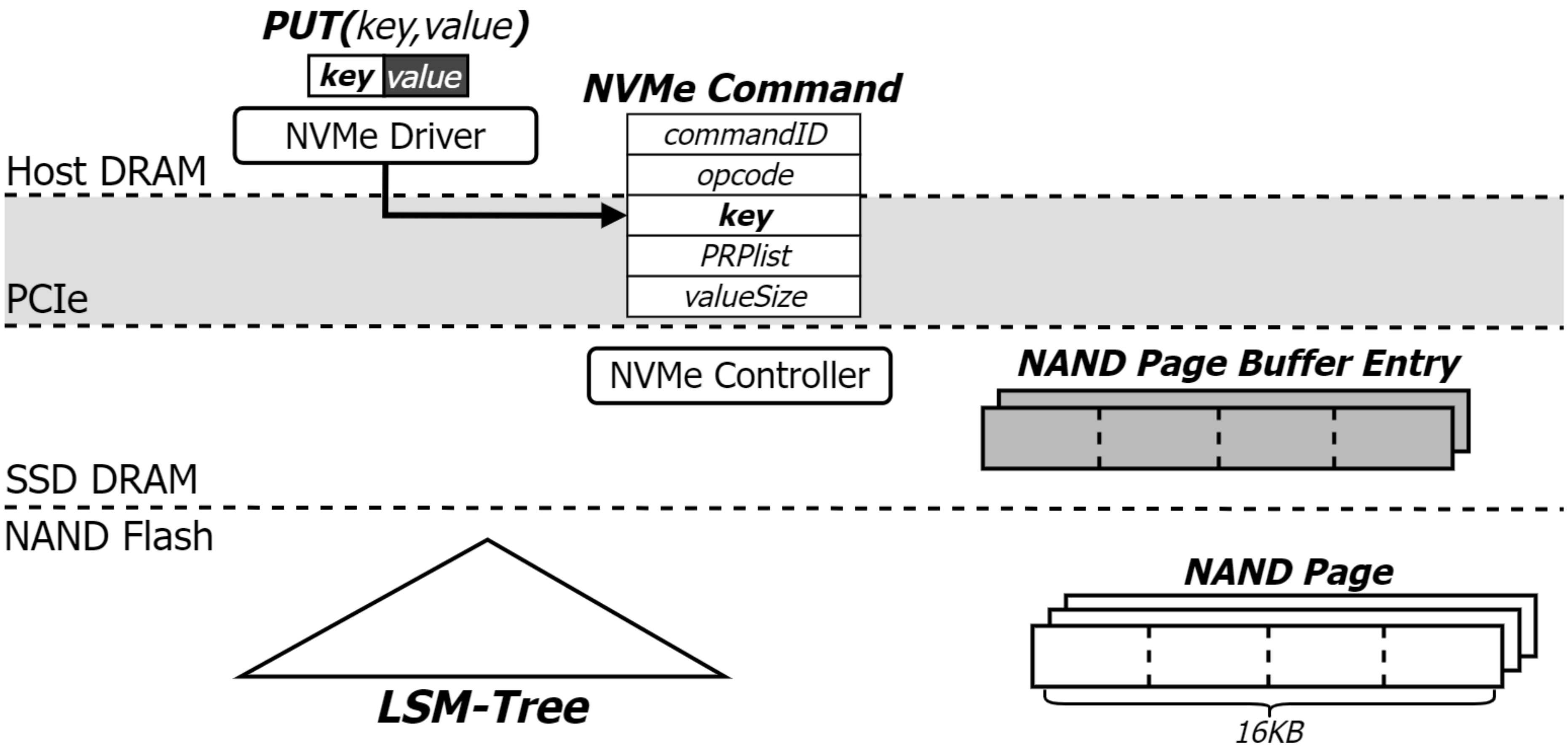
NVMe Key-Value Write Mechanism

- In a case of NVMe KV-SSD based on the LSM-tree with a key-value separation (e.g., iLSM-SSD, KV-CSD), when writing key-value pairs, ...



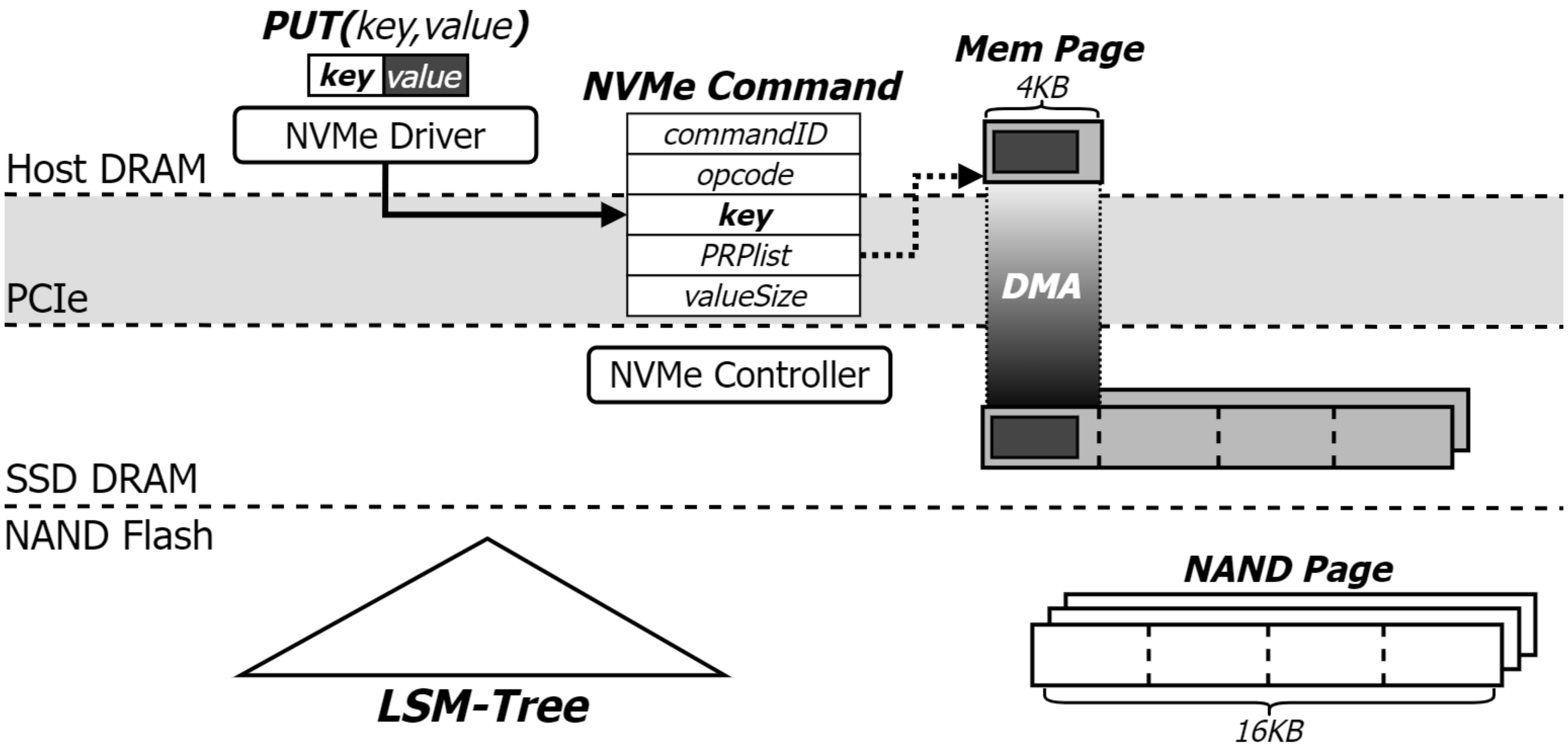
NVMe Key-Value Write Mechanism

- The NVMe driver stores a key and metadata in the NVMe command, and then submits the command to the SQ and rings the doorbell.



NVMe Key-Value Write Mechanism

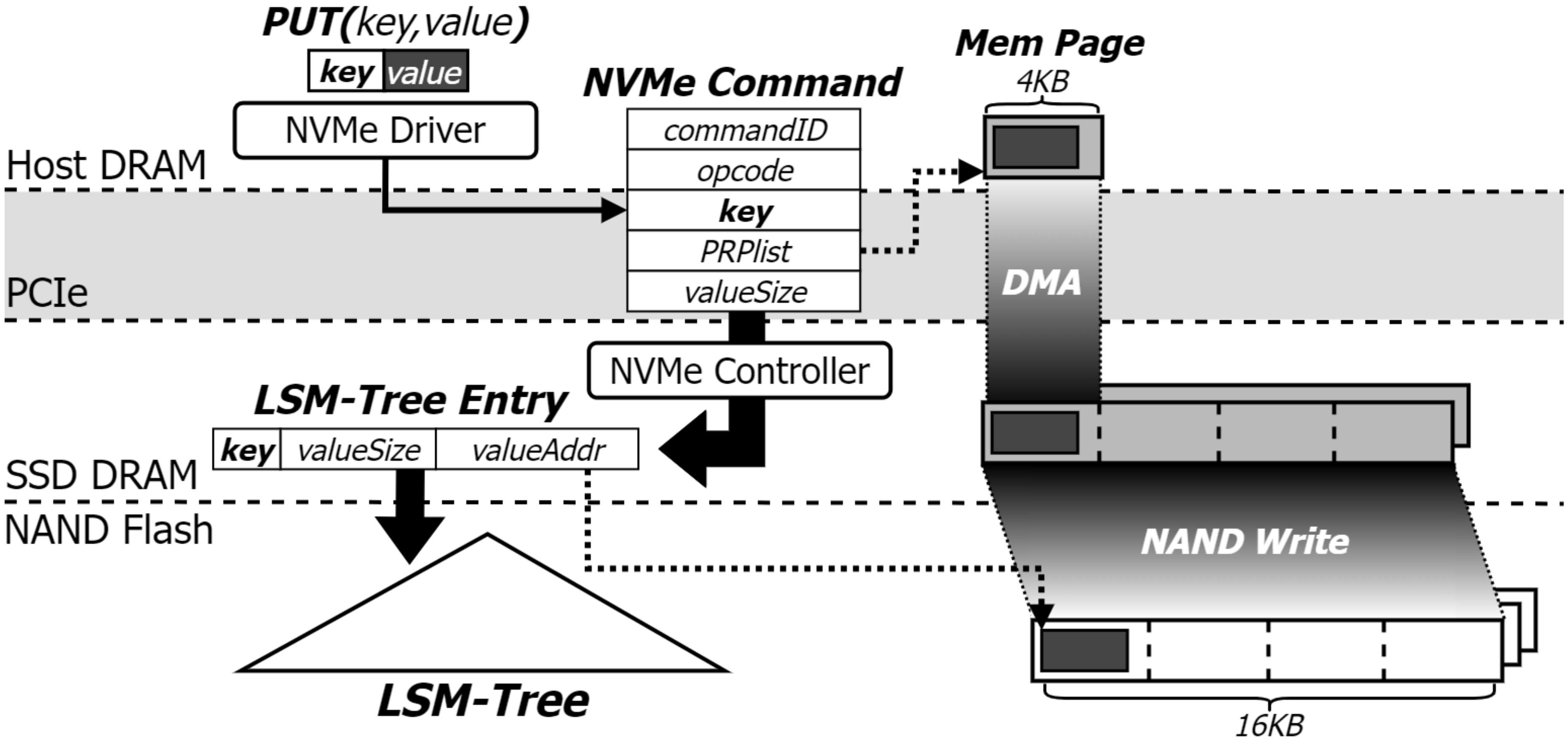
- The NVMe controller issues a DMA transaction to copy the payload (value) to the NAND page buffer within the device's DRAM.



NVMe Key-Value Write Mechanism

- The controller constructs the LSM-tree entry containing the key, value size, and value pointer, and programs the NAND page buffer entry.

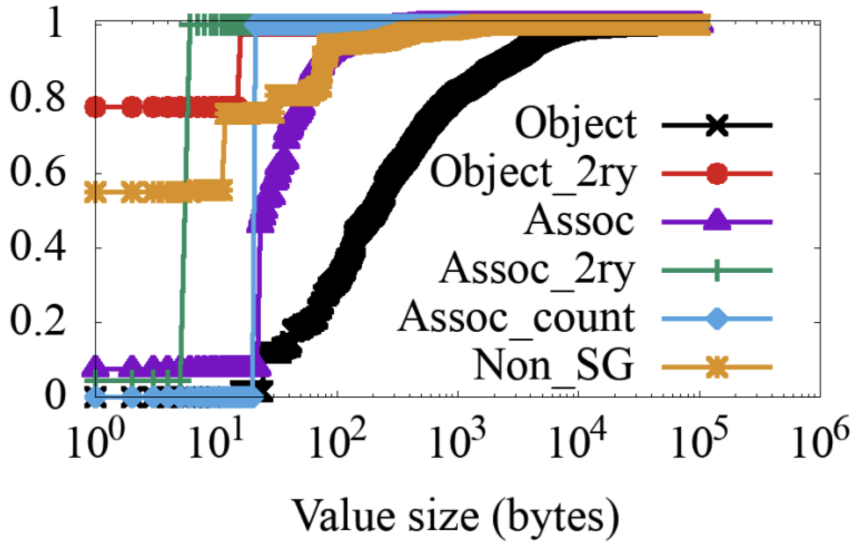
(to show the flow clearly, it programs the NAND page buffer entry even though it's not full)



Motivation

Problem Definition

- As in typical KVSs, the key and value size are variable and small, and not necessarily aligned to a block or a memory page.
  - According to Meta, their popular LSM KVS, 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 KiB memory page size.



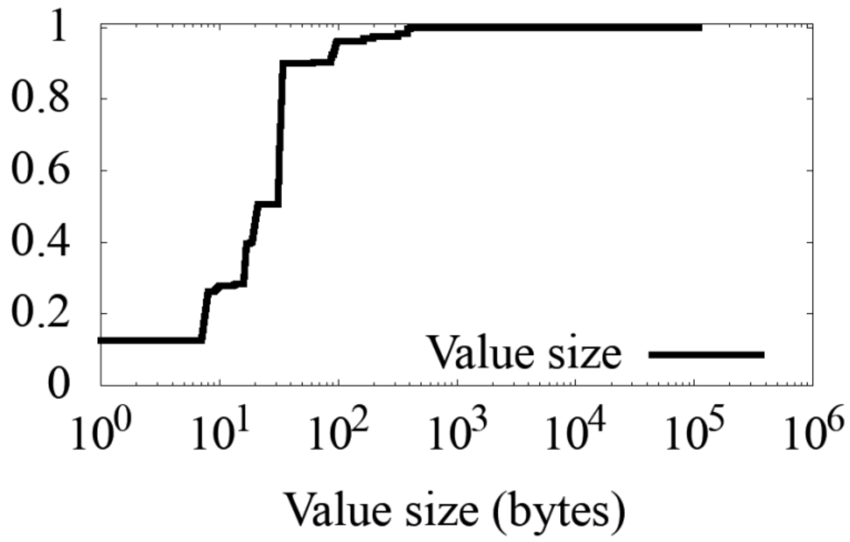
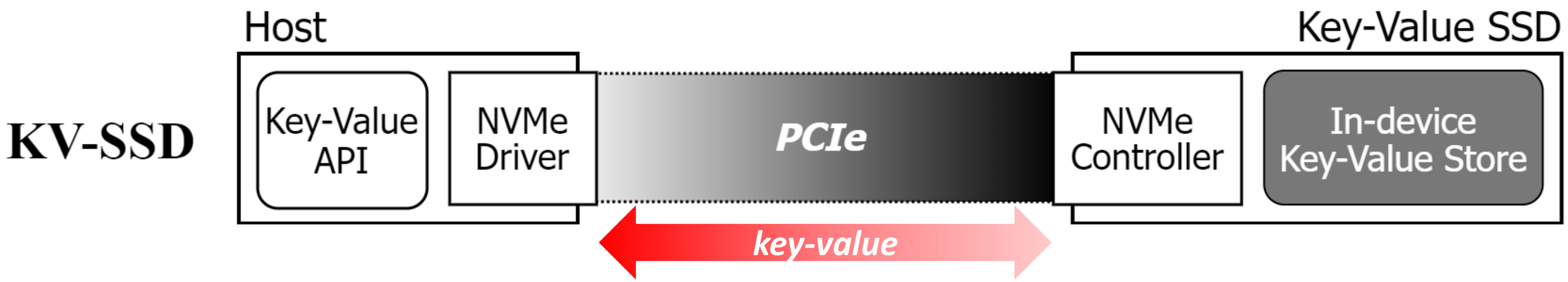


Figure – Value Size CDF for RocksDB as a MySQL storage layer (left) and RocksDB as a distributed KVS (right)

[3] Cao, Z., Dong, S., Vemuri, S., & Du, D. H. C. (2020). *Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In Proceedings of the 18th USENIX Conference on File and Storage Technologies* (FAST '20) (pp. 1-14). Santa Clara, CA, USA.

Problem Definition

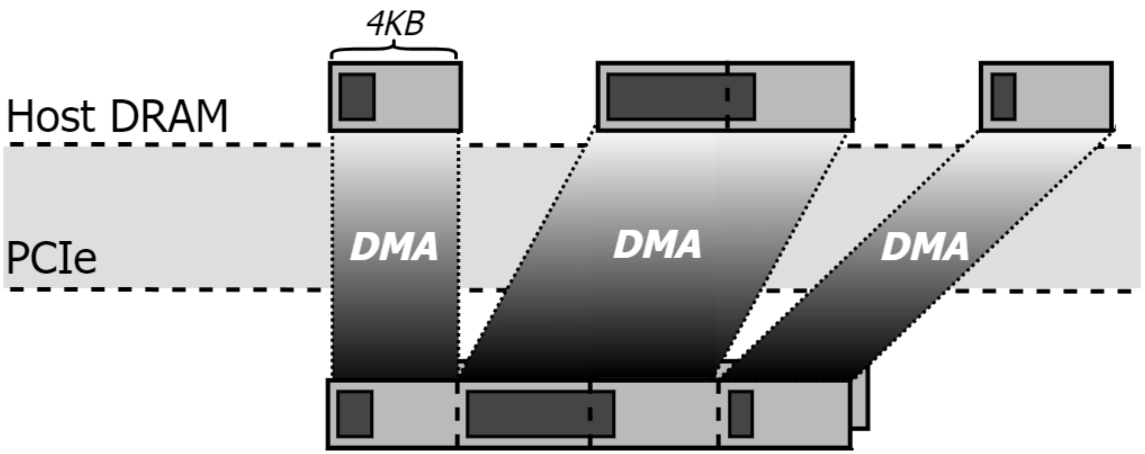
- The problem occurs with the fact that the NVMe key-value interface still cannot extricate itself from the deeply entrenched block-interface-assumed storage mechanisms and frameworks.



