BTS: Exploring Effects of Background Task-Aware Scheduling for Key-Value CSDs

Yeohyeon Park[1], Chang-Gyu Lee[1], Seungjin Lee[1], Inhyuk Park[2], Soonyeal Yang[2]
Woosuk Chung[2], Youngjae Kim[1]
[1]*Dept. of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea*
[2]*Memory System R&D, SK Hynix*
{yeohyeon, changgyu, seungjinn, youkim}@sogang.ac.kr, {inhyuk.park, soonyeal.yang, woosuk.chung}@sk.com

***Abstract*—A computational storage device (CSD) using Intel SPDK guarantees low latency and high throughput. The CSD must aid background tasks for the storage service applications (background tasks) without harming user I/O performance (foreground I/O) since the CSD is also a storage device. However, in practice, SPDK often increases foreground I/O latencies and underutilizes CPU cores in the CSD. These problems proceed from allocating foreground I/Os and background tasks to the same CPU core because SPDK processes them as the same request without distinguishing them. To tackle this, we propose a Background Task-aware Scheduler (BTS) for CSDs built using SPDK. BTS solves the following problems: (i) idle CPU cores in the CSD are not used, and (ii) the latency of foreground I/O increases due to interference with background tasks. For evaluation, we implemented a key-value interface CSD using SPDK. With BTS, the results show that idle CPUs are properly used to process background tasks by guaranteeing the low latency of foreground I/O when the background tasks are set to deduplication.**

***Index Terms*—High Performance I/O, Intel SPDK, Computational Storage Device (CSD)**

I. Introduction

High-Performance Computing (HPC) is starting to carefully look at the potential of Computational Storage Device (CSD) for fast data retrieval and analysis with data generated from simulations. CSDs reduces the transfer of data between host and device by moving computation tasks formerly performed by the host into the storage device, thereby improving overall system performance [1]–[17]. Recently, Los Alamos National Lab. (LANL) and SK Hynix demonstrated the world's first key value CSD (KV-CSD) to accelerate the analysis of HPC scientific applications [3]. Typically, scientific applications in HPC entail analysis of the output data produced after a simulation. In LANL's use case, a portion of the analysis tasks, particularly point and range queries for data retrieval, were carried out by storage devices due to the indexing and the searching capabilities of KV-CSD.

The typical hardware architecture of a CSD embeds an accelerator or processor such as an embedded CPUs in the storage device to perform computations. There are two major approaches to the internal software of a CSD up to date as follows. First, devices such as Samsung's SmartSSD [4], Insider [18], and PolarDB [19] execute computation tasks directly in firmware or FPGA without operating system (OS) support, like bare-metal applications in embedded systems. On the other hand, devices such as Willow [20] and Newport SSD [21] of NGD system, and DragonFire Card [22] have an embedded OS inside. These devices run offloaded computation tasks as a user-mode process on top of the OS. Compared to the bare-metal application approach, the CSD with an embedded OS has advantages in programmability and manageability. For example, offloaded tasks can benefit from OS features such as existing libraries, easier multitasking, and well-defined hardware abstraction via API and OS device drivers. However, incorporating OS in a CSD comes with costs such as user-kernel mode switching, interrupt handling, and context switching overheads.

The aforementioned OS overheads are not only the CSD's problem. Intel SPDK is one of the state-of-the-art projects that solve OS problems. SPDK implements a user-mode NVMe driver that employs a polled-mode that uses polling instead of expensive interrupts to communicate with low-latency SSDs. The SPDK also emphasizes a lock-less and asynchronous design with per-core event loops to minimize communication overhead between two CPU cores, such as locking or cache coherence protocols. These design choices made by Intel – user-mode, polled mode, lock-less NVMe driver, and per-core event loops – also cover similar problems from a high-performance perspective, not computational storage [23]. A CSD may not have the powerful hardware where the SPDK is typically employed, such as ultra-low latency SSDs and tens of enterprise-class CPU cores, but CSDs with an OS can benefit from the SPDK's design effort to minimize the OS overhead. By directly adopting core design principles or SPDK, a CSD with an OS can mitigate the OS overhead.

