Fine-grained Control of Concurrency within KV-SSDs

Janki Bhimani*, Jingpei Yang†, Ningfang Mi§, Changho Choi†, Manoj Saha*, and Adnan Maruf*

*Knight Foundation School of Computing and Information Sciences, Florida International University

† Memory Solution Research Lab, Samsung Semiconductor

§ Dept. of Electrical & Computer Engineering, Northeastern University

ABSTRACT

The development of KV-SSDs allows simplifying the I/O stack compared to the traditional block-based SSDs. We propose a novel Key-Value-based Storage infrastructure for Parallel Computing(KV-SiPC)-a framework for multi-thread OpenMP applications to use NVMe-based KV-SSDs. We design a new capability to execute workloads with multiple parallel data threads along with traditional parallel compute threads, that allow us to improve the overall throughput of applications, utilizing the maximum possible storage bandwidth. We implement our KV-SiPC infrastructure in a real system by extending various processing layers (e.g., program, OS, and device layers) and evaluate the performance of KV-SiPC by using block-based NVMe SSDs in the traditional I/O stack as a baseline for comparisons. The experimental results show that KV-SiPC can better utilize the available device bandwidth and significantly increases application I/O throughput.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; • **Hardware** → *External storage.*

KEYWORDS

Solid State Drive (SSD), Key-Value SSD, Multi-threading

ACM Reference Format: Janki Bhimani, Jingpei Yang, Ningfang Mi, Changho Choi, Manoj Saha, Adnan Maruf. 2021. Fine-grained Control of Concurrency within KV-SSDs. *In The 14th ACM International Systems and Storage Conference (SYSTOR '21)*, June 14–16, 2021, Haifa, 12 pages. https://doi.org/10.1145/3456727.3463777

[1]This work was initiated during Janki Bhimani's internship at Samsung Semiconductor Inc. [9]. This work was partially supported by National Science Foundation Awards CNS-2008324, CNS-2008072, and Career Award CNS-1452751.

SYSTOR '21, June 14–16, 2021, Haifa, Israel

ACM ISBN 978-1-4503-8398-1/21/06…$15.00

https://doi.org/10.1145/3456727.3463777

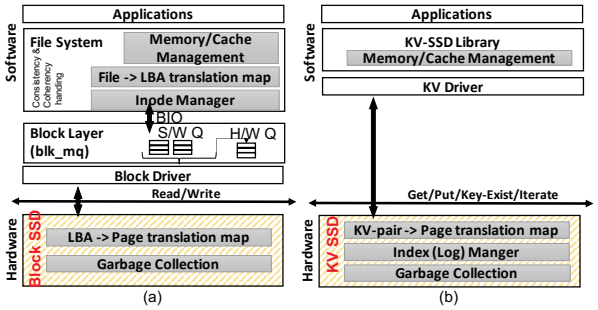


Figure 1: I/O Stack layers for (a) Block SSD, and (b) KV-SSD.

1 INTRODUCTION

The intermediate layers of the operating system (OS) increase the depth of the I/O stack. Data conversions (such as from files to blocks) through these intermediate layers and the corresponding spin-lock management further consume a lot of processing time. It becomes even worse when using multi-thread applications. The single software queue in block I/O (BIO) serializes these data accesses of all threads. Even for the kernels equipped with the multi-queue block layer (blk-mq) [11] that maintains per-CPU software queues, the parallel data accesses issued by multiple threads of each core are serialized. Although the blk-mq can reduce the number of I/O operations by merging multiple requests with adjacent logical block addresses (LBAs) from different software queues to a single hardware issue queue, it still consumes many CPU cycles to search through the software queues for merging. Our in-depth analysis reveals that the high-performance storage bandwidth of NVMe SSDs is underutilized by more than 46% due to such possible I/O processing overhead.