➔ **is it really a `*key-value*` interface?**

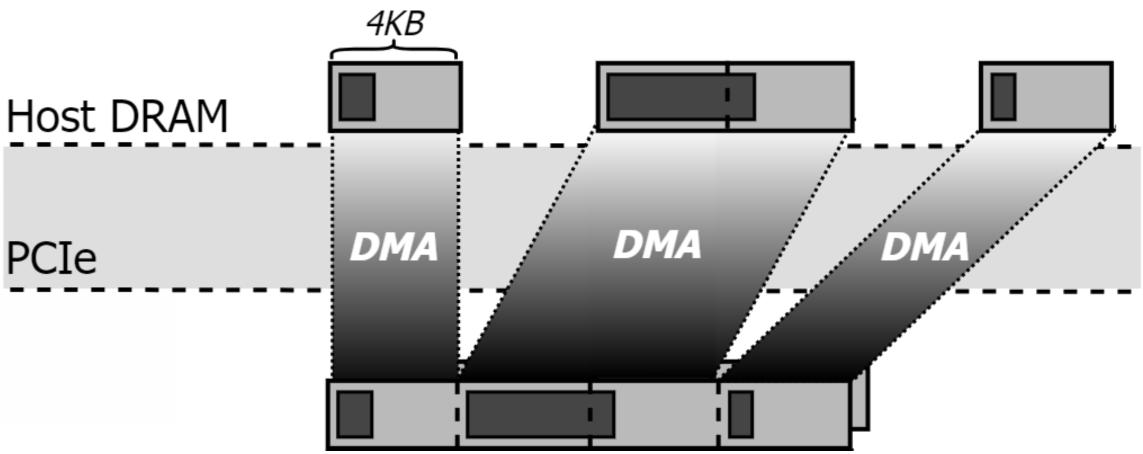
Problem #1. PCIe Traffic Amplification

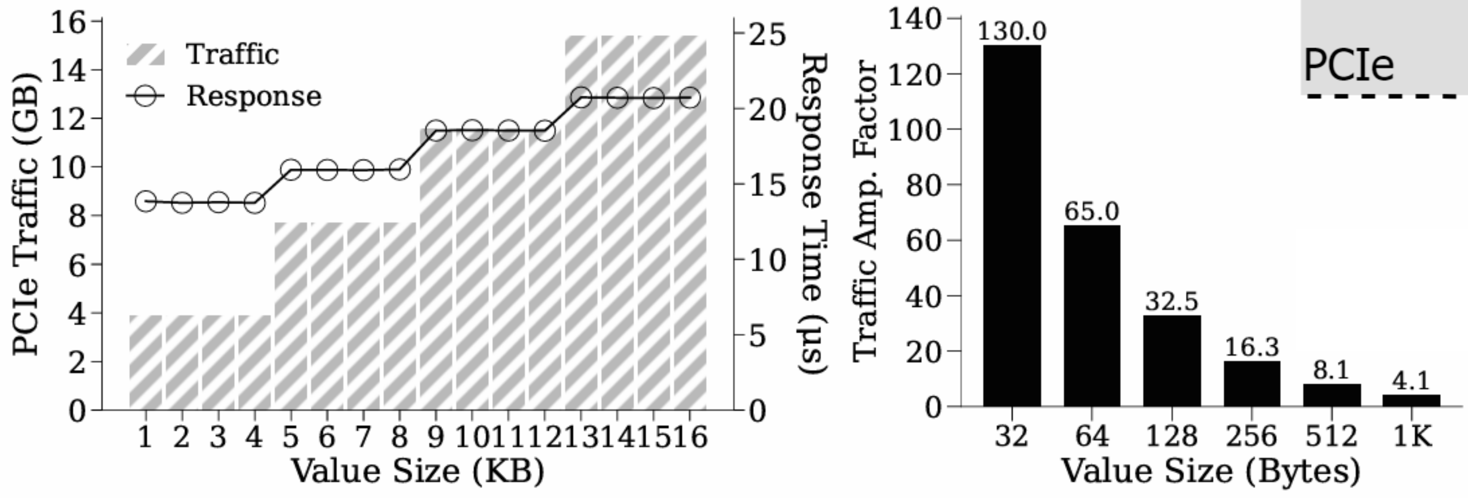
- The NVMe's payload transfer method, PRP, restricts DMA transfers to occur in units of 4 KiB, a size of memory page.
  - This leads to the bloated PCIe traffic during value transfers, especially for variable-sized, small values.



Problem #1. PCIe Traffic Amplification

- The NVMe's payload transfer method, PRP, restricts DMA transfers to occur in units of 4 KiB, a size of memory page.
  - This leads to the bloated PCIe traffic during value transfers, especially for variable-sized, small values.





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

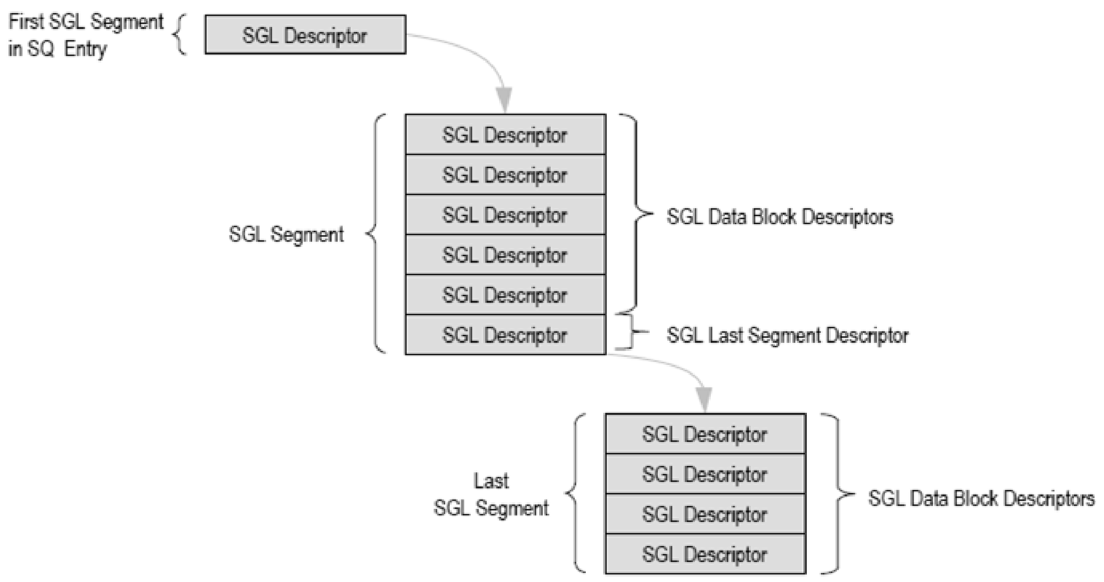
※ Traffic Amplification = (value size) / (PCIe traffic)

Setup	IterKVSSD (Systor '23) on Cosmos+ OpenSSD platform - feature: SOTA LSM-based KV-SSD - PCIe Gen2 x8 lane - 1GB of DRAM, 1TB of NAND (Toshiba), Xilinx zynq-7000
Workload	fillsequential of RocksDB's db_bench - number of PUTs: 1 million unique KV pairs - key size: 4 B

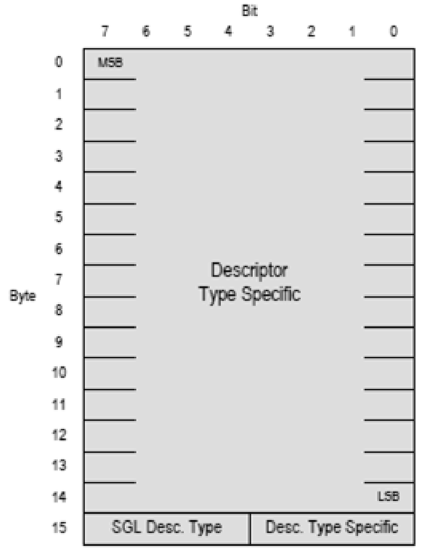
Problem #1. PCIe Traffic Amplification

- NVMe’s another payload transfer mechanism, Scatter-Gather List (SGL), can support multiple variable-sized DMAs across scattered memory segments.

SGL List



SGL Descriptor



Problem #1. PCIe Traffic Amplification

- However, it has been reported that the cost of enabling the SGL outweighs the benefit for I/O smaller than 32 KiB [4].
  - The Linux kernel thus establishes a minimum threshold for data transferred via SGL at 32 KiB [5], indicating that using SGL for small value transfers is not advisable.

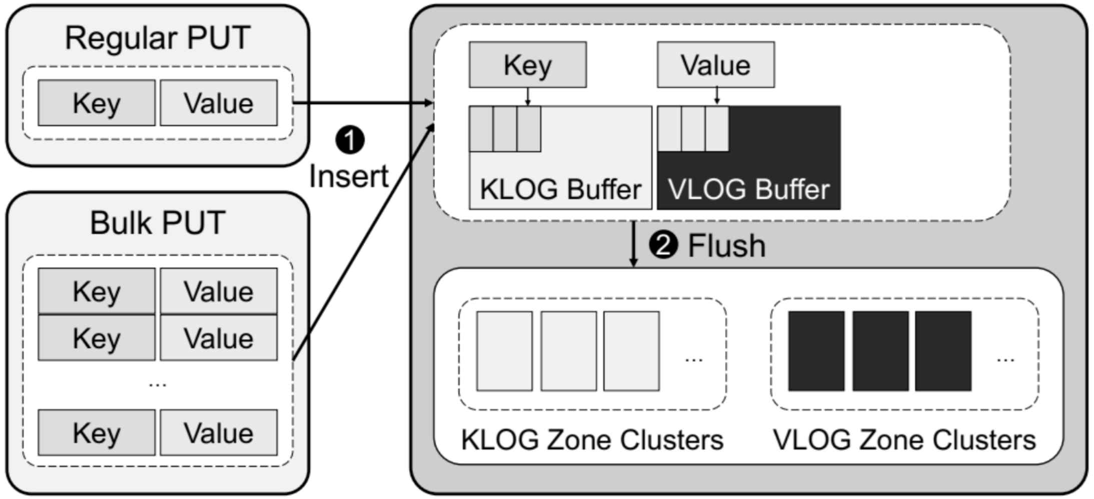
```
static unsigned int sgl_threshold = SZ_32K;
module_param(sgl_threshold, uint, 0644);
MODULE_PARM_DESC(sgl_threshold,
                "Use SGLs when average request segment size is larger or equal to "
                "this size. Use 0 to disable SGLs.");

#define NVME_PCI_MIN_QUEUE_SIZE 2
#define NVME_PCI_MAX_QUEUE_SIZE 4095
static int io_queue_depth_set(const char *val, const struct kernel_param *kp);
static const struct kernel_param_ops io_queue_depth_ops = {
        .set = io_queue_depth_set,
        .get = param_get_uint,
};
```

[4] 2017. nvme : add Scatter-Gather List (SGL) support in NVMe driver. https://lore.kernel.org/all/04aaed5c-1a8a-f601-6c9c-88bf1cf66e8a@mellanox.com/T/
[5] The Linux Kernel source code. sgl_threshold. https://github.com/torvalds/linux/blob/master/drivers/nvme/host/pci.c

Problem #1. PCIe Traffic Amplification

- KV-CSD and Dotori [6] have tackled this issue 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.

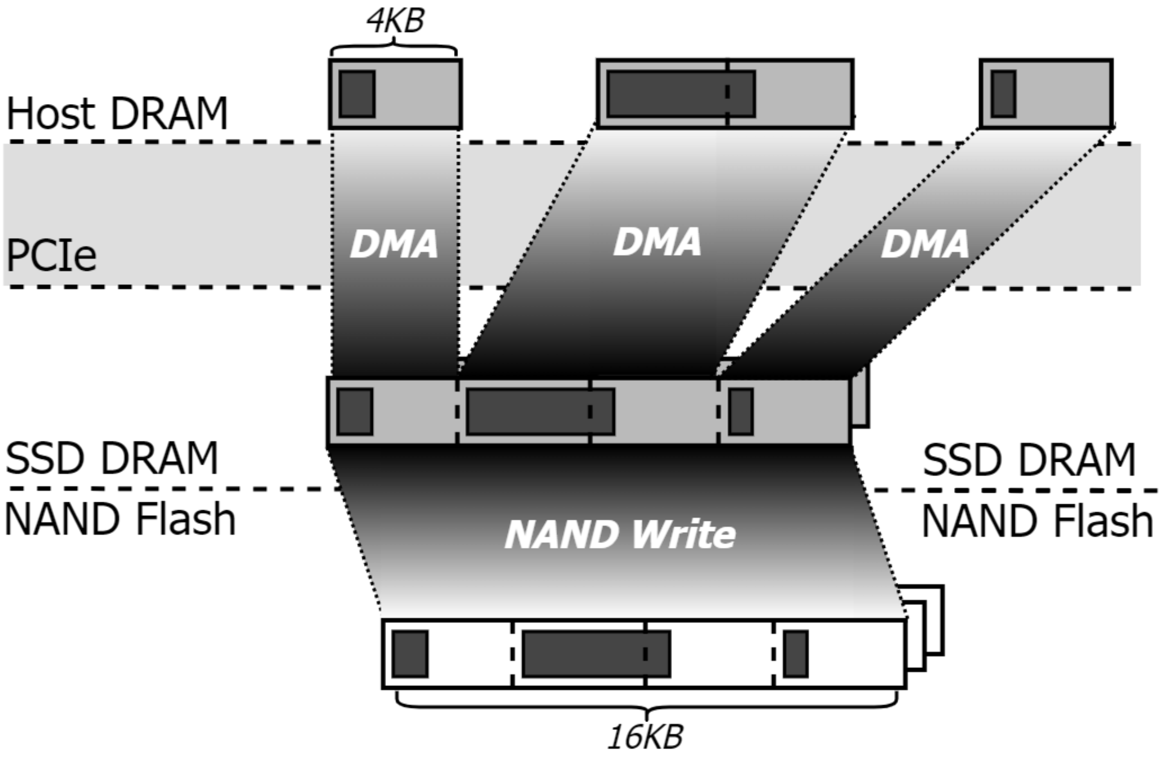


not suitable for mission-critical scenarios

[6] Duffy, C., Shim, J., Kim, S.-H., & Kim, J.-S. (2023). *Dotori: A Key-Value SSD Based KV Store. Proceedings of the VLDB Endowment,* 16(6), 1560–1572.

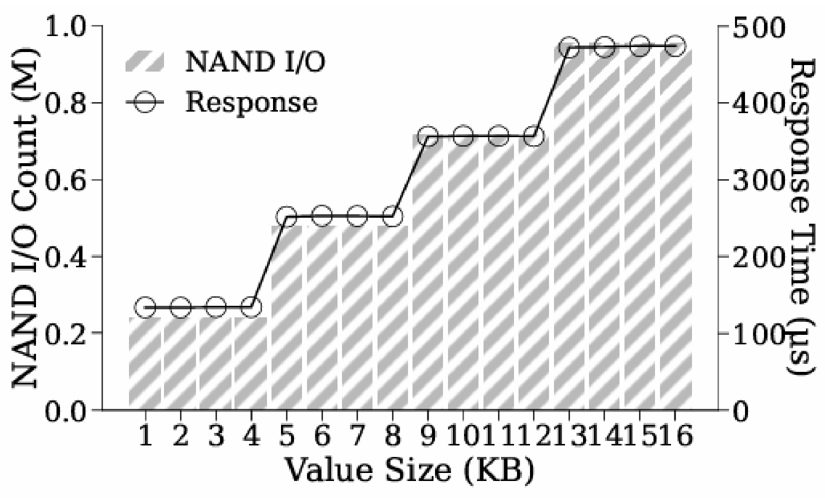
Problem #2. NAND Write I/O Amplification

- The packing of received payloads (values) into NAND pages within NVMe SSDs also occurs in units of 4 KiB.
  - This in-device page-unit packing clearly clashes with KV-SSDs, leading to severe NAND write amplification.

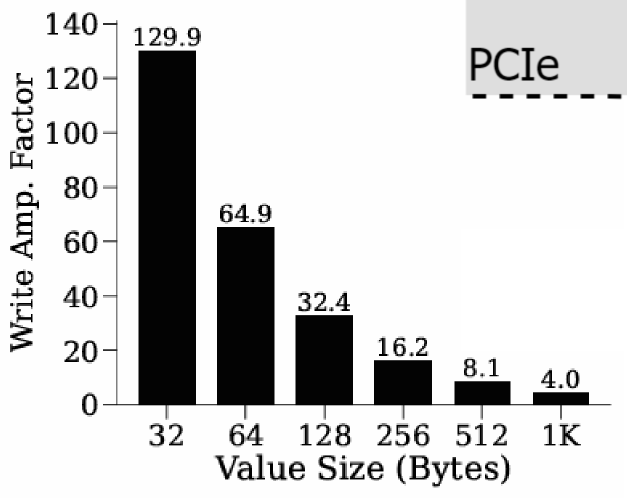


Problem #2. NAND Write I/O Amplification

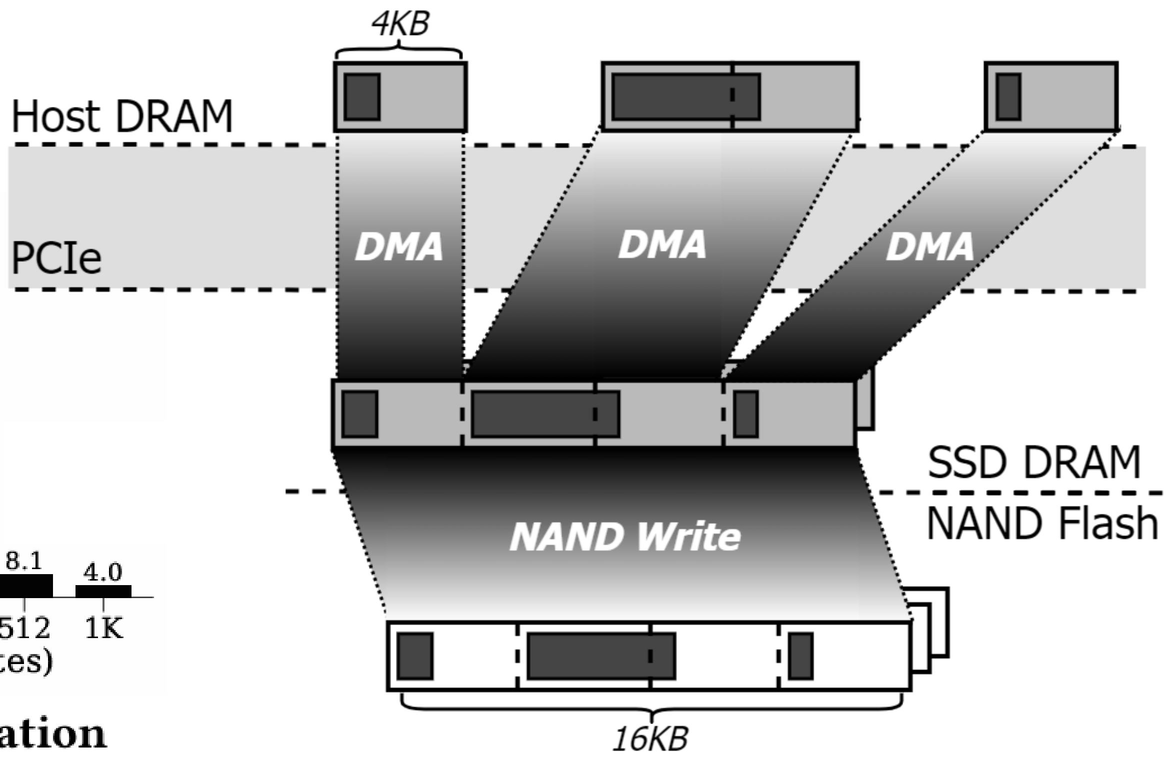
- The packing of received payloads (values) into NAND pages within NVMe SSDs also occurs in units of 4 KiB.
  - This in-device page-unit packing clearly clashes with KV-SSDs, leading to severe NAND write amplification.