However, there is still a critical problem in SPDK's design when it comes to executing offloaded computation tasks. SPDK limits the I/O request to be processed by a dedicated CPU core to avoid any overheads induced by communication between CPU cores. More specifically, in SPDK, when a CPU core receives an I/O request and starts to process it, that CPU is responsible for handling it all the way to sending completion to the client. This core binding is critical to the CSD, particularly when the task offloaded has long latency because the task will block all I/O requests and any subroutines derived from I/O pending on that specific CPU. Furthermore, this core binding not only increases overall I/O latency but cannot use other CPU cores even if they are idle.

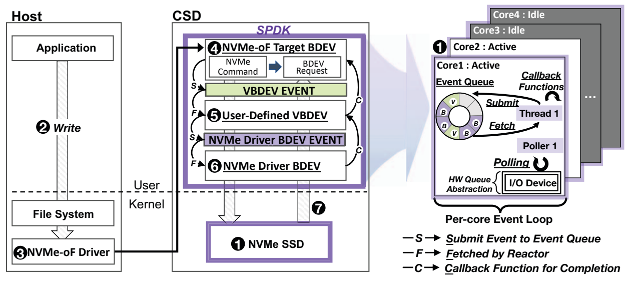


Fig. 1. Depiction of the software stack of a block-based CSD using SPDK and the I/O flow from the host to the CSD.

In this paper, we build a CSD with a key-value interface as the KV-CSD [3] model but use SPDK on top of the CSD with an OS. The computational storage capability is implemented inside SPDK as a background storage service. The background storage service generates background tasks that are not limited to only computation without any extra I/O, but it also includes possible I/O originating from I/O data from the client. To tackle the OS overhead problem occurring in our CSD model, we propose a background task-aware scheduler (BTS). The BTS specifically solves the following problems: (i) SPDK does not distinguish between foreground I/O and background tasks. Therefore it binds the same core for both requests and causes resource contention in that CPU core (execution thread[1]). (ii) SPDK can not use idle resources/CPU core due to the inability to flexibly relocate any requests to idle cores.

To emulate KV-CSD with a BTS, we configured two machines connected to 10 Gbps Ethernet and took one server for the NVMe-oF target using SPDK. The NVMe-oF target is seen as KV-CSD with BTS to another server. Experiments have shown that the BTS minimizes the latency overhead on foreground I/O due to the background tasks and actively uses idle cores.

II. Background and Motivation

A. Computational Storage Device Using Intel SPDK

The SPDK block device layer is BDEV, the C library equivalent to the OS block storage layer. BDEV provides a pluggable module API for implementing block devices that interface with block storage devices. Users can use the available BDEV modules or create VBDEV (virtual BDEV) modules that build block devices on an existing BDEV. Thus, storage developers can easily implement storage service applications such as compression and deduplication using their VBDEV in the SPDK.

Figure 1 depicts the software stack of an SPDK-based CSD. We call the device a CSD because it runs a software stack related to the NVMe driver and SPDK on the device side. As shown in Figure 1, SPDK provides essential functions such as NVMe-oF target and NVMe driver to operate as a device driver as a BDEV module. Users can insert custom storage functions using the VBDEV module. Therefore, they can build high-performance storage applications by configuring the I/O path with VBDEV and BDEV with the necessary functions.

[1]Hereafter, we denote an OS thread bound to a CPU core for processing foreground I/O requests and background tasks as an execution thread.

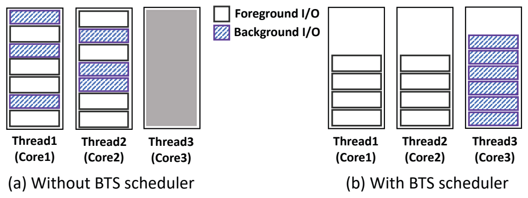


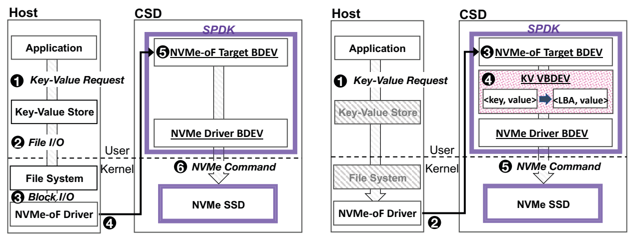
Fig. 2. Illustration of each foreground I/O slowing down due to interference from background tasks.