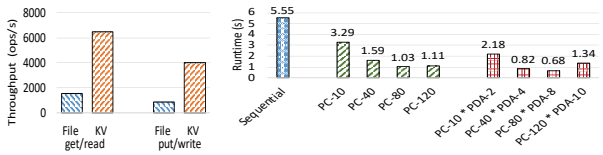
Recently, a new storage technology, called Key-Value SSD (KV-SSD) [19, 21, 39], has been developed to preserve the nature of application data (e.g., variable value sizes) and not restrict accessing SSDs with only a fixed block size. Figure 1 shows the comparison of software and hardware layers for block-based SSDs (see Figure 1 (a)) and KV-SSDs (see Figure 1 (b)). The block mapping becomes unnecessary when the underlying storage device is a KV-SSD [21]. This new technology exposes the key-value oriented interface directly to applications. The KV-SSD firmware at the hardware layer can manage read/write/update of key-value pairs of data. This reduces the I/O processing overhead by saving the CPU cycles used for converting data. Thus, KV-SSD has shown the ability to accelerate key-value stores and key-value based applications by streamlining the I/O stack [10, 20, 25].

In this work, we explore a new research question that *"Can KV-SSDs improve performance of multi-threaded file-based applications, too?"* The state-of-the-art shows that current KV-SSDs cannot support running parallel multi-threaded file-based applications due to the following limitations. First, the current KV-SSD only supports key-value based operations, e.g., store, retrieve, and delete [42]. These operations can be easily used by applications whose data is natively in key-value format, e.g., RocksDB [29], WiredTiger [13]. The other general-purpose applications, such as most applications that are not key-value, cannot directly use KV-SSDs. Second, for most multi-threaded file-based applications, critical operations such as shared memory buffer management are traditionally taken care of by the filesystem. However, the I/O stack of KV-SSD does not include the filesystem layer. Thus, an alternative method is needed to manage the data of these parallel applications. Third, the current KV-SSD provides asynchronous operations in order to obtain better performance with less system resource usage. Whereas, multi-threaded file-based applications are not designed to directly manage asynchronous I/Os because the intermediate filesystem and block layers are in charge of thread safety with asynchronous I/Os on conventional block-based NVMe SSDs. To address these limitations, we present a new method - Key-Value based Storage infrastructure for Parallel Computing (named *KV-SiPC*) - to use NVMe Key-Value SSDs for accelerating multi-threaded OpenMP applications.

The major contributions of our work are as follows.

- We provide a new capability to execute workloads with parallel data threads while using KV-SSDs along with traditional parallel compute threads.
- We develop a new technique to allow fine-grained control and configuration of the optimal number of parallel data and compute threads, which results in better resource utilization.
- We build a key-value concurrency manager to maintain memory mapping and thread-safety when running OpenMP applications using KV-SSDs.
- We design and construct different processing modules (such as sequential, parallel compute, and parallel data access along with parallel compute) to investigate the impacts of parallel compute and data threads.
- We implement our *KV-SiPC* infrastructure in a real system and evaluate the performance by using Samsung PM983 KV-SSDs with different firmwares to support both traditional block and key-value interfaces for fair comparison.

Our evaluation results show that *KV-SiPC* can allow multi-thread applications to experience a significant reduction in end-to-end latency using KV-SSDs, compared to the traditional block-based SSDs.



(a) (b)

Figure 2: (a) Throughput of Get/Read, and Put/Write, async. 28KB size file and key-value handling by block and KV-SSD, (b) Speed-up using a multiple parallel compute (PC) threads and parallel data access (PDA) threads.

2 MOTIVATION

Even in the era of big data, most things in many file systems are small [2, 3]. Inevitably, scalable systems should expect the numbers of small files to exceed billions. Application-level KV stores handle many small objects efficiently and have resulted in better performance and scalability [4, 14, 15]. These KV stores benefit largely from KV-SSDs [10, 20, 25]. However, it is unclear that if the thin OS I/O stack of KV-SSDs can also accelerate file-based workloads especially with small objects.