(a) Total NAND I/O & Avg. Resp. Time



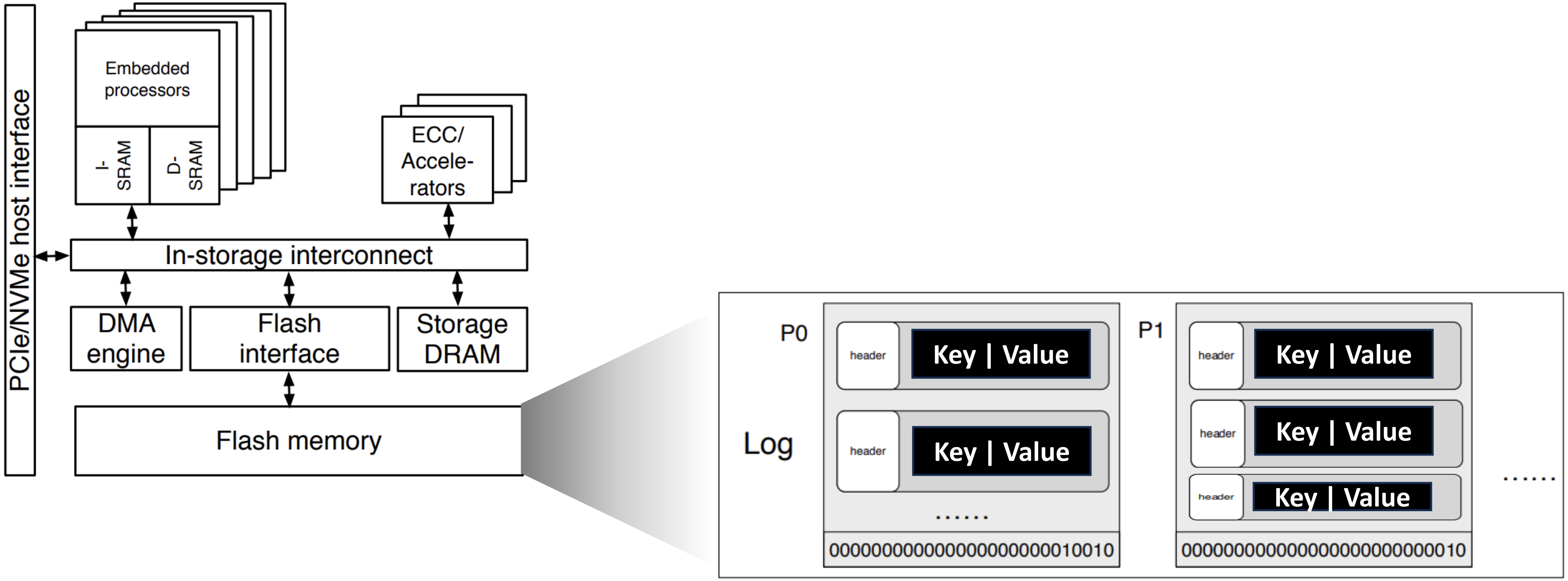
(b) Write Amplification



※ Write Amplification = (value size) / (written bytes)

Problem #2. NAND Write I/O Amplification

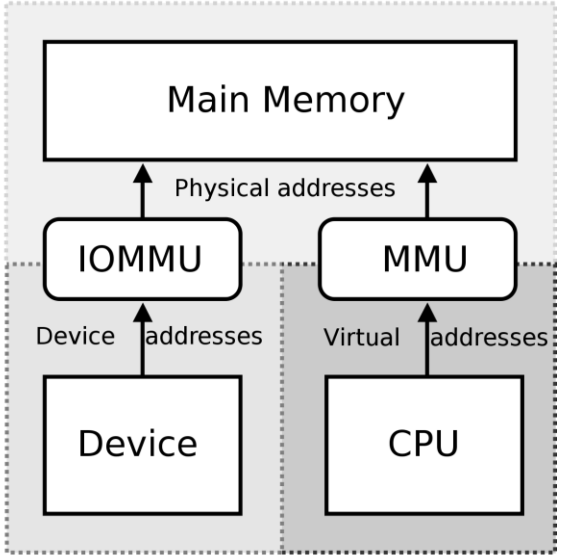
- KAML [7] proposed the batching for multiple values and stored them at the NAND page level in a log-fashion.
  - However, the design for efficiently packing sub-page values was not detailed enough when considering some limitations of real-world storage devices.



[7] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, *KAML: A Flexible, High-Performance Key-Value SSD, in Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture* (HPCA), Feb. 2017.

Problem #2. NAND Write I/O Amplification

- **Limitation.** some DMA engines in real-world SSDs, including our testbed, require that the transfer size and destination addresses be page-aligned [8].
  - This is because the assumption that the payload is multiple blocks guided the storage stack to be optimized for block-size transfer from memory allocations for DMA in the both-side to the DMA engine within the device.
    - Ex) **IOMMU** (Input/Output Memory Management Unit)

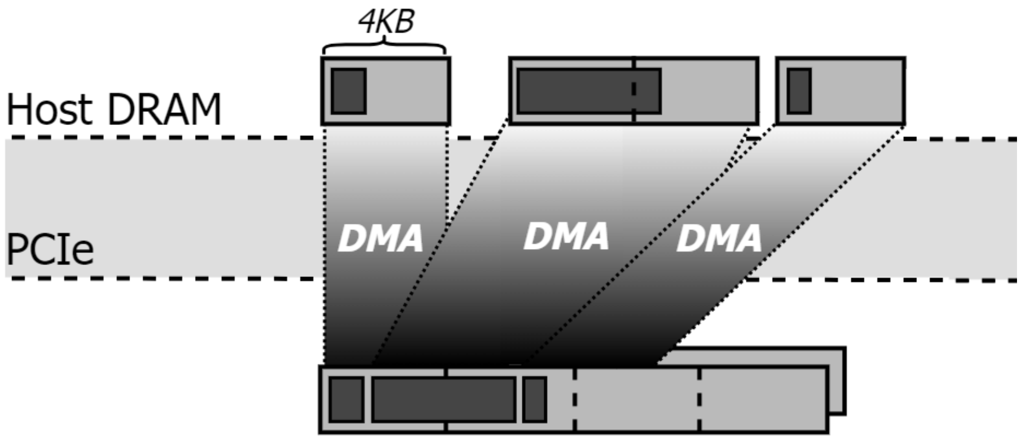


implicit page-unit restrictions on DMA

[8] W. Kwon, S.-W. Sok, C.-H. Park, M.-H. Oh, and S. Hong. 2022. *Gen-Z memory pool system implementation and performance measurement. ETRI Journal 44* (2022), 450–461. Issue 3
[9] The Linux Kernel documentation. 2020. Dynamic DMA mapping Guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

Problem #2. NAND Write I/O Amplification

- **Limitation.** some DMA engines in real-world SSDs, including our testbed, require that the transfer size and destination addresses be page-aligned [8]
  - The device drivers are typically designed to accommodate this requirement [9].

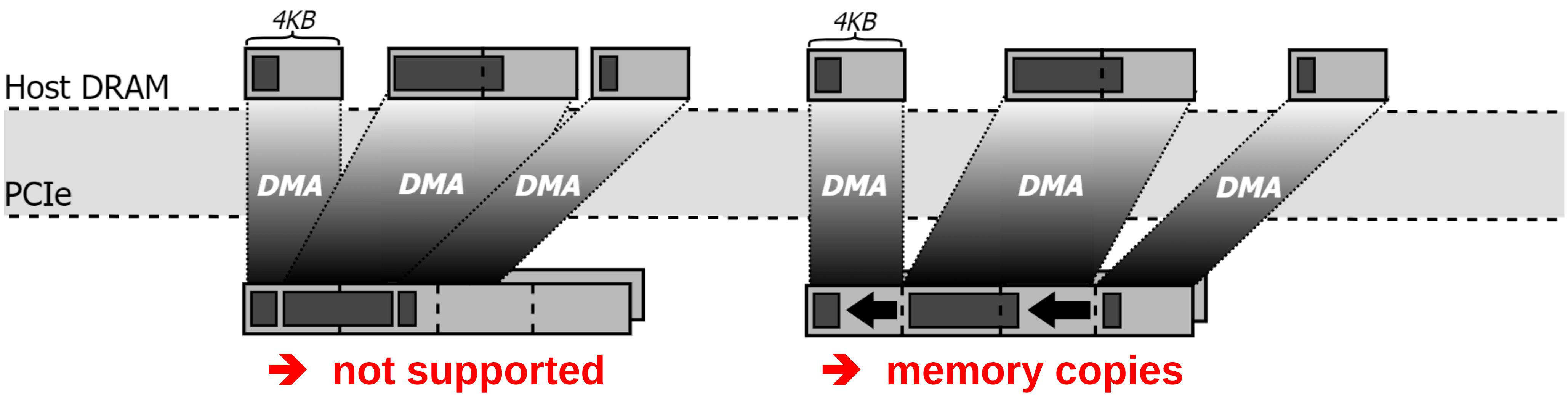


➔ **not supported**

[8] W. Kwon, S.-W. Sok, C.-H. Park, M.-H. Oh, and S. Hong. 2022. *Gen-Z memory pool system implementation and performance measurement. ETRI Journal 44* (2022), 450–461. Issue 3
[9] The Linux Kernel documentation. 2020. Dynamic DMA mapping Guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

Problem #2. NAND Write I/O Amplification

- **Limitation.** some DMA engines in real-world SSDs, including our testbed, require that the transfer size and destination addresses be page-aligned [8].
  - Therefore, fine-grained value packing (logging) within the NAND page buffer **necessitates memory copies** extensively using device’s compute resources.

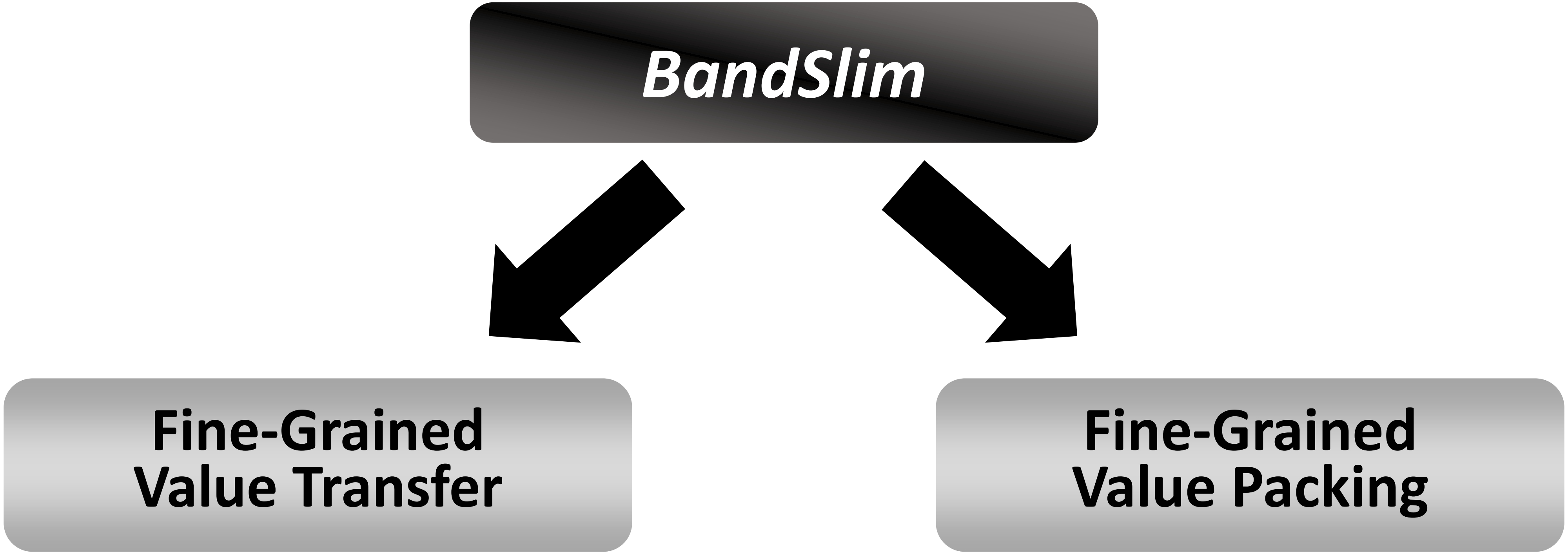


[8] W. Kwon, S.-W. Sok, C.-H. Park, M.-H. Oh, and S. Hong. 2022. *Gen-Z memory pool system implementation and performance measurement. ETRI Journal 44* (2022), 450–461. Issue 3
[9] The Linux Kernel documentation. 2020. Dynamic DMA mapping Guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

Proposed Solution: *BandSlim*

Proposed Solution: ***BandSlim***

- To tackle both amplifications occurring in small key-value transfer and storing NAND flash pages, we introduce ***BandSlim***.



(1) Fine-Grained Value Transfer

- ***BandSlim*** employs a fine-grained inline value transfer mechanism that piggybacks values smaller than a memory page size to NVMe commands using the reserved fields (gray-colored in Figure (a)&(b)).

<table>
<tr><th>dword</th><th colspan="5">description</th></tr>
<tr><td>dword0</td><td>commandID</td><td>P</td><td></td><td>F</td><td>opcode</td></tr>
<tr><td>dword1</td><td colspan="5">namespaceID</td></tr>
<tr><td>dword2</td><td colspan="5" rowspan="2">key</td></tr>
<tr><td>dword3</td></tr>
<tr><td>dword4</td><td colspan="5" rowspan="2">metadataPointer (PRP)</td></tr>
<tr><td>dword5</td></tr>
<tr><td>dword6</td><td colspan="5" rowspan="2">PRPlistEntry1</td></tr>
<tr><td>dword7</td></tr>
<tr><td>dword8</td><td colspan="5" rowspan="2">PRPlistEntry2</td></tr>
<tr><td>dword9</td></tr>
<tr><td>dword10</td><td colspan="5">valueSize</td></tr>
<tr><td>dword11</td><td>reserved</td><td colspan="3">option</td><td>keySize</td></tr>
<tr><td>dword12</td><td colspan="5" rowspan="2">reserved</td></tr>
<tr><td>dword13</td></tr>
<tr><td>dword14</td><td colspan="5" rowspan="2">key</td></tr>
<tr><td>dword15</td></tr>
</table>

(a) Write Command

<table>
<tr><th>dword</th><th colspan="5">description</th></tr>
<tr><td>dword0</td><td>commandID</td><td>P</td><td></td><td>F</td><td>opcode</td></tr>
<tr><td>dword1</td><td colspan="5">namespaceID</td></tr>
<tr><td>dword2</td><td colspan="5" rowspan="2">key</td></tr>
<tr><td>dword3</td></tr>
<tr><td>dword4</td><td colspan="5" rowspan="2">metadataPointer (PRP)</td></tr>
<tr><td>dword5</td></tr>
<tr><td>dword6</td><td colspan="5" rowspan="2">PRPlistEntry1</td></tr>
<tr><td>dword7</td></tr>
<tr><td>dword8</td><td colspan="5" rowspan="2">PRPlistEntry2</td></tr>
<tr><td>dword9</td></tr>
<tr><td>dword10</td><td colspan="5">valueSize</td></tr>
<tr><td>dword11</td><td>reserved</td><td colspan="3">option</td><td>keySize</td></tr>
<tr><td>dword12</td><td colspan="5" rowspan="2">reserved</td></tr>
<tr><td>dword13</td></tr>
<tr><td>dword14</td><td colspan="5" rowspan="2">key</td></tr>
<tr><td>dword15</td></tr>
</table>

(b) Transfer Command

(1) Fine-Grained Value Transfer

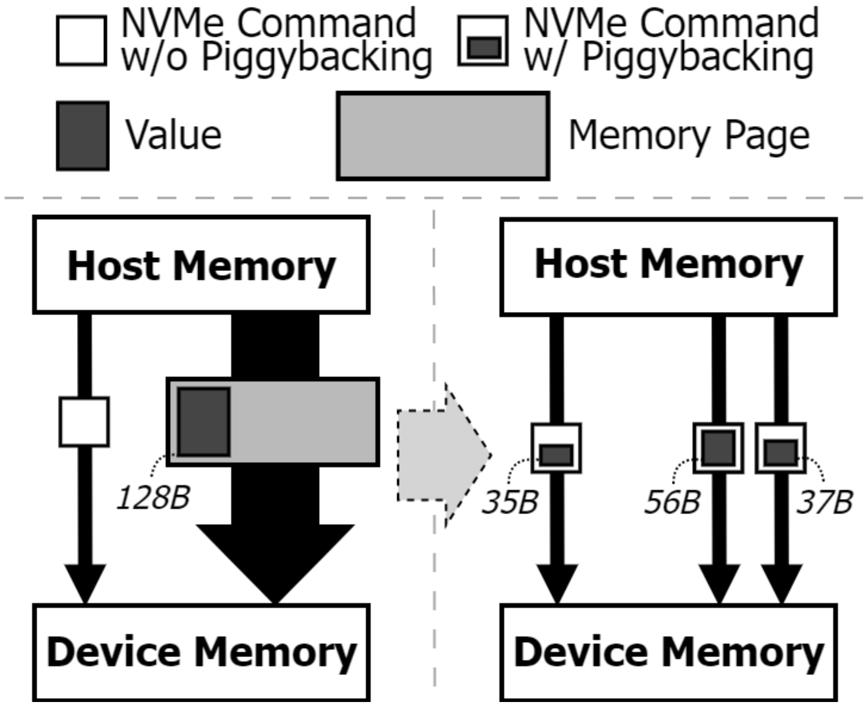
- ***BandSlim*** employs a fine-grained inline value transfer mechanism that piggybacks values smaller than a memory page size to NVMe commands using the reserved fields (gray-colored in Figure (a)&(b)).