SPDK abstracts the CPU core into threads, event queues, and pollers. When I/O is delivered to the CPU core, the execution thread handles the I/O. I/O and the BDEV information to be processed are delivered to the event queue by an SPDK event, and a thread processes I/O by fetching an event from the event queue and executing the BDEV function. The poller periodically checks whether I/O to the storage device is completed, and when I/O is completed, it notifies BDEV and the host of I/O completion through a callback.

Figure 1 shows an example of how a write I/O request is processed along the I/O path defined by the user. ❶ The SPDK occupies the storage device, and the user/host application will access the storage device through the SPDK. ❷ Host and CSD are connected through the NVMe-oF protocol to communicate, and the host application sends a write request to the NVMe-oF driver through the file system. ❸ The NVMe-oF driver converts the received write into an NVMe command and sends it to the CSD using the NVMe-oF protocol. ❹ The NVMe-oF target receives the NVMe command via the NVMe-oF protocol, selects one of the cores activated in the SPDK, and delivers it. The execution thread of the core starts I/O processing in units of BDEV and VBDEV. They are inserted into the event queue and then executed in order. To this end, the execution thread executes the NVMe-oF target BDEV and converts the NVMe command into a BDEV request, and the BDEV request is submitted to the event queue along with the VBDEV information to be executed next. ❺ The execution thread executes the VBDEV (if any) defined by the user included in the I/O path along the specified order. Then, when the processing of the last VBDEV is completed, it is responsible to submits an I/O request to retrieve actual data from the device for NVMe BDEV to the event queue. ❻ The NVMe driver BDEV transfers I/O to the NVMe SSD, and then the NVMe SSD serves that I/O. ❼ Since SPDK uses polling, not the interrupt mechanism for communicating with the device, the poller keeps polling the NVMe completion queue for I/O completion. Then, a callback function registered for that I/O is invoked to inform the completion of I/O to (V)BDEVs.

B. Motivation

We found that SPDK places foreground I/O and background tasks derived from the foreground I/O on the same core. So they compete for use of the cores, and there is interference between them, resulting in the following problems:



(a) Key-Value Store atop block-based CSD (b) Key-value CSD

Fig. 3. Depiction of the software stack of key-value CSD.

First, the response time of the user's I/O request (foreground I/O operation) is increased. Figure 2(a) describes this situation well. The background task derived from the foreground I/O is bound to the core on which the foreground I/O runs. Therefore, in the end, the foreground I/O and background task compete for the same CPU core. Figure 2(a) depicts a problem that occurs in a situation where our proposed BTS scheduler is not applied. Assume that the execution time of each task is 1 unit in Figure 2(a). Then, the average response time of the foreground I/O tasks in Figure 2(a) increases by about 31% due to resource contention with the background task.

Second, the background tasks originating from the foreground I/O increase the CPU load of the core that the original I/O started. However, the placement of the coming I/O request occurs without any knowledge about background tasks in SPDK. As a result, the I/O processing of each active core may be overloaded. In other words, idle cores are not used appropriately unless the foreground I/O or background task is distributed evenly over the CPU cores considering their loads.

On the other hand, Figure 2(b) depicts a situation where our proposed BTS scheduler is applied and foreground I/O tasks and background tasks do not compete for cores. In this case, the BTS allocated background tasks to Core3 for their background storage service other than Core1 and Core2, which are used for foreground I/Os. This way, it is possible to prevent an increase in the average response time of 31% for the foreground I/O that occurred in Figure 2(a).

III. Key-Value CSD

A. *Architecture for Key-Value CSD using Intel SPDK*

Figure 3 shows the design of two CSDs using Intel SPDK. Figure 3(a) uses a block-based CSD that exposes the CSD to the host as a block device to run the key-value store using the file system on the host. On the other hand, Figure 3(b) is a key-value CSD that implements indexing corresponding to the storage engine in the key-value store in the CSD, and the host accesses it using the key-value API. Figure 3(b) shows the design of KV-CSD in which the CSD implements the storage engine managing indexes for data inside the device, just like the design of iLSM-SSD [24]. KV-CSD bypasses the host's kernel stack, minimizing I/O software overhead.