[Throughput using Block SSD and KV-SSD.] As many applications are file-based, exploring if KV-SSDs can be used for them is important. One straightforward way of converting file *read/write* on a block device to key-value pair *get/put* on a KV-SSD is to save the content of the file as "value" and the unique path to the file as its "key." We use the same SSD hardware with block firmware and key-value firmware to compare throughput while reading and writing files/KV-pairs of the same sizes. We program put/write and get/read operations using FIO bypassing the page cache. Figure 2(a) compares the I/O throughput while performing get/put of KV-pairs on KV-SSD and read/write of files on block-SSD. We perform ten thousand random asynchronous I/Os, each of either KV-pair or file size 28KB. We observe that using lightweight KV stack on KV-SSD, we obtain better throughput than the traditional I/O stack on block SSD. With KV stack, we can saturate 8% to 50% of the best obtainable device read and write bandwidth for Samsung NVMe PM983 U.2 SSD [40]. While reading and writing files using traditional I/O stack on the same model of SSD can only saturate 2% - 12% of the device bandwidth. This is because that KV-SSDs have a thin software stack in the I/O path, which eliminates the overhead of file open-close and conversion at multiple software translation layers. We hypothesize that not only key-value based applications but also file-based applications can be accelerated using KV-SSDs.

[Impact of Parallelism.] Next, we try to understand the impact of simultaneously using parallel data threads (PDA) to utilize internal parallelism of SSD channels along with the parallel compute threads (PC) that improves the utilization of system CPUs. We use block SSD for this study as we are

not aware of any existing system kernel and device drive that allows us to control I/O parallelism on KV-SSDs. We run a matrix multiplication application with a dataset of 1 million floats over the XFS file system on block SSD. From Figure 2(b), we observe that upon using multiple data access threads launched by each compute thread, the runtime can further be reduced. This also results in obtaining better device bandwidth saturation. However, with further in-depth exploration, we observe that while using the traditional I/O path with NVMe block SSD, each core use only one submission and one completion queue. Although, NVMe interface supports 64K queues, the block device driver restricts using more than one set of queues per core. Thus, the host writes a fixed-sized circular buffer space of a submission queue, either at the host or drive memory, and triggers the doorbell register when commands are ready to execute by NVMe SSD. The SSD controller then picks up the queue entries in the order received. The completion queues post the status for completed commands. Thus, although using alternative approaches such as Storage Performance Development Kit (SPDK), the software stack can be reduced. However, the restrictions imposed by the block device driver may still yield sub-optimal performance. To take advantage of the thin software stack in the I/O path of KV-SSDs and no block layer queues, we design a new device driver that can use multiple submission and completion queues per core. This will allow us to use parallel threads in the same core to leverage parallel data accesses. Suppose the number of queues is a function of the complexity of the application workload and the number of cores in a system. In that case, we may reduce the end-to-end latency of multi-thread applications and better utilize the massive available bandwidth of modern NVMe SSDs. This motivates us to design I/O stack that allows applications to control compute and data parallelism using KV-SSDs even for non key-value based applications.

3 FRAMEWORK

3.1 *KV-SiPC* Architecture

The technical design of *KV-SiPC* includes the *compute path* - a set of functional units that carry out parallel computing operations, and the *data path* - a set of functional units that carry out data processing operations. Specifically, *KV-SiPC* is a prototype of key-value based parallel computing that is deployed using Samsung's KV-SSD [39] and multi-thread OpenMP library [12]. *KV-SiPC* uses OpenMP's manager-worker paradigm. The main thread is the manager. Initially, the manager launches different worker threads. All worker threads run in parallel to execute the same code on different parts of data. When all worker threads finish, the manager further processes results from these worker threads. Depending on the implementation of the algorithm, the manager decides to fork multiple workers again or terminate. All threads (manager or workers) have their own thread context, thread ID, stack, stack pointer, program counter, condition codes, and general-purpose registers. The OpenMP specification consists of APIs, a set of pragmas, and several settings for OpenMP-specific environment variables. One can mark some parts of the code to be executed in parallel by using special #pragma directives. Then, multiple threads will be launched to run the marked code. Each thread is the abstraction of control flow during the execution of a program.