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

(a) Write Command

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

(b) Transfer Command



(1) Fine-Grained Value Transfer

- ***BandSlim*** employs a fine-grained inline value transfer mechanism that piggybacks values smaller than a memory page size to NVMe commands using the reserved fields (gray-colored in Figure (a)&(b)).

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

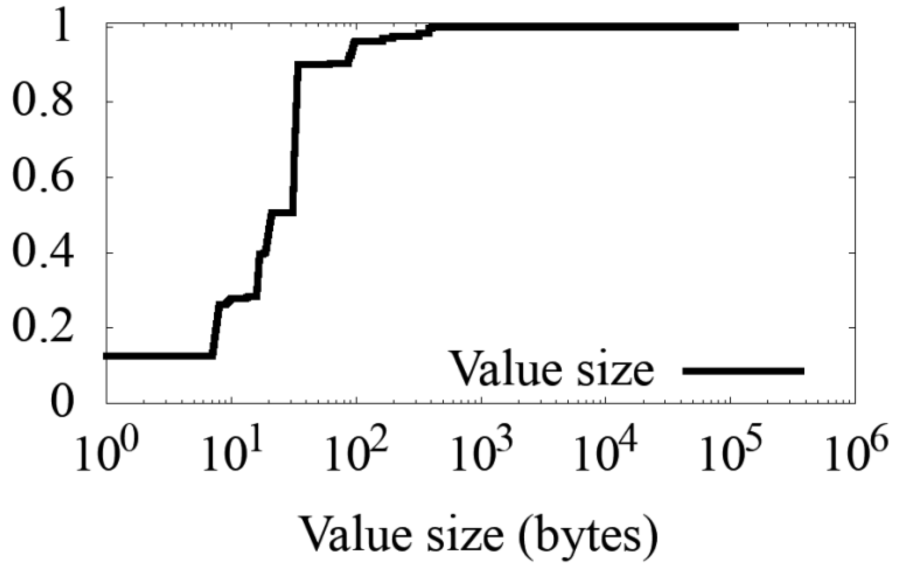


Figure – Value Size CDF for RocksDB in a production environment

(1) Fine-Grained Value Transfer

- ***BandSlim*** employs a fine-grained inline value transfer mechanism that piggybacks values smaller than a memory page size to NVMe commands using the reserved fields (gray-colored in Figure (a)&(b)).

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

64 B NVMe command gives an opportunity

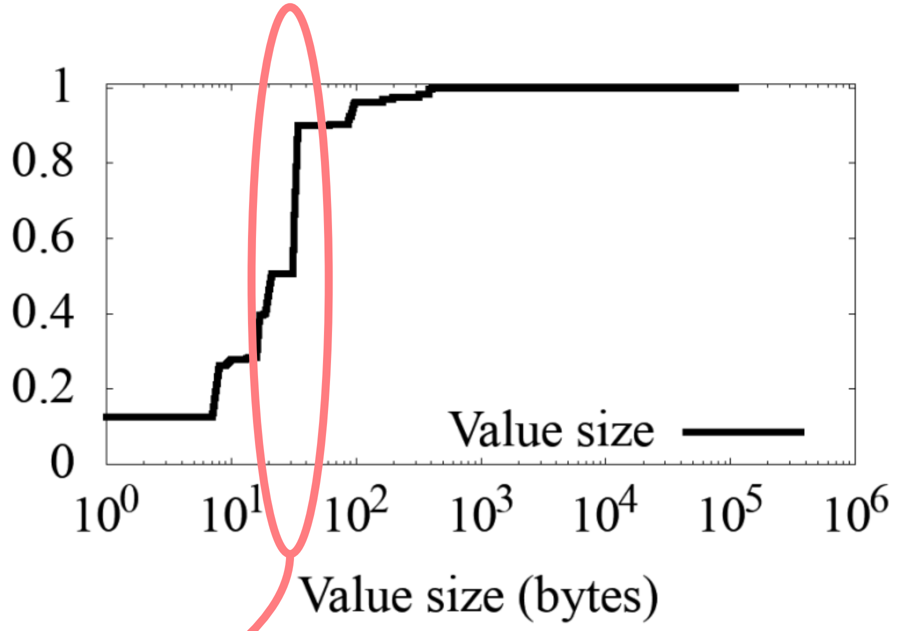
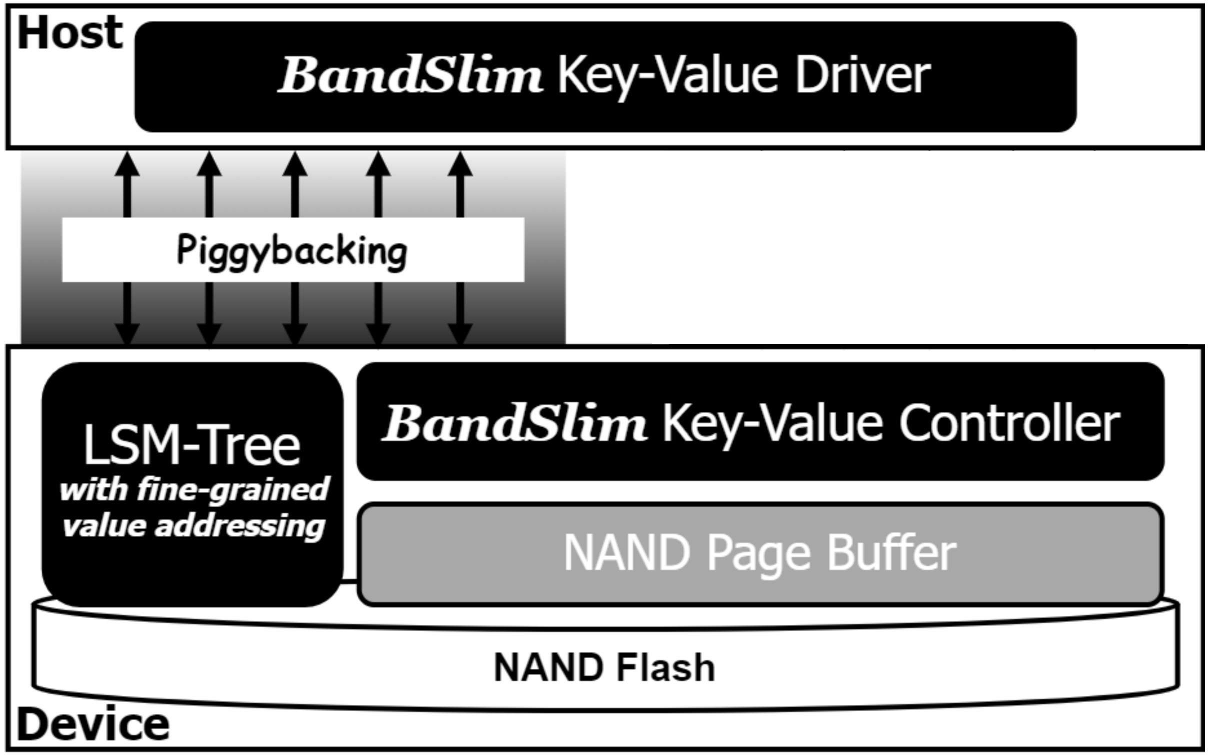


Figure – Value Size CDF for RocksDB in a production environment

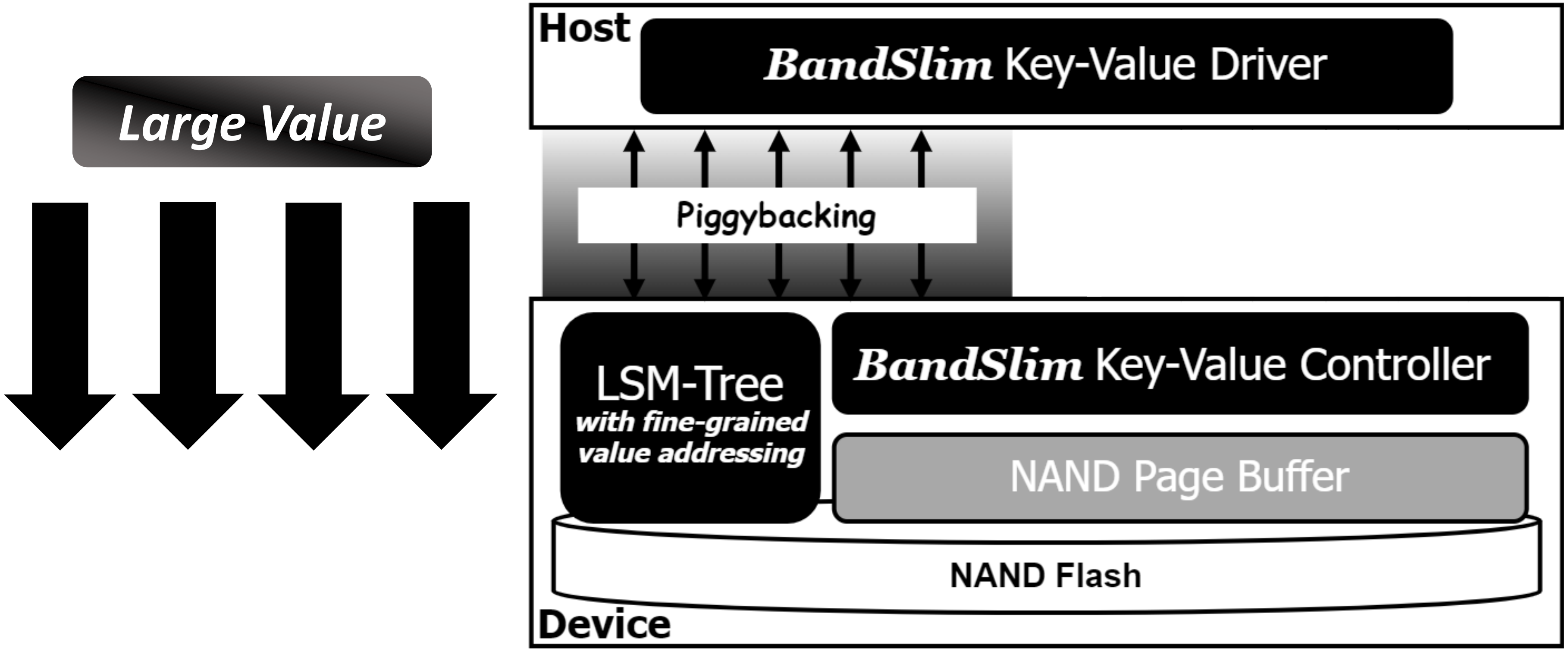
(1) Adaptive Value Transfer Optimization

- When transmitting large values, generating and sending multiple NVMe commands in this manner can result in longer response times.
  - Thus, ***BandSlim*** also incorporates an <u>*adaptive value transfer*</u> strategy that switches back and forth piggybacking and page-unit DMA.



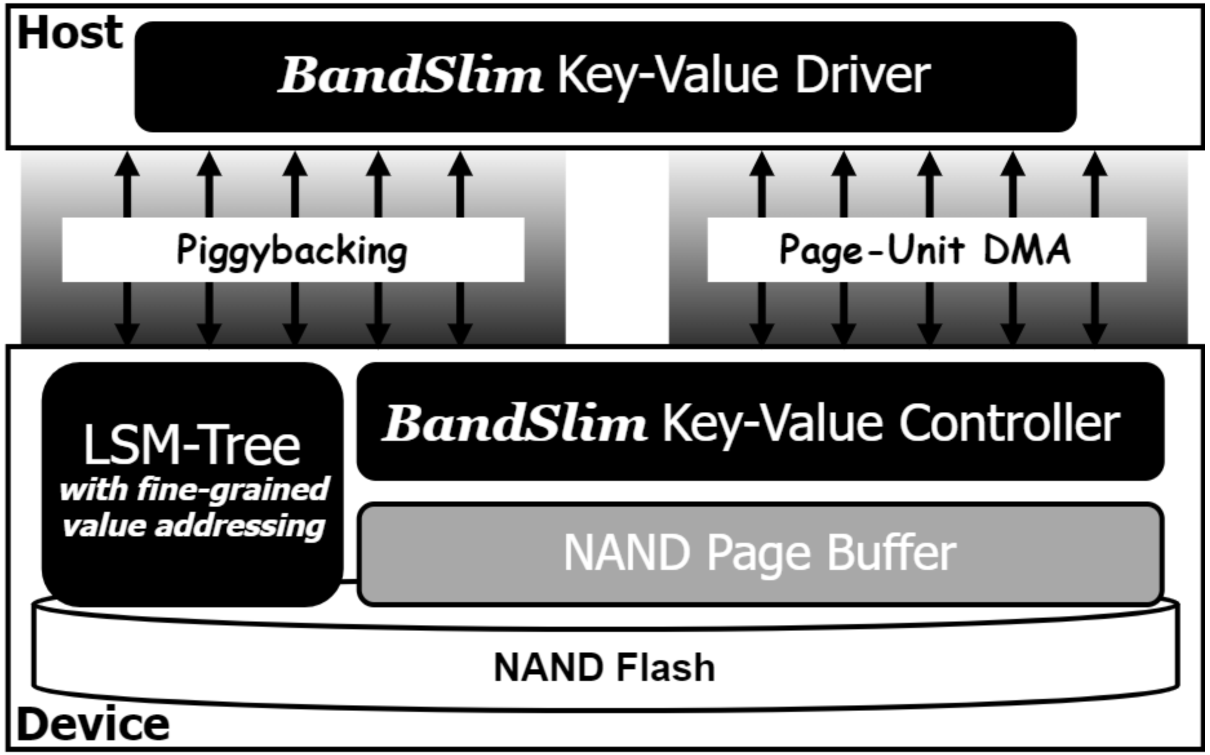
(1) Adaptive Value Transfer Optimization

- When transmitting large values, generating and sending multiple NVMe commands in this manner can result in longer response times.
  - Thus, ***BandSlim*** also incorporates an *adaptive value transfer* strategy that switches back and forth piggybacking and page-unit DMA.



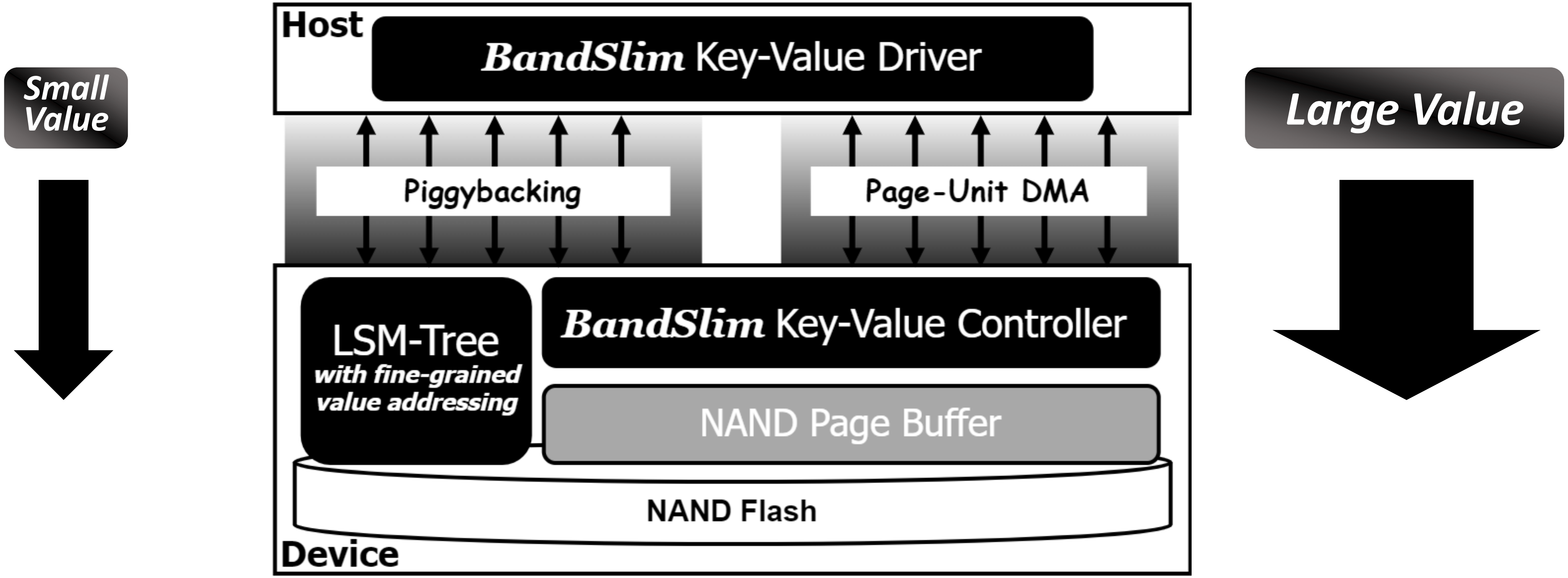
(1) Adaptive Value Transfer Optimization

- When transmitting large values, generating and sending multiple NVMe commands in this manner can result in longer response times.
  - Thus, ***BandSlim*** also incorporates an *adaptive value transfer* strategy that switches back and forth piggybacking and page-unit DMA.