In Figure 3(a), the user's key-value request is performed as follows. ❶ An application passes a put or get key-value request to the key-value store. ❷ The key-value store converts the key-value requests into file I/O requests and forwards them to the kernel-space file system. ❸ The file system converts file I/O requests into block I/O requests and passes them to the NVMe-oF driver. ❹ The NVMe-oF driver converts block I/O into NVMe commands and sends them to the CSD using the NVMe-oF protocol. ❺ The CSD uses the SPDK to run the NVMe-oF target and NVMe driver in the user space. The NVMe command the CSD receives is delivered to the NVMe-oF target in the SPDK. Then, the NVMe-oF target passes block I/O to the BDEV, and the BDEV requests the userspace NVMe driver. ❻ The NVMe driver makes block I/O requests to the NVMe SSD. After that, when the NVMe SSD completes block I/O processing, it notifies completion to the application through the same path.

In Figure 3(b), the user's key-value request is performed as follows. ❶ Since KV-CSD implements the key-value store's storage engine (index manager for data) as KV VBDEV using SPDK VBDEV, the host does not require the key-value store or file system on its side. Thus, unlike Figure 3(a), key-value requests are directly passed to KV-CSD. For the KV-CSD's storage engine, we selected a hash-based index for ease of implementation. ❷ The NVMe-oF driver converts the key-value request into a key-value NVMe command and then forwards it to the KV-CSD using the NVMe-oF protocol. ❸ KV-CSD transfers the received key-value NVMe command to KV VBDEV in SPDK. ❹ Note that we implemented a hash-based key-value store engine. KV VBDEV calculates the hash value for the key in the received NVMe command. Then, KV VBDEV searches the hash table with the value to obtain the LBA with the same hash value (for get requests) or assigns a new LBA (for put requests). KV VBDEV converts key-value NVMe commands into block I/O and passes them to NVMe BDEV. The NVMe BDEV requests the received block I/O to the user space NVMe driver. ❺ The NVMe driver delivers block I/O to the NVMe SSD. When the NVMe SSD completes block I/O processing, it notifies completion to the application through the same path.

B. *Key-Value API and NVMe Command Extensions*

To enable user-level applications to make key-value requests to KV-CSD using the NVMe protocol, we implemented a key-value API library using the NVMe I/O passthrough command. The key-value API stores the key in the starting LBA of the NVMe command and stores the addresses of the pages corresponding to the value using the PRP list. Refer to the key-value API of iLSM-SSD [24] for related implementation.

IV. Background Task-Aware Scheduling

A. *Implementation for* BTS *scheduler*

BTS is implemented using SPDK VBDEV and consists of a monitoring module and a core selection module.

Monitoring Module: When the SPDK starts up, all cores are idle. When a client I/O request (foreground I/O) arrives in the SPDK, the SPDK core scheduler selects a core to handle the requested I/O. In SPDK, the CPU core that processes at least one foreground I/O is called an active core. Then, the SPDK core scheduler adjusts the number of active cores

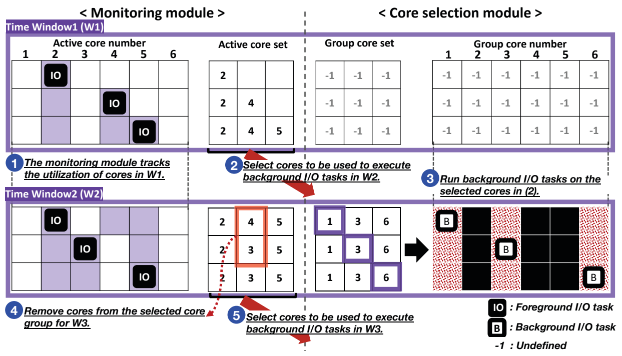

< Monitoring module >
< Core selection module >
Time Window1 (W1)
Active core number
Active core set
Group core set
Group core number
❶ The monitoring module tracks the utilization of cores in W1.
❷ Select cores to be used to execute background I/O tasks in W2.
❸ Run background I/O tasks on the selected cores in (2).
Time Window2 (W2)
❹ Remove cores from the selected core group for W3.
❺ Select cores to be used to execute background I/O tasks in W3.
IO : Foreground I/O task
B : Background I/O task
-1 : Undefined

Fig. 4. Description of the interaction between the monitoring module and the core selection module.

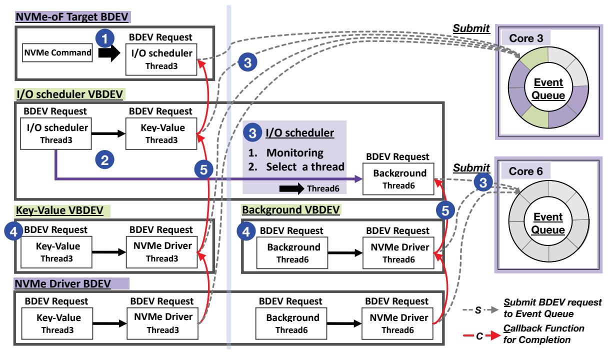

NVMe-oF Target BDEV
NVMe Command
BDEV Request I/O scheduler Thread3
I/O scheduler VBDEV
BDEV Request I/O scheduler Thread3
BDEV Request Key-Value Thread3
❸ I/O scheduler
1. Monitoring
2. Select a thread
Thread6
BDEV Request Background Thread6
Key-Value VBDEV
BDEV Request Key-Value Thread3
BDEV Request NVMe Driver Thread3
Background VBDEV
BDEV Request Background Thread6
BDEV Request NVMe Driver Thread6
NVMe Driver BDEV
BDEV Request Key-Value Thread3
BDEV Request NVMe Driver Thread3
BDEV Request Background Thread6
BDEV Request NVMe Driver Thread6
Submit
Core 3
Event Queue
Submit
Core 6
Event Queue
S : Submit BDEV request to Event Queue
C : Callback Function for Completion

Fig. 5. Description of the implementation of BTS's scheduler and background services and their interaction.

according to the number of foreground I/O requests. The number of active cores changes over time. The monitoring module is responsible for periodically tracking which cores are active. Also, it periodically tracks the utilization of all cores in an array format for every W request (time window). As long as the monitoring module does not update the CPU utilization array too often, the monitoring overhead is not large.

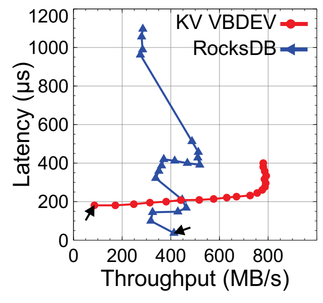
Core Selection Module: The core selection module is responsible for building a group of idle cores based on the CPU utilization array tracked by the monitoring module and selecting cores that process background tasks. For this, the core selection module first selects M idle cores for every W among all N cores in the KV-CSD and constructs an idle core group (G), assuming that the total number of cores in KV-CSD is N, and M is less than or equal to N. Second, the core selection module randomly selects one core from G for the idle core selection request if G is not empty. Otherwise, it sorts the cores according to CPU utilization and selects the core with the lowest utilization. Moreover, background task migration is undesirable if all cores are too busy. It is because the migration overhead may outweigh the performance benefits of the migration. Thus, the BTS can implement an algorithm to determine whether a background task is migrated based on a certain threshold. This threshold can be set using information such as the average and standard deviation of utilization of all cores tracked by the monitoring module. The specific design of the controller for dynamic task migration is left for future work.

Figure 4 describes how the monitoring module of BTS and the core selection module work. Assume that in time window 1 ($W1$), the active cores (processing foreground I/O) are cores 2, 4, and 5, and the non-active cores (idle cores) are cores 1, 3, and 6. Here, an idle core means a core that does not process foreground I/O; thus, core utilization is 0.

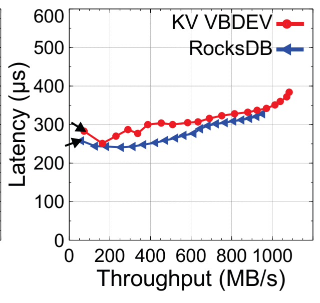
❶ The monitoring module periodically tracks the utilization of cores [1-6]. ❷ The core selection module builds the group G in time window $W1$. Here, cores 1, 3, and 6 are included in group G. And the cores in group G are used as cores to execute background tasks in time window $W2$. ❸ The core selection module runs background tasks on the cores in group G created in time window $W2$. At this time, the core to be used in group G is selected randomly. ❹ Group G is rebuilt in every time window. The utilization rate of each core can change over time. In Figure 4, in time window $W2$, core 3 in group G is removed and core 4 is newly included in group G. Then, the cores of group G built in time window $W2$ become cores 1, 4, and 6. ❺ Cores in group G built in time window $W2$ are used as cores to execute background tasks in time window $W3$.