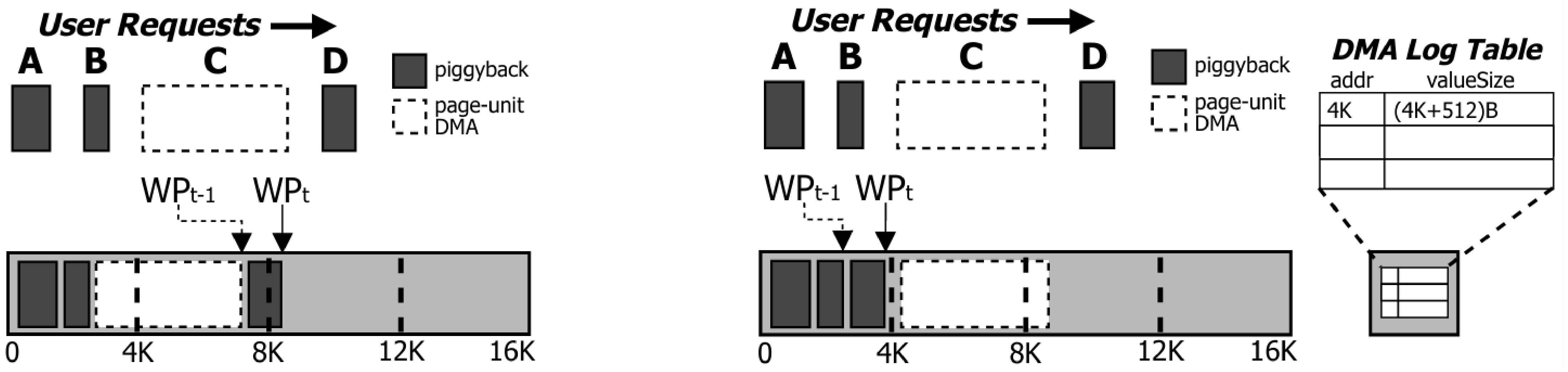
(1) Adaptive Value Transfer Optimization

- When transmitting large values, generating and sending multiple NVMe commands in this manner can result in longer response times.
  - Thus, ***BandSlim*** also incorporates an *adaptive value transfer* strategy that switches back and forth piggybacking and page-unit DMA.



(2) Fine-Grained Value Packing

- ***BandSlim*** implements a *Selective Packing with Backfilling Policy* locating small values to fill the gap formed by the page-aligned, DMA-transferred value under the adaptive value transfer method.

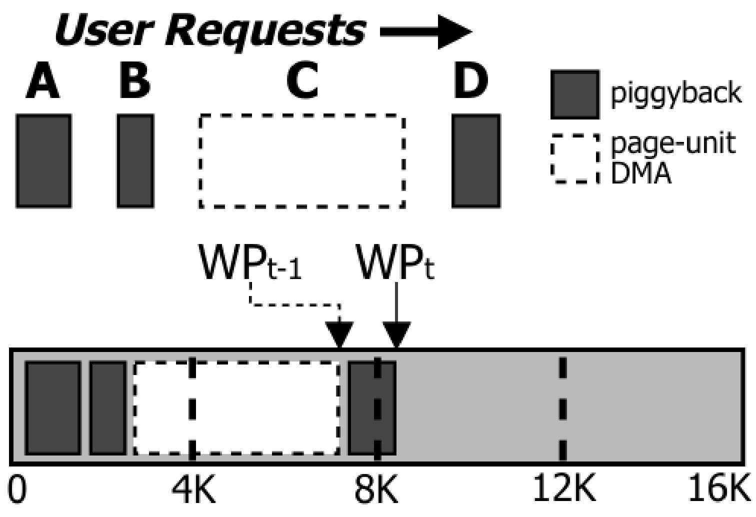


(a) All Packing *from KAML*

(b) Selective Packing w/ Backfilling

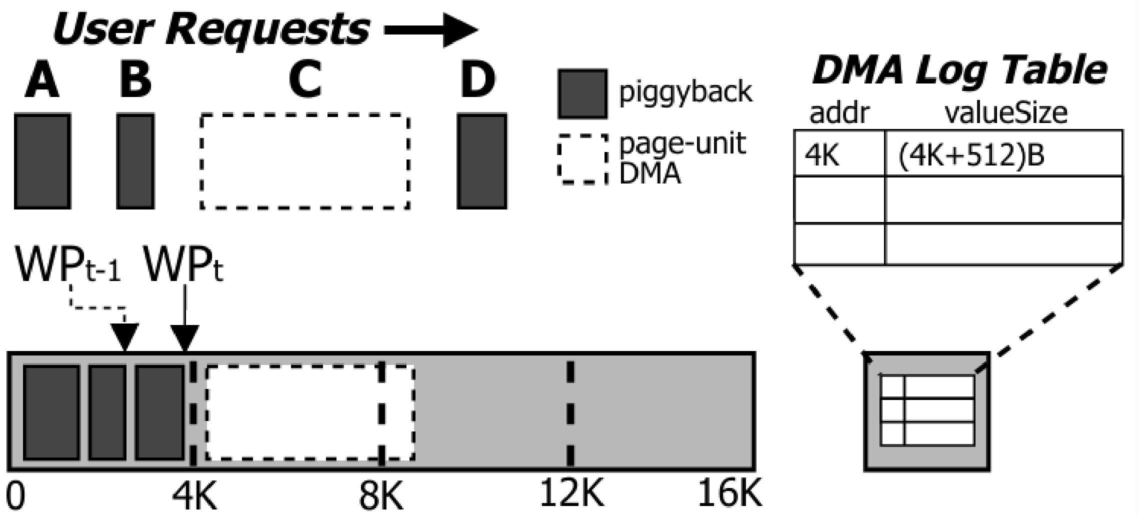
(2) Fine-Grained Value Packing

- ***BandSlim*** implements a *Selective Packing with Backfilling Policy* locating small values to fill the gap formed by the page-aligned, DMA-transferred value under the adaptive value transfer method.



(a) All Packing *from KAML*

➔ memory copies for large values



(b) Selective Packing w/ Backfilling

➔ NO memory copies for large values

Evaluation

Evaluation Setup

- Testbed:

KV-SSD on Cosmos+ OpenSSD Platform

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

Evaluation Setup

- Test Configurations:

Baseline	**State-of-the-art LSM-based NVMe KV-SSD, IterKVSSD (Systor ’23).**
Piggyback	**It transfers values using only piggybacking-based transfer method.**
Adaptive	**It transfers values using the adaptive value transfer method.**

Evaluation Setup

- Workloads (Meta's db_bench):

W(A)	**fillseq, 1 million PUTs. The value size does not change.**
W(B)	**fillrandom, 1 million PUTs, value sizes of 8 B or 2 KiB at a 9:1 ratio.**
W(C)	**Same as *W(B)* but with the value size ratio reversed to 1:9.**
W(D)	**fillrandom, 1 million PUTs, values sizes of 8 B, 16 B, 32 B, 64 B, 128 B, 256 B, 512 B, 1 KiB, and 2 KiB with each size having an equal ratio.**
W(M)	**mixgraph (real-world workloads with a maximum value size of 1 KiB and almost 70% of values being under 35 B), 1 million PUTs.**

Evaluation Setup

- Workloads (Meta's db_bench):

W(A)	→ **Fixed Value Size**
W(B)	→ **Small Value Dominant**
W(C)	→ **Large Value Dominant**
W(D)	→ **Balanced Value Size**
W(M)	→ **Real-World Pattern**

(1) Fine-Grained Value Transfer

Sequential Write Workload (*W(A)*)

- *Piggyback* achieves a remarkable reduction in PCIe traffic of up to 97.9%.
- As the value size increases with piggybacking applied, the PCIe traffic and the response time begins to increase due to the addition of trailing commands.

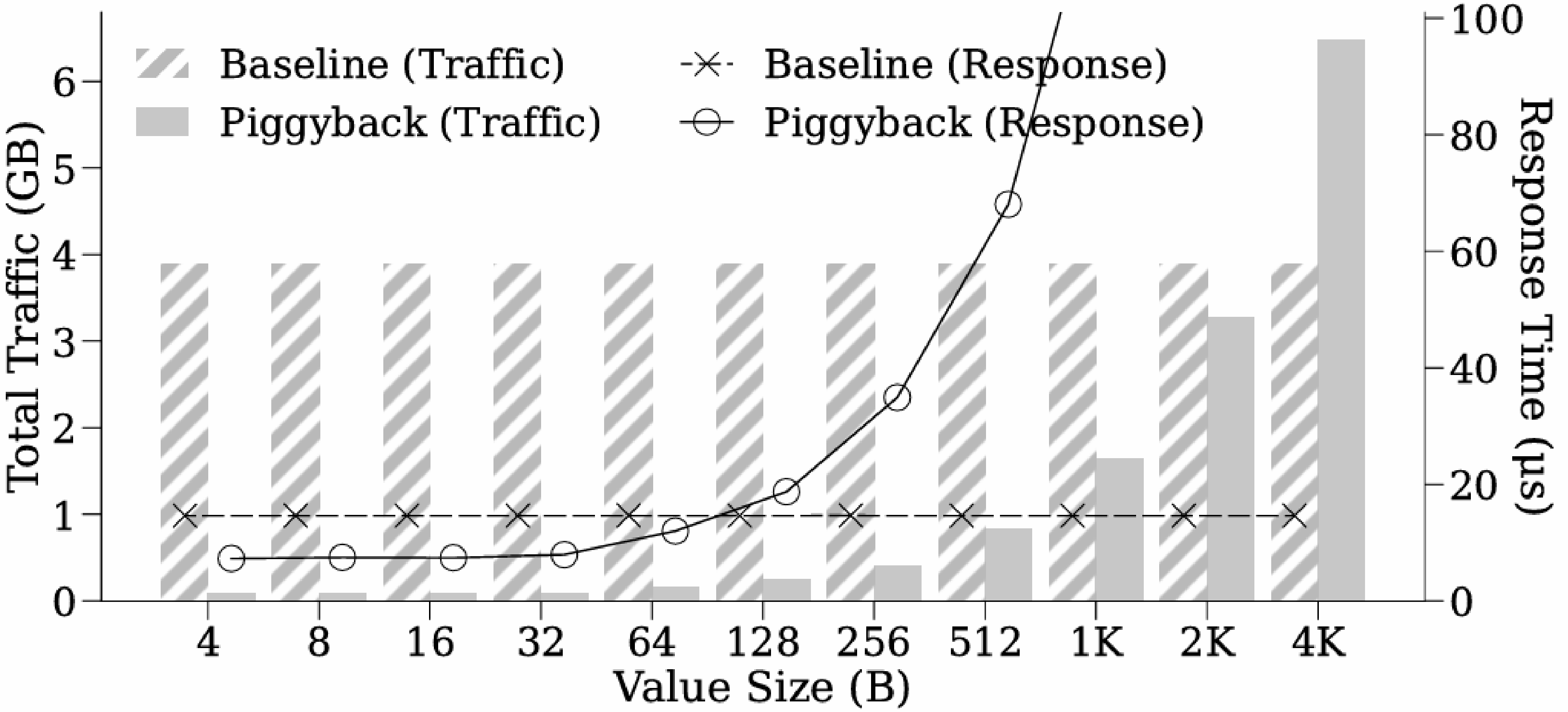


Figure 1. Total PCIe Traffic and Avg. Response Time.

(1) Fine-Grained Value Transfer
Sequential Write Workload (*W(A)*)

- *Piggyback* achieves a remarkable reduction in PCIe traffic of up to 97.9%.
- As the value size increases with piggybacking applied, the PCIe traffic and the response time begins to increase due to the addition of trailing commands.

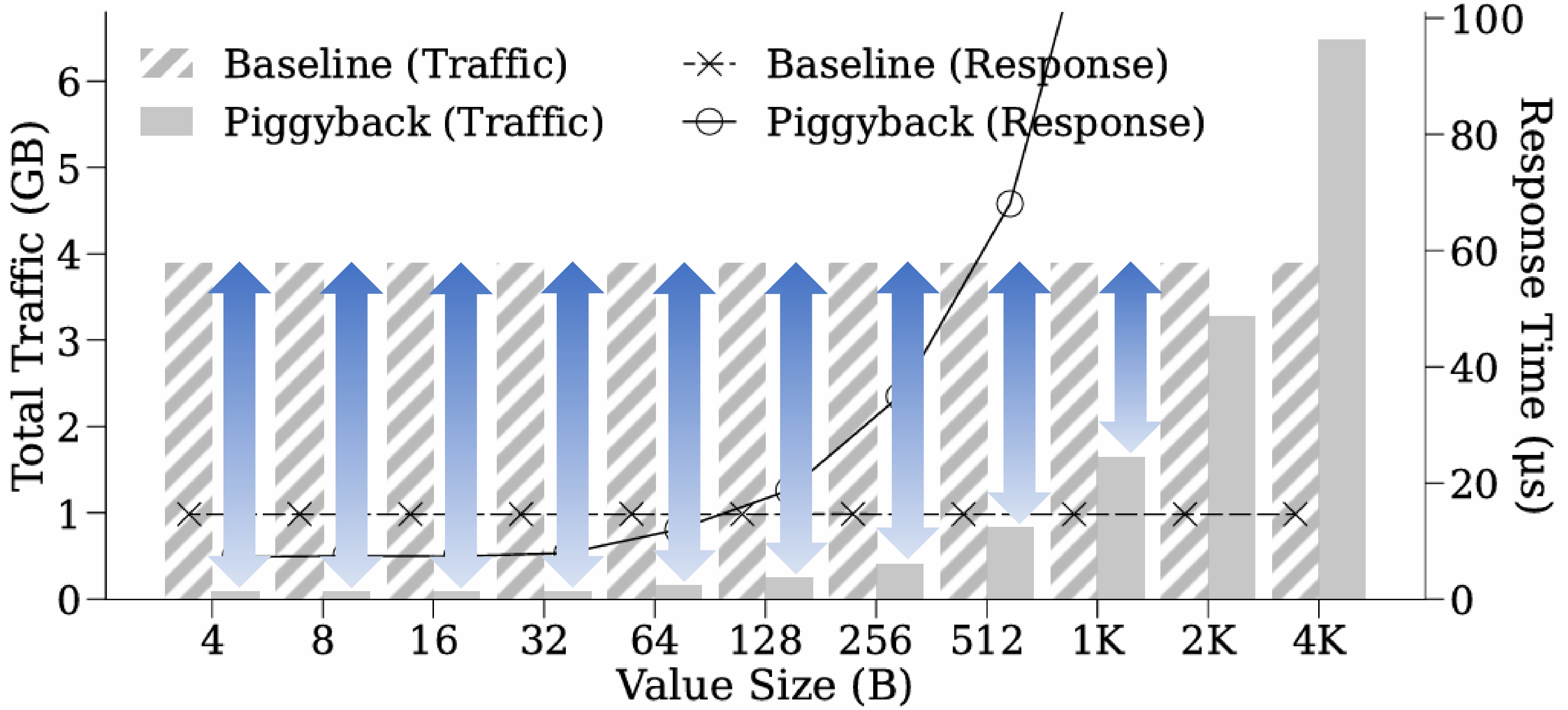


Figure 1. Total PCIe Traffic and Avg. Response Time.

(1) Fine-Grained Value Transfer

Sequential Write Workload (*W(A)*)

- *Piggyback* achieves a remarkable reduction in PCIe traffic of up to 97.9%.
- As the value size increases with piggybacking applied, the PCIe traffic and the response time begins to increase due to the addition of trailing commands.