The BTS scheduler is implemented as I/O scheduler VBDEV and connected between the NVMe-oF target VBDEV and KV VBDEV, which is described in detail in Section III-A. The background service is implemented as a background VBDEV. The foreground I/O executed by the I/O scheduler VBDEV can derive background tasks. The background VBDEV executes background tasks received from I/O scheduler VBDEV. The background VBDEV is connected between the I/O scheduler VBDEV and the NVMe driver VBDEV.

Figure 5 describes the execution process of background tasks in detail. ❶ The NVMe-oF target BDEV extracts information such as opcode, data buffer, key, and data size from the foreground I/O and puts it in a BDEV request. Information about the core currently executing this foreground I/O (core number) is stored in the BDEV request. The core/thread handling the foreground I/O creates an event with the BDEV request and information/function name about the I/O scheduler VBDEV that will process it next, and inserts it into its own event queue. ❷ The I/O scheduler VBDEV derives background tasks while handling foreground I/Os. The core executing the foreground I/O creates a new BDEV request and an event including key-value VBDEV information (function name), and inserts it into its own event queue. ❸ The derived background task is converted to a BDEV request, and then created as an event and inserted into the event queue of the core selected by the I/O scheduler VBDEV for future processing. The core selection algorithm used by the I/O scheduler VBDEV is executed by the core selection module of BTS described in the previous section. ❹ Each core runs KV VBDEV and background VBDEV. ❺ When all processing is completed, the foreground I/O notifies NVMe driver BDEV, KV BDEV, I/O scheduler VBDEV, and client using a callback function. The background task notifies the completion of the



(a) Put workload



(b) Get workload

Fig. 6. Performance evaluation of KV-CSD and RocksDB systems.

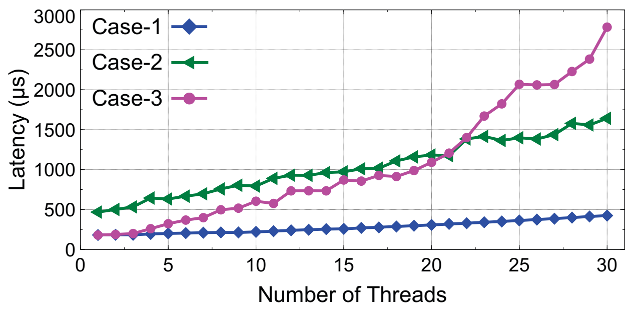


Fig. 7. Performance evaluation of KV-CSD system equipped with BTS scheduler with respect to the increased number of threads in the workload. Each thread runs the dbbench's "Fill sequential" workload.

NVMe driver BDEV, background VBDEV, and I/O scheduler VBDEV.

B. Background Storage Service Application

Offline deduplication was implemented as a representative background storage service application using the background VBDEV described earlier. The I/O scheduler VBDEV sends background tasks to the background VBDEV (Dedup VBDEV) with the key and value's memory address as a BDEV request. Dedup VBDEV splits the data buffer into chunks of a certain size and then calculates a hash value for each chunk. Dedup VBDEV references a deduplication table that stores hash information of each chunk. If no chunks have the same hash value, the chunk's location and hash value are stored, otherwise Dedup VBDEV increments the chunk's reference count to indicate that the chunk is duplicated.

V. Evaluation

A. Experimental Setup

We emulated a KV-CSD using SPDK v.21.1, connected to a server via 10 Gbps Ethernet. KV-CSD communicates with the host through NVMe-oF. The host system and KV-CSD use the same machine with the same specifications (Refer to Table I), but the number of CPU cores used was limited to 10 and 6, respectively. We also implemented hash-based indexer BDEV and offline deduplication VBDEV. Deduplication VBDEV uses a chunk size of 128 bytes and a SHA-1 cryptographic hash algorithm. The monitoring module is set up to track the CPU utilization of all cores for every 60 requests. The cost of updating the CPU utilization array mentioned in Section IV-A is 90 μs.