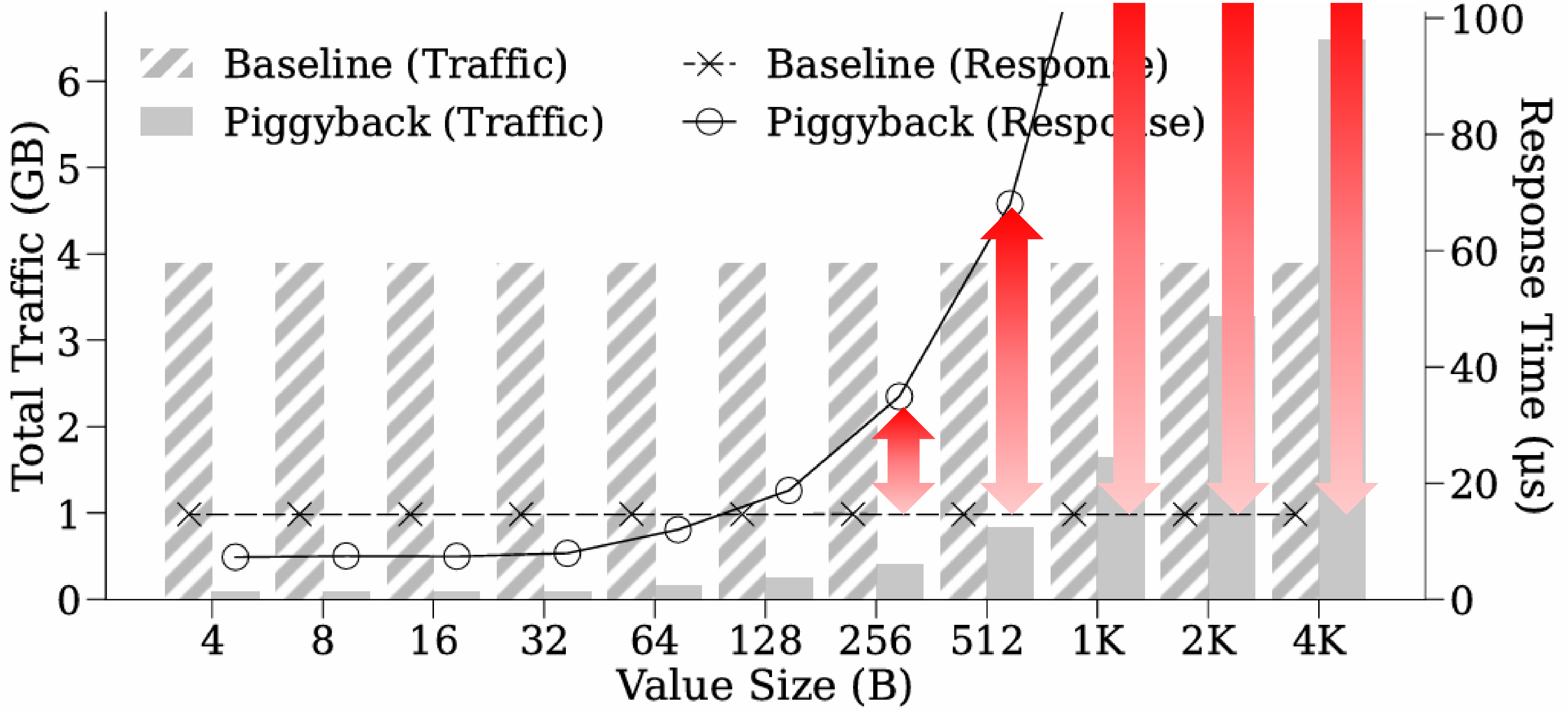
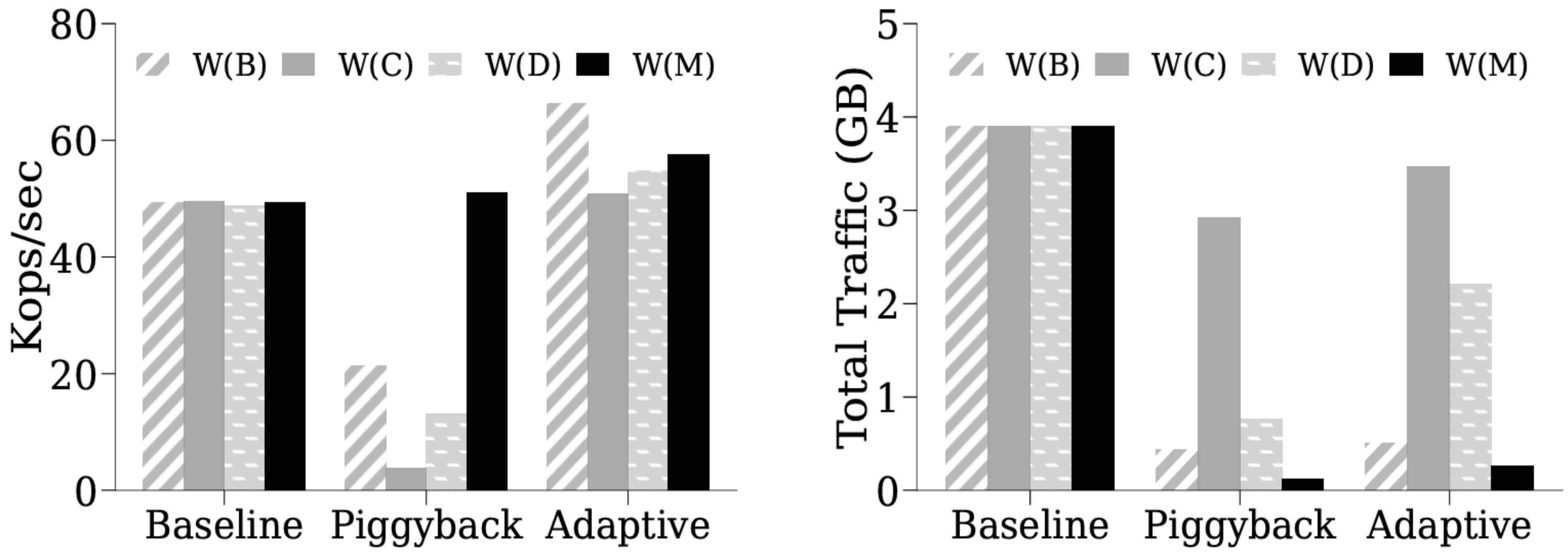


Figure 1. Total PCIe Traffic and Avg. Response Time.

(1) Fine-Grained Value Transfer

Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to *Baseline* for ***W(M)***.
- Above all, *Adaptive* proves to be the best in all workloads.



(a) Avg. Throughput (b) Total PCIe Traffic

Figure 2. Performance analysis of transfer methods.

(1) Fine-Grained Value Transfer
Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to *Baseline* for ***W(M)***.
- Above all, *Adaptive* proves to be the best in all workloads.

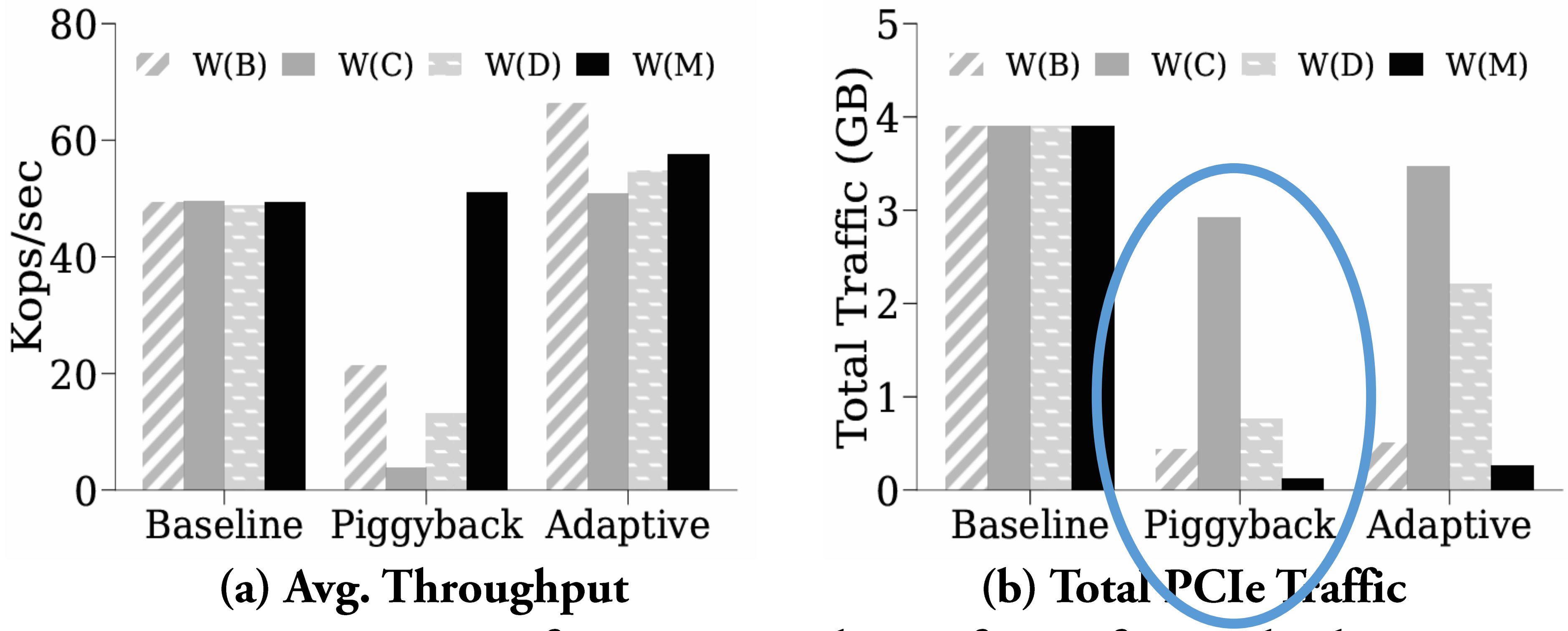


Figure 2. Performance analysis of transfer methods.

(1) Fine-Grained Value Transfer

Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to *Baseline* for ***W(M)***.
- Above all, *Adaptive* proves to be the best in all workloads.

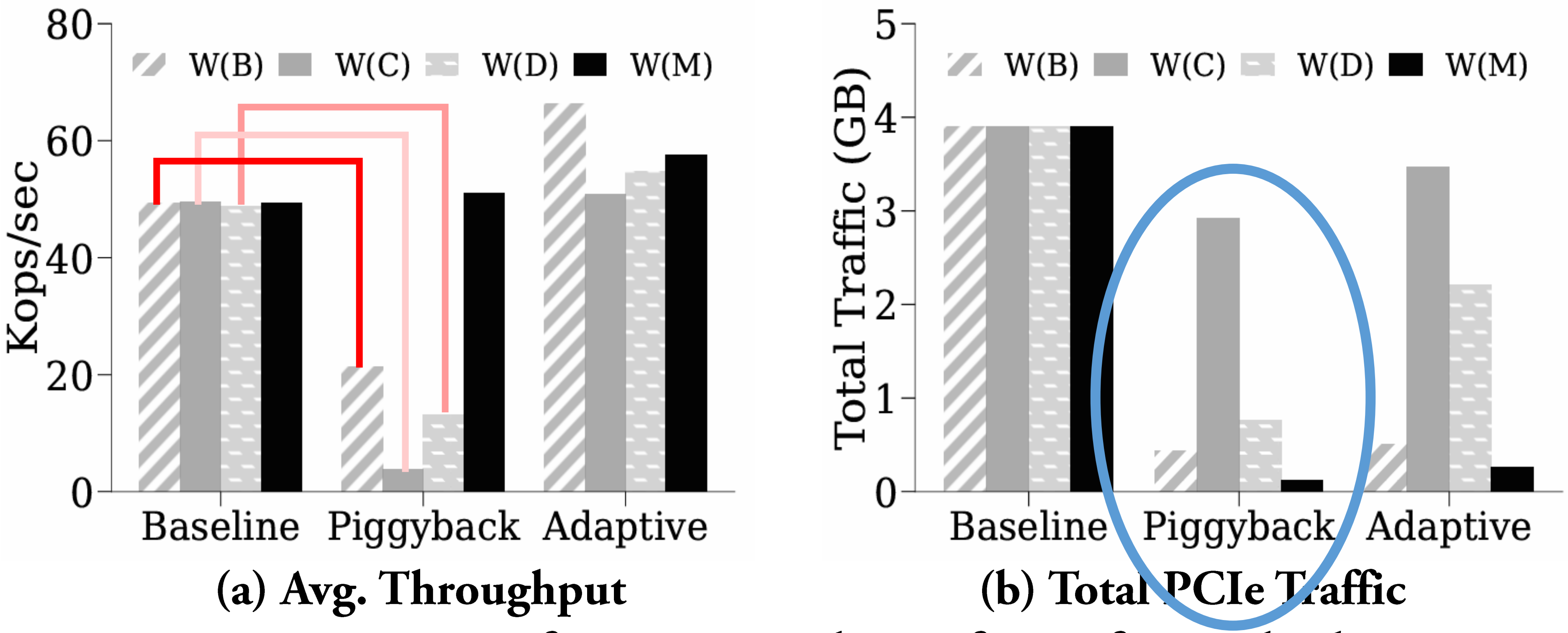
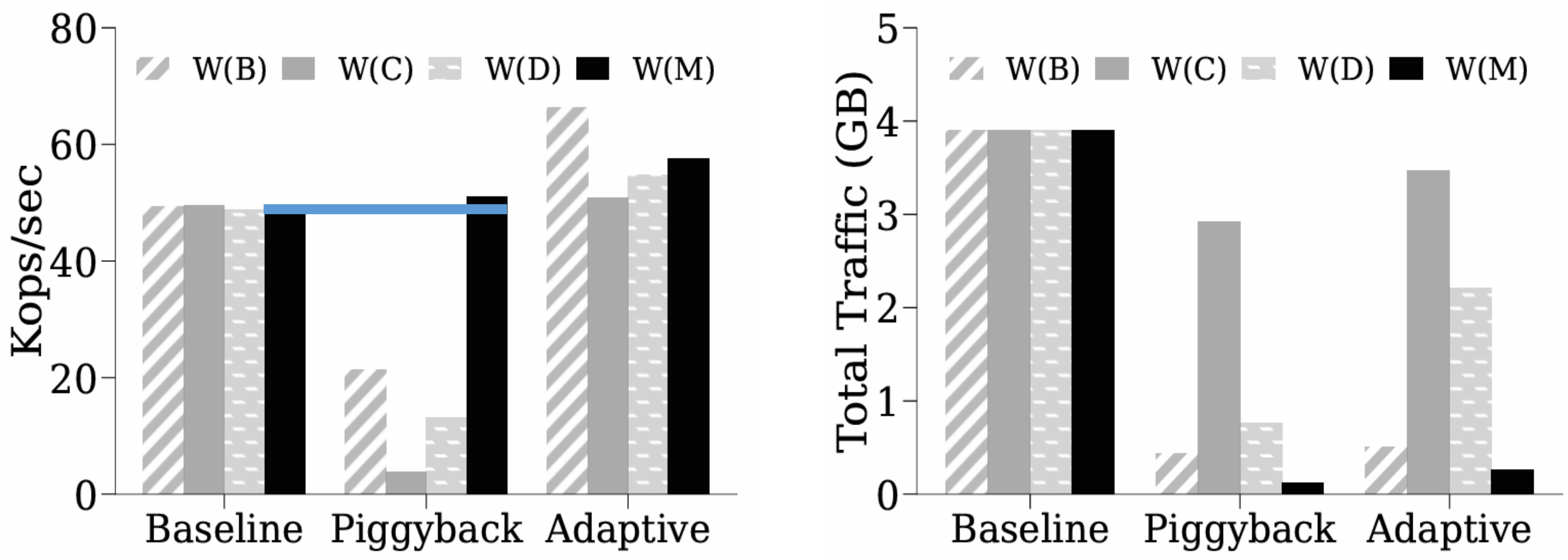


Figure 2. Performance analysis of transfer methods.

(1) Fine-Grained Value Transfer

Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to *Baseline* for ***W(M)***.
- Above all, *Adaptive* proves to be the best in all workloads.



(a) Avg. Throughput (b) Total PCIe Traffic

Figure 2. Performance analysis of transfer methods.

(1) Fine-Grained Value Transfer

Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to *Baseline* for ***W(M)***.
- Above all, *Adaptive* proves to be the best in all workloads.

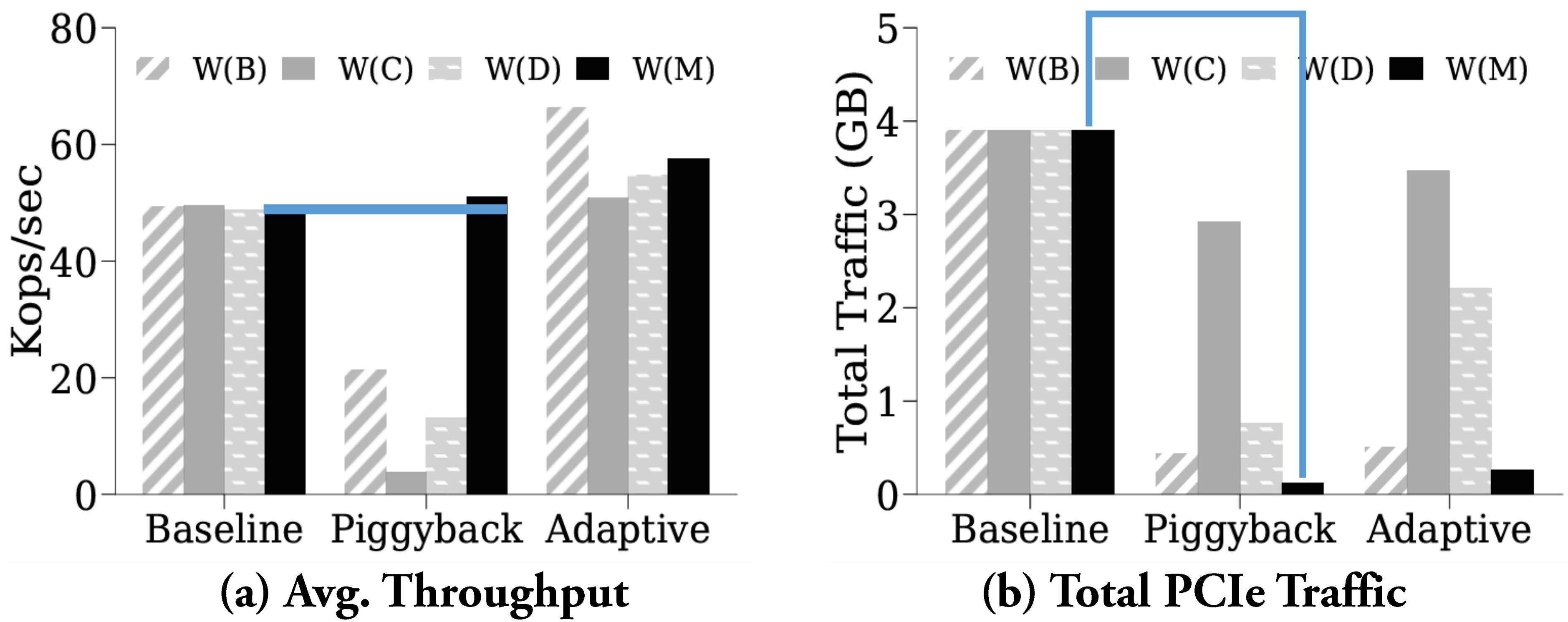


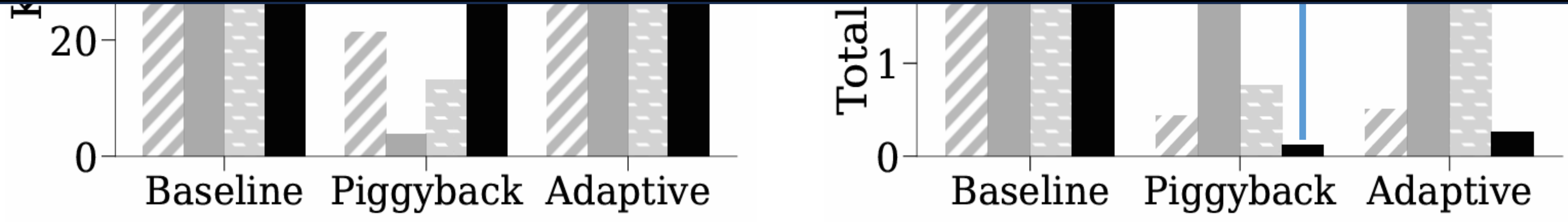
Figure 2. Performance analysis of transfer methods.

(1) Fine-Grained Value Transfer

Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to ***Baseline*** for ***W(M)***.
- Above all, ***Adaptive*** proves to be the best in all workloads.

The proposed approach performs better than the baseline under real-world workloads while reducing PCIe traffic significantly.



(a) Avg. Throughput (b) Total PCIe Traffic

Figure 2. Performance analysis of transfer methods.

(1) Fine-Grained Value Transfer
Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to ***Baseline*** for ***W(M)***.
- Above all, ***Adaptive*** proves to be the best in all workloads.

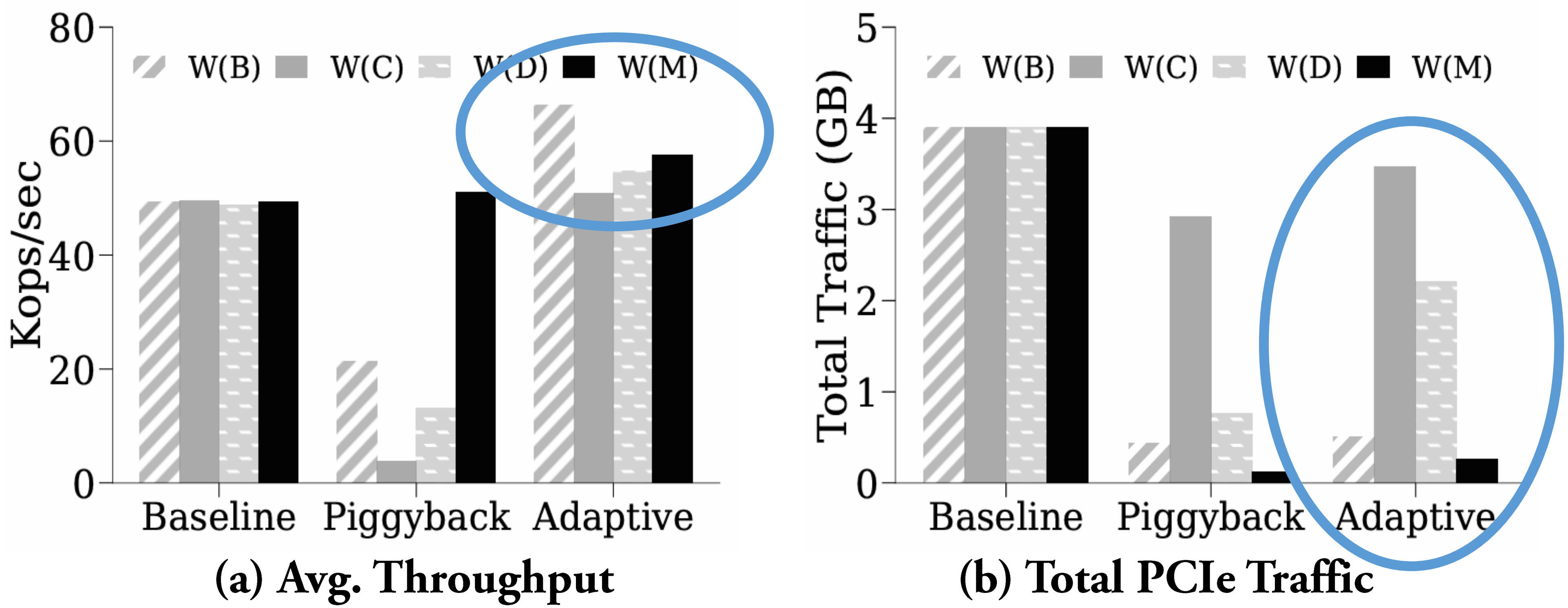
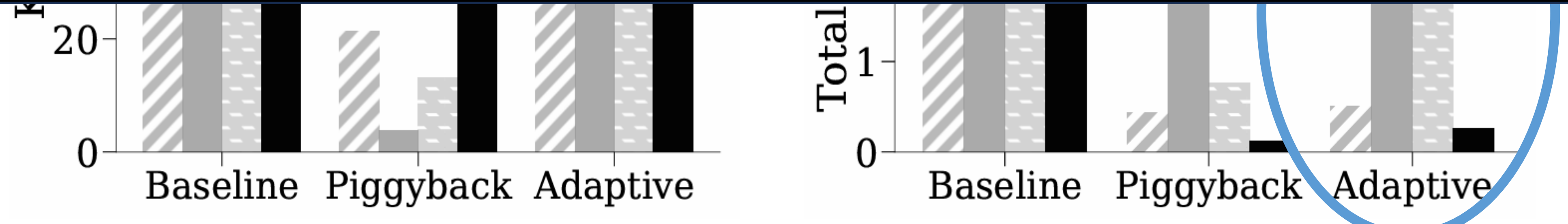


Figure 2. Performance analysis of transfer methods.

(1) Fine-Grained Value Transfer
Various Workloads (*W(B)* ~ *W(M)*)

- Even though *Piggyback* can increase response times greatly, *Piggyback* still improved the average throughput by about 22% compared to *Baseline* for *W(M)*.
- Above all, *Adaptive* proves to be the best in all workloads.

If we cover most of values by piggybacking, and large values by fast DMA, we can achieve an optimal transfer performance.



(a) Avg. Throughput

(b) Total PCIe Traffic

Figure 2. Performance analysis of transfer methods.

Evaluation Setup

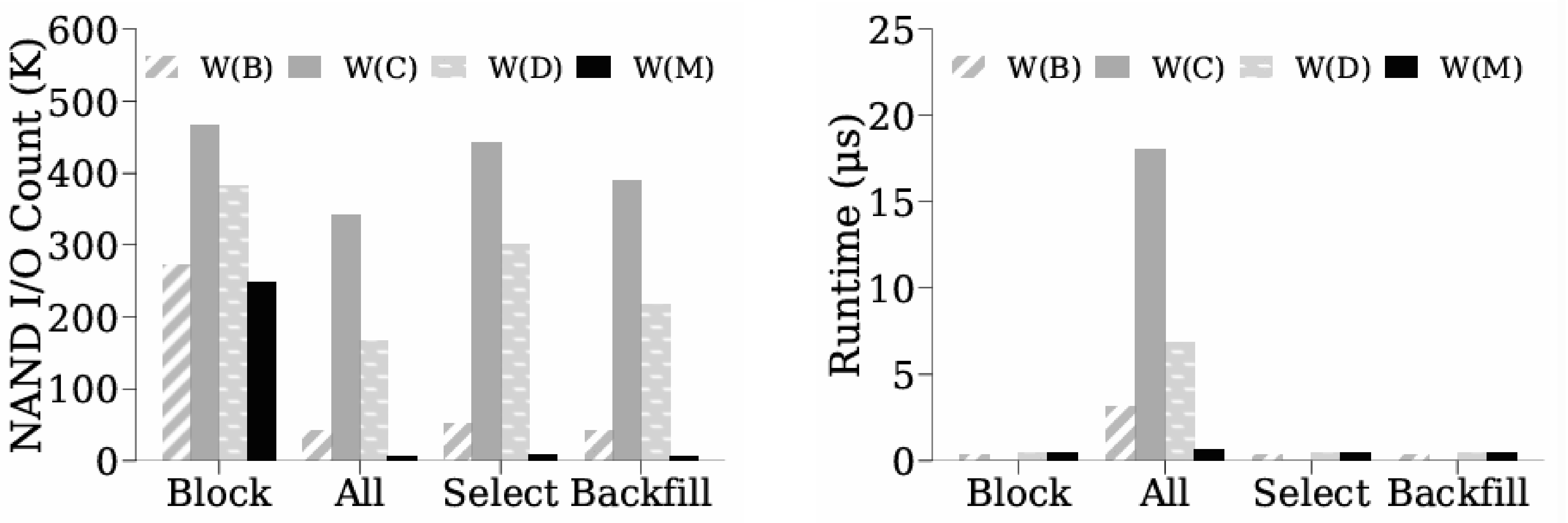
- Test Configurations:

Block	**The baseline block-based page-unit payload packing of NVMe SSDs.**
All	**The *All Packing Policy* from KAML**
Select	**The *Selective Packing Policy* proposed in *BandSlim***
Backfill	**The *Selective Packing with Backfilling Policy* proposed in *BandSlim***

(2) Fine-Grained Value Packing

Various Workloads (***W(B)*** ~ ***W(M)***)

- With packing applied, the total number of NAND writes reduces greatly.
- ***Backfill*** reduces NAND writes as much as ***All*** in small-value-dominant workloads (***W(B)*** & ***W(M)***).



(a) Total NAND I/O Cnt. (b) Avg. Memcpy Time

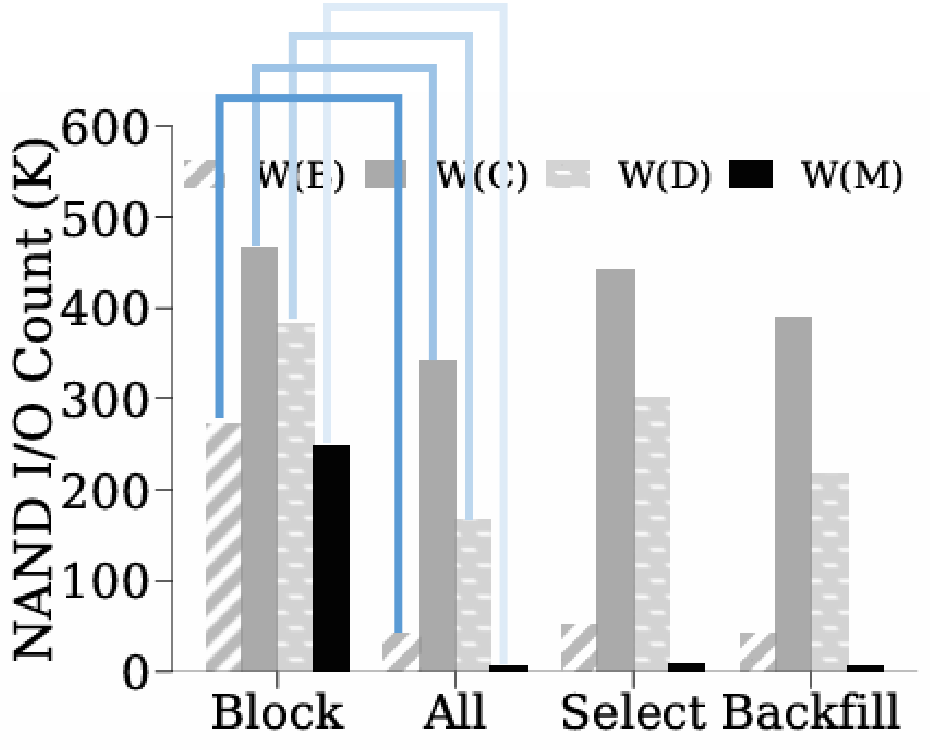
Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

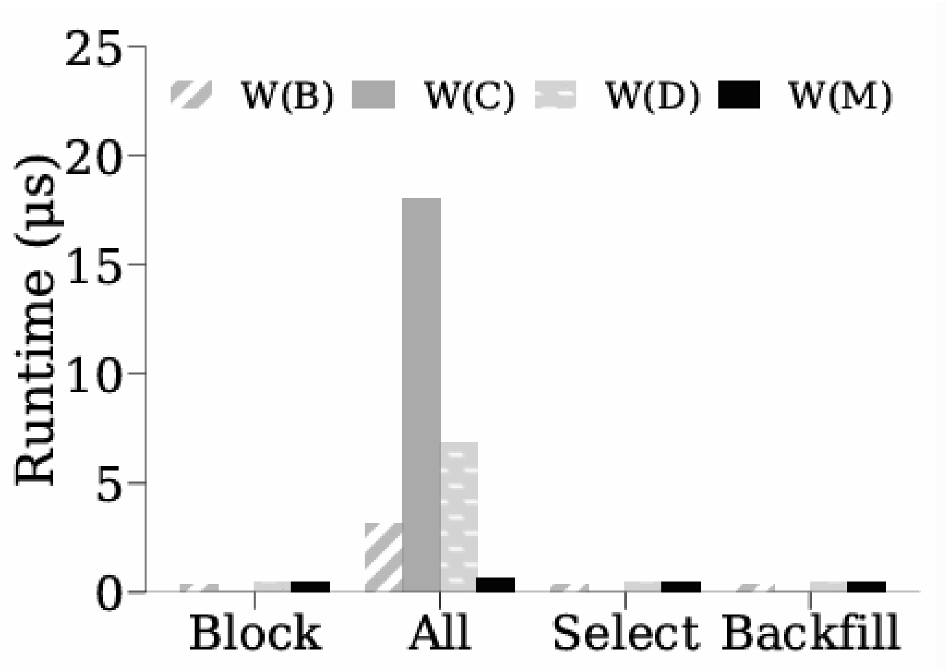
(2) Fine-Grained Value Packing

Various Workloads (***W(B)*** ~ ***W(M)***)

- *Block* shows the worst performance regardless of the workload.
- *Selective* performs as poorly as *Block* in large-value-dominant situations (***W(C)***).



(a) Total NAND I/O Cnt.



(b) Avg. Memcpy Time

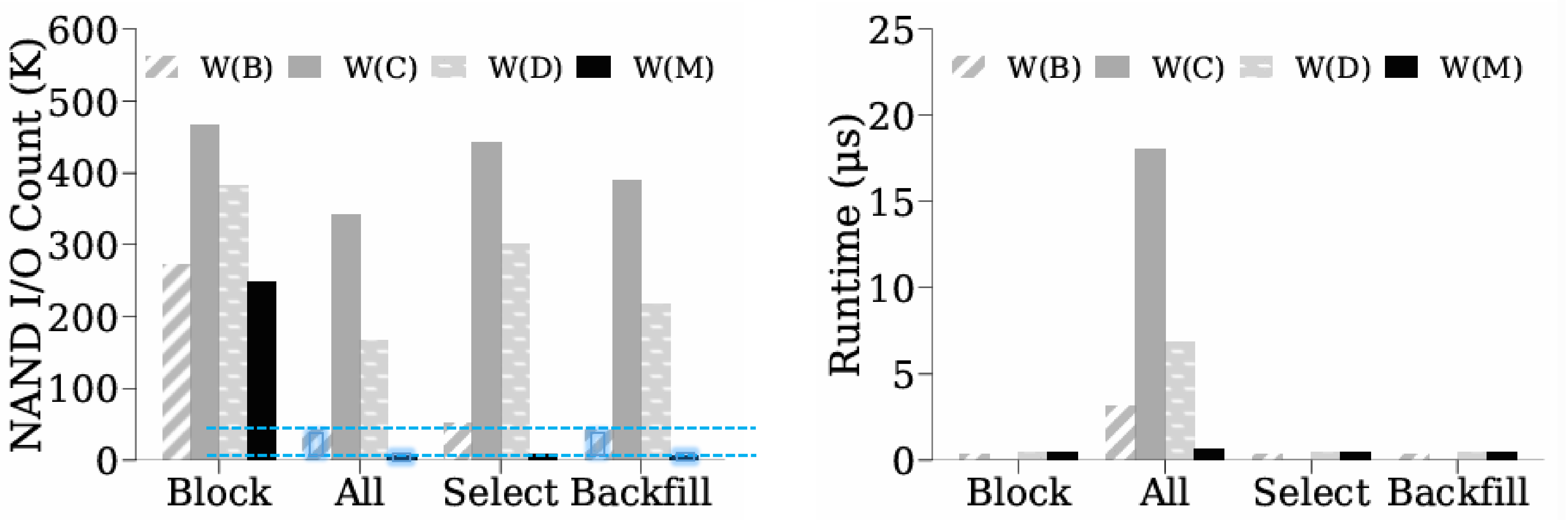
Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

(2) Fine-Grained Value Packing

Various Workloads (*W(B)* ~ *W(M)*)

- *Block* shows the worst performance regardless of the workload.
- *Selective* performs as poorly as *Block* in large-value-dominant situations (***W(C)***).



(a) Total NAND I/O Cnt. (b) Avg. Memcpy Time

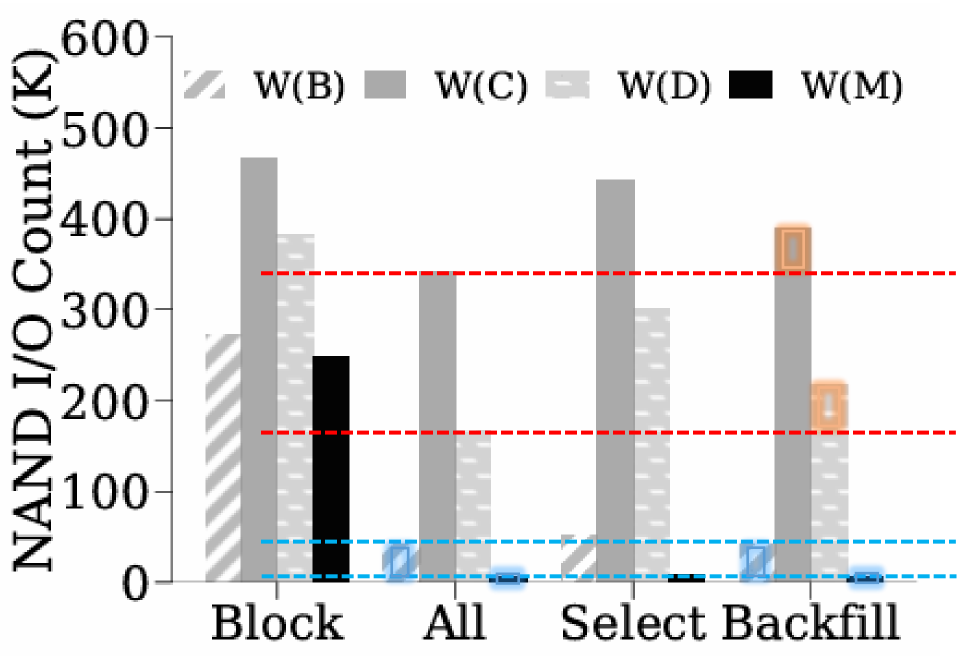
Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

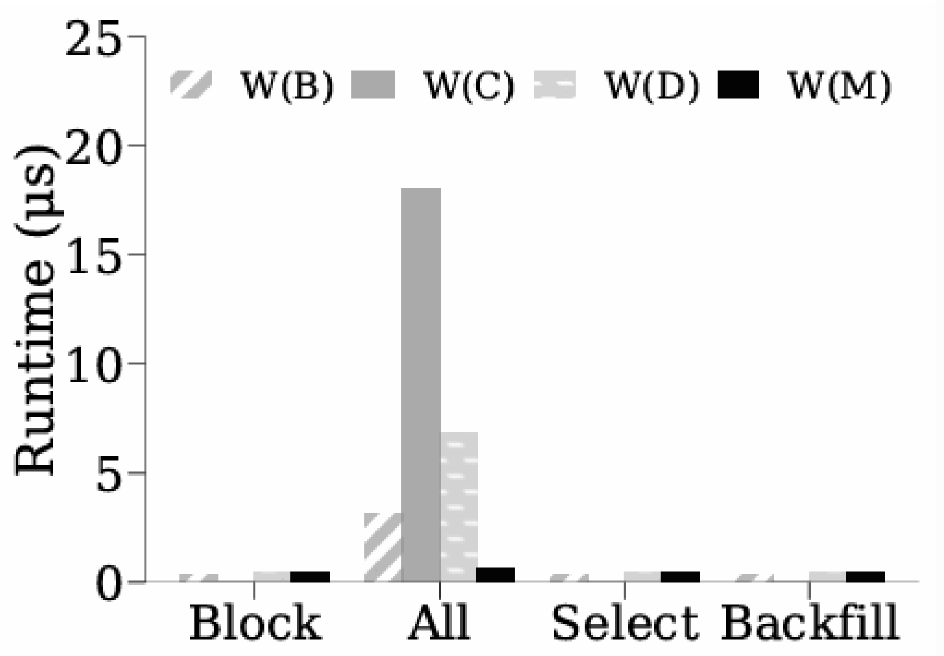
(2) Fine-Grained Value Packing

Various Workloads (*W(B)* ~ *W(M)*)

- *Block* shows the worst performance regardless of the workload.
- *Selective* performs as poorly as *Block* in large-value-dominant situations (***W(C)***).