We used "Fill sequential" and "Read sequential" of the RockDB dbbench benchmark [25] for put (write) and get (read) workloads respectively and with key-value pairs (4B key and 16KB value). This size was set in consideration of the page size of NAND flash. NAND flash based SSD is a block device that can be written and read in units of pages (eg, 16KB). Reading and writing of key-value pairs smaller than this size causes I/O amplification. Therefore, we assume that the host can read/write key-value pairs in a page unit after buffering small key-value pairs.

TABLE I
Hardware/software specifications for host and CSD

CPU	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz (Host machine: 10 cores, CSD: six cores)
Memory	32GB DDR4
Disk	500 GB Samsung 970 EVO SSD
CSD interface	NVMe-oF (10 Gbps Ethernet)
Software	Ubuntu 20.04, SPDK v.21.10, RocksDB v.6.23

B. Performance Evaluation of KV-CSD

We evaluated how much KV-CSD reduces the host's I/O stack overhead through kernel bypass. For this, we compared the following two systems.

- **RocksDB:** This is a key-value store that runs RocksDB with 32MB buffer on the host's file system.
- **KV VBDEV:** This is a system using KV-CSD equipped with a hash structure-based storage engine.

Figure 6(a)&(b) shows the change in latency and throughput as the amount of I/O requested increases. Figure 6(a) is a performance comparison for the put workload. For more than six threads, KV VBDEV outperforms RocksDB. KV-CSD minimizes the I/O stack overhead by bypassing the kernel of the host. However, under six threads, RocksDB shows better performance than KV VBDEV, which is presumably due to the performance gain from RocksDB's internal 32MB buffer. Figure 6(b) is a performance comparison for the get workload. The two systems show almost similar performance. However, RocksDB has slightly lower latency and higher throughput for all cases, which is presumably due to the effect of the host's OS cache.

C. Performance Evaluation of the BTS Scheduler

We compared performance for the following three cases with respect to the increased number of threads issuing I/O in the benchmark. We experimented with put-only workloads and considered offline deduplication described in Section IV-B as a background task. Figure 7 shows the results of experimenting with a put workload to account for a situation where background tasks require some computation time.

- **Case 1:** This implementation is a case in which only foreground I/O is processed without a background task service.

- **Case 2:** The background task is handled with foreground I/O, but without the BTS scheduler.
- **Case 3:** The background task is handled with foreground I/O and the BTS scheduler is used.

In all cases, the latency increases as the foreground I/O requests increase. Case 1 has the lowest latency, while Case 2 and Case 3 have high latency. Case 1 and Case 2 increase linearly, whereas Case 3 increases exponentially after 21 I/O threads. In Case 3, when the number of host threads is three or less, background task is placed in an idle thread that does not process foreground I/O, so it actively uses idle threads and does not interfere with foreground I/O processing. Therefore, it shows a similar delay time to Case 1. Case 3 shows lower latency than Case 2 when the number of I/O threads is less than 21 (normal situation), whereas Case 3 shows higher latency than Case 2 when the number of host threads is greater than 21 (overloaded situation). In the normal situation, BTS actively uses the idle cores to take advantage of the performance advantage, whereas in the overloaded situation, the overhead of the BTS scheduler outweighs the performance advantage and shows no performance gain. Therefore, the BTS scheduler can set the threshold described in Section IV-A 21 (threads) and eliminate performance loss due to unnecessary task migration.

VI. Concluding Remarks

In this paper, we proposed a background task-aware scheduler (BTS) that can actively use idle cores for processing background tasks to minimize the slowdown of foreground I/O in the SPDK-based KV-CSD. Extensive evaluation has shown that BTS enables the active use of idle cores and minimizes the increase in response time of foreground I/O when the background task is executed together. In addition, although this paper investigated the effect of BTS in KV-CSD, we believe that BTS can be applied to any storage server environment. In particular, in a storage server environment with multiple high-performance CPU cores, the performance improvement effect of BTS is expected to be significant. Therefore, we will further verify the effectiveness of BTS for various hardware settings of storage server and KV-CSD in a future study.