(a) Total NAND I/O Cnt.



(b) Avg. Memcpy Time

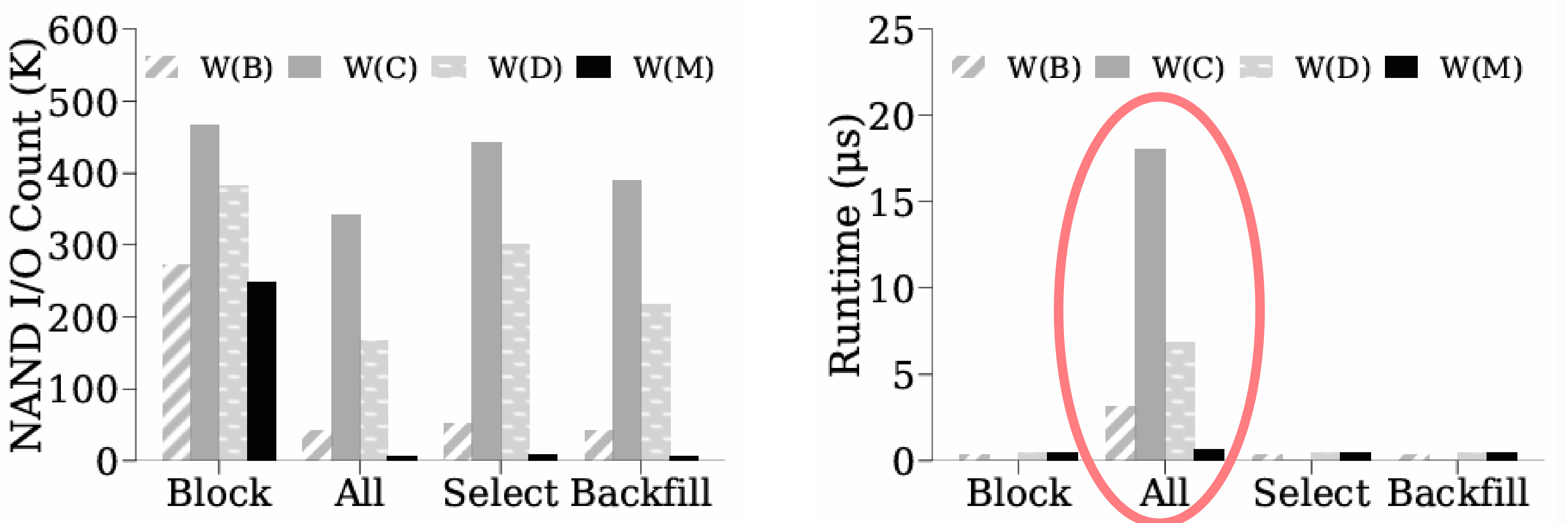
Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

(2) Fine-Grained Value Packing

Various Workloads (*W(B)* ~ *W(M)*)

- *Block* shows the worst performance regardless of the workload.
- *Selective* performs as poorly as *Block* in large-value-dominant situations (***W(C)***).



(a) Total NAND I/O Cnt. (b) Avg. Memcpy Time

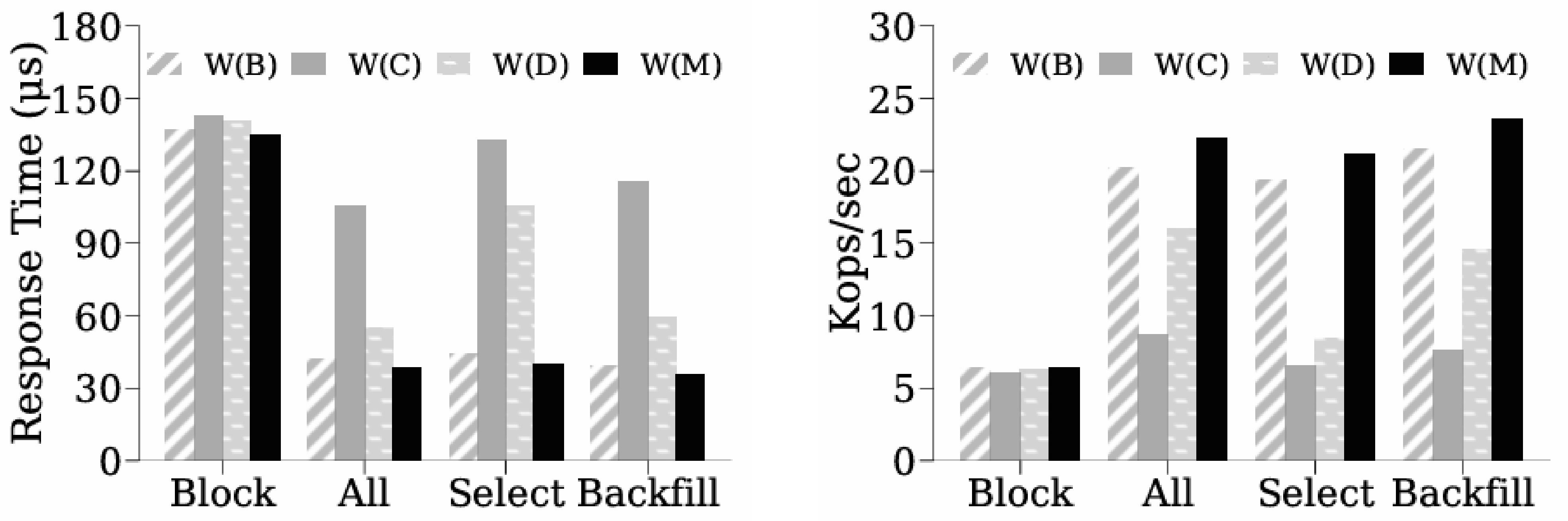
Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

(2) Fine-Grained Value Packing

Various Workloads (*W(B)* ~ *W(M)*)

- *Block* shows the worst performance regardless of the workload.
- *Selective* performs as poorly as *Block* in large-value-dominant situations (***W(C)***).



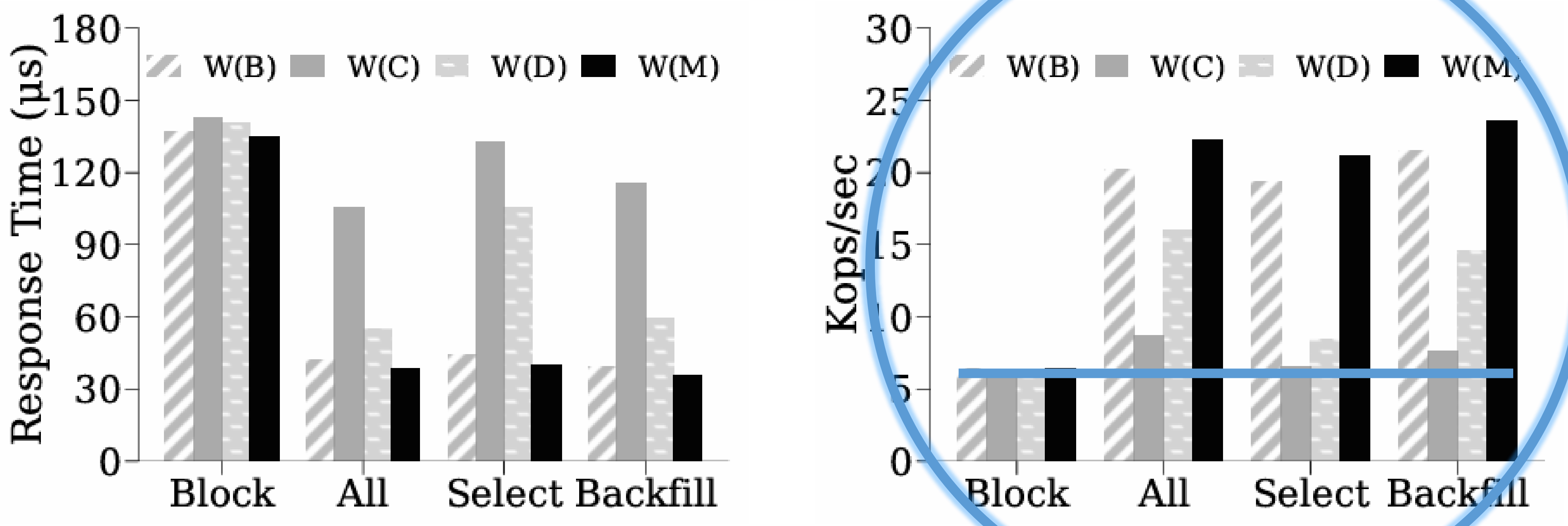
(c) Avg. Resp. Time (d) Avg. Throughput

Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

(2) Fine-Grained Value Packing
Various Workloads (*W(B)* ~ *W(M)*)

- *Block* shows the worst performance regardless of the workload.
- *Selective* performs as poorly as *Block* in large-value-dominant situations (***W(C)***).

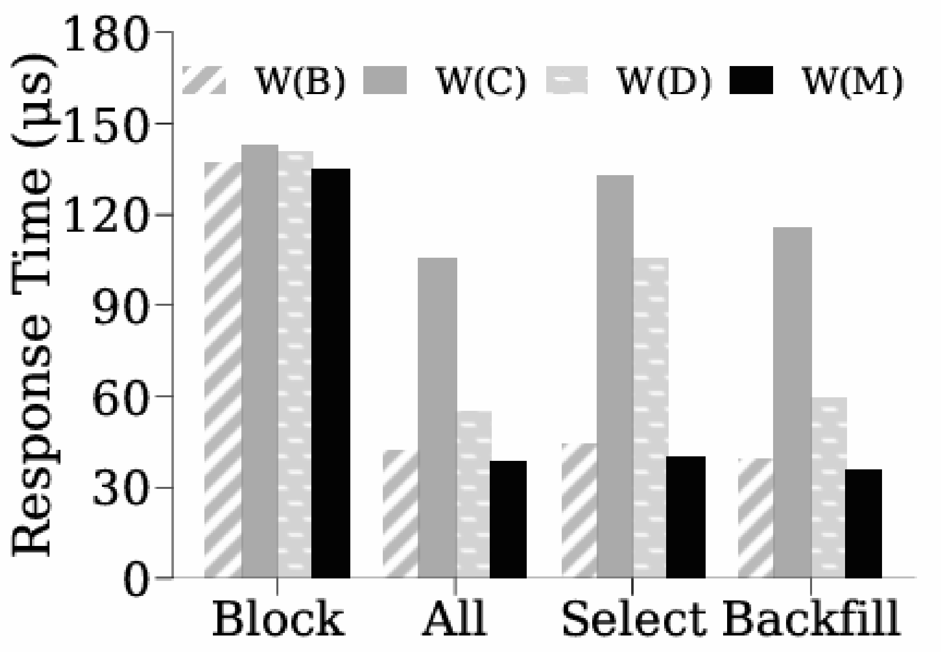


(c) Avg. Resp. Time (d) Avg. Throughput

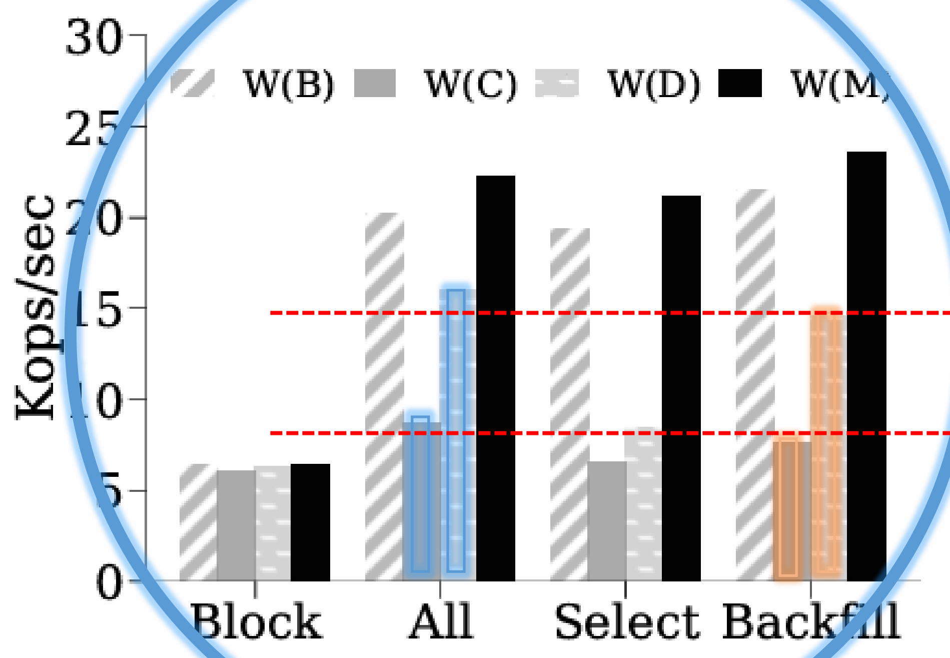
Figure 3. Performance analysis of in-device packing policies.
The host uses the adaptive value transfer method.

(2) Fine-Grained Value Packing
Various Workloads (*W(B)* ~ *W(M)*)

- *Block* shows the worst performance regardless of the workload.
- *Selective* performs as poorly as *Block* in large-value-dominant situations (***W(C)***).



(c) Avg. Resp. Time



(d) Avg. Throughput

Figure 3. Performance analysis of in-device packing policies.
The host uses the adaptive value transfer method.

(2) Fine-Grained Value Packing

Various Workloads (***W(B)*** ~ ***W(M)***)

- However, in scenarios where small values predominate, such as in ***W(B)*** or ***W(M)***, the throughput of the *Selective* dips by at most 4.5% compared to the *All*.
- ***Backfill*** showcases the most optimal performance across both ***W(B)*** and ***W(M)***.

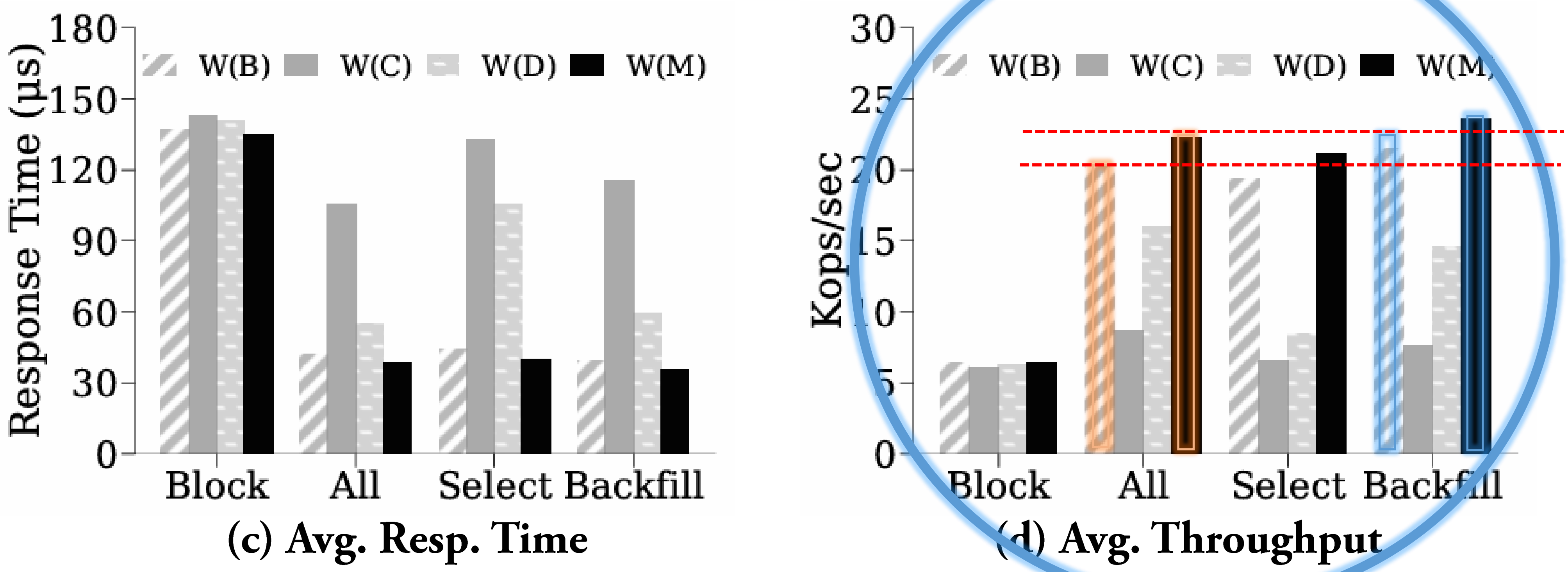


Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

(2) Fine-Grained Value Packing

Various Workloads (*W(B)* ~ *W(M)*)

- However, in scenarios where small values predominate, such as in ***W(B)*** or ***W(M)***, the throughput of the *Selective* dips by at most 4.5% compared to the *All*.
- ***Backfill*** showcases the most optimal performance across both ***W(B)*** and ***W(M)***.

Each packing policy has its own strengths and weaknesses, but the proposed approach performs better under real-world workloads.

(c) Avg. Resp. Time

(d) Avg. Throughput

Figure 3. Performance analysis of in-device packing policies.

The host uses the adaptive value transfer method.

Conclusion

Conclusion

We introduce ***BandSlim*** to address the incompatibilities between traditional block-interfaced storage protocols (e.g., NVMe) and the new key-value interface of KV-SSDs.

The 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, Fine-Grained In-Device Value Packing*.

Thank You

Q&A

Presenter: Junhyeok Park

Contact: *junttang@sogang.ac.kr*