Acknowledgments

We thank the reviewers and our shepherd, Qing Zheng, for their constructive comments that have significantly improved the paper. This work was supported in part by SK hynix research grant and by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. NRF-2021R1A2C2014386) and by Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT)(No.2021-0-00528, Development of Hardware-centric Trusted Computing Base and Standard Protocol for Distributed Secure Data Box). Y. Kim is the corresponding author.

References

[1] Scaleflux Inc, "Scaleflux." [Online]. Available: http://www.scaleflux.com/

[2] Eideticom, "Noload computational storage processor," https://www.eideticom.com/media/attachments/2020/06/03/noload-compression-zfs.pdf, 202020.

[3] SK hynix and Los Alamos National Laboratory, "Los Alamos National Laboratory and SK hynix to demonstrate first-of-a-kind ordered Key-value Store Computational Storage Device," https://discover.lanl.gov/news/0728-storage-device, 2022.

[4] Samsung Electronics, "Samsung Electronics Develops Second-Generation SmartSSD Computational Storage Drive With Upgraded Processing Functionality," Jul 2022. [Online]. Available: https://news.samsung.com/global/

[5] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves, "Cost-Effective, Energy-Efficient, and Scalable Storage Computing for Large-Scale AI Applications," *ACM Trans. Storage*, vol. 16, no. 4, Oct 2020.

[6] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou, "In-storage Processing of I/O Intensive Applications on Computational Storage Drives," 2021. [Online]. Available: https://arxiv.org/abs/2112.12415

[7] A. HeydariGorji, S. Rezaei, M. Torabzadehkashi, H. Bobarshad, V. Alves, and P. H. Chou, "HyperTune: Dynamic Hyperparameter Tuning for Efficient Distribution of DNN Training over Heterogeneous Systems," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20, 2020.

[8] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, ser. FAST '13, Feb. 2013, pp. 119–132.

[9] C. Lukken and A. Trivedi, "Past, Present and Future of Computational Storage: A Survey," *CoRR*, vol. abs/2112.09691, 2021.

[10] A. Barbalace and J. Do, "Computational storage: Where are we today?" in *CIDR*, 2021.

[11] A. Barbalace, M. Decky, J. Picorel, and P. Bhatotia, "BlockNDP: Block-Storage Near Data Processing," in *Proceedings of the 21st International Middleware Conference Industrial Track*, ser. Middleware '20, 2020, p. 8–15.

[12] C. Lukken, G. Frascaria, and A. Trivedi, "ZCSD: a Computational Storage Device over Zoned Namespaces (ZNS) SSDs," *arXiv preprint arXiv:2112.00142*, 2021.

[13] G. Frascaria, "e2bpf: an evaluation of in-kernel data processing with ebpf," Ph.D. dissertation, Universiteit van Amsterdam, 2021.

[14] M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, H. Bobarshad, V. Alves, and N. Bagherzadeh, "Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics," in *Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP '19, 2019, pp. 430–437.

[15] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 153–165.

[16] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, "NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21, New York, NY, USA, 2021, p. 262–272.

[17] S. Salamat, H. Zhang, Y. S. Ki, and T. Rosing, "NASCENT2: Generic Near-Storage Sort Accelerator for Data Analytics on SmartSSD," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 2, Jan 2022.

[18] Z. Ruan, T. He, and J. Cong, "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive," in *In Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '19, Jul. 2019, pp. 379–394.

[19] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang, "POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '14, 2014, p. 29–41.

[20] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, "Willow: A User-Programmable SSD," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '14, 2014, p. 67–80.

[21] NGD Systems, "Newport CSD." [Online]. Available: https://www.ngdsystems.com/

[22] Argonboards, "LS2088A Intelligent-SSD Card," Aug 2022. [Online]. Available: https://www.argonboards.com/ls2088a-intelligent-ssd-card

[23] Intel. SPDK. https://spdk.io/.

[24] C.-G. Lee, H. Kang, D. Park, S. Park, Y. Kim, J. Noh, W. Chung, and K. Park, "iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics," in *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '19, 2019, pp. 384–395.

[25] Facebook, "RocksDB." [Online]. Available: http://rocksdb.org