OpenMP applications are originally file-based multi-thread applications designed to run on a block device that does not need to control data threads. The traditional OpenMP launches parallel compute threads, while the intermediate block layer controls data access. KV-SSDs do not have a block layer. Thus, we can directly expose the control of parallel data threads to the user or the program layer. This will allow us to have fine-grained control over resource management of resources such as CPU, memory, and SSD bandwidth and improve overall performance. We need to have solutions to manage parallel compute and data threads and perform thread-safe asynchronous key-value I/O operations as well. Thus, in *KV-SiPC*, we design two new components: (1) capabilities to execute workloads with multiple parallel data threads along with traditional parallel compute threads, that allows us to improve overall throughput of applications, utilizing the maximum possible storage bandwidth. (2) key-value concurrency manager to integrate OpenMP pragmas with the key-value Linux kernel device driver (KDD) to maintain memory mapping and thread-safety when running a multi-threaded application on KV-SSDs. The proposed idea to have parallel data threads to exploit the high internal concurrency of KV-SSDs is not specific to just OpenMP applications instead generally applies to even other parallel POSIX thread libraries. However, the real system evaluation of this idea required considerable initial efforts to integrate library functions with key-value KDD. Thus in this paper, we mainly focus on OpenMP. In the remaining section, we discuss the functions of each component in detail.

3.2 Parallel Compute (PC) and Parallel Data Access (PDA) Threads

During the execution of a general multi-thread program, data accesses issued by all parallel compute threads are served by a single thread. This is because the current I/O stack assumes that writing a large buffer with one thread is as fast as writing many small buffers with multiple concurrent threads [24, 35]. However, this is only true for storage devices with low bandwidth, but not for high bandwidth devices such as KV-SSDs that are capable of performing parallel I/Os (see Figure 2(b)). To take full advantage of the internal parallelism of KV-SSDs, we develop an infrastructure that support users to launch and control multiple parallel data

threads for each parallel compute thread. First, we modify some OpenMP library functions to control the launch of parallel compute and data threads. Second, we design a new controller that helps KV-SiPC decide the best number of compute and data threads by monitoring the online usage of compute and memory resources. Our modified OpenMP library and key-value KDD represent metadata as key-value pairs indexed into a hash table. Persisting metadata uses the same submission and completion queues as data. Most metadata management operations are inspired from TableFS [37]. For example, creating a hard link modifies the value with an attribute prefix indicating the value entry is a hard link to the object. Whenever KV-SiPC creates the second hard link to an object, it creates a separate hash table entry for the object itself, with a null value and its new key as the rehashed value of its parent's key. Some other metadata operations such as the object rename operations need to be handled by the application library because the device driver only maintains a unique key to locate the data at any particular location. We modify the file to the key mapping table entry while renaming a file.

[Modified OpenMP Library Functions.] In our infrastructure, #pragma (a language construct directive that specifies how a compiler processes its input) is used to support the launch of parallel tasks. To ensure end-to-end lock-free execution of parallel threads, we integrate OpenMP pragma to the functions of the Samsung key-value API [19] and the key-value related functions of the KV-SSD's device driver such as get(), put(), and delete() to take care of launching parallel data threads without any specific changes required by the programmer to application code. Users need to just specify the processing module (such as PDA_PC) in the execution command. We discuss processing modules in detail in Section 3.3. The segments specified by pragma directives of application code are exposed to both parallel compute and parallel data access at runtime, depending upon the desired processing module.

The launch of parallel compute and data threads from the program layer is controlled in our compute path. We modify the functionality of omp_set_num_threads in the OpenMP library. Particularly, KV-SiPC decides the number of compute and data threads by monitoring the online usage of compute and memory resources. Rather than fixing <thread_no> to <omp_get_max_thread()> throughout the execution, we initialize <thread_no> to 2. Then, during the execution, omp_set_num_threads is set to the optimal <thread_no>. The optimal value of the number of compute and data threads is managed by our new controller that we explain next.

[Number of PC and PDA Threads.] Apparently, increasing the number of threads can improve performance. However, saturation will eventually happen because more threads consume more CPU and memory resources. Therefore, the next question that KV-SiPC has to address is *"how to dynamically identify an optimal number of PC threads and PDA threads?"* We design two methods "KV-ins" and "KV-mod" to get the answer. First, for a set of less compute-intensive applications, the straightforward method KV-ins is to instrument system resources and accordingly adjust the number of threads at runtime. Specifically, KV-SiPC periodically instruments the runtime CPU and memory utilization of each processor by using mpstat [16] and memstat [5], respectively. KV-SiPC then launches new compute threads for each application until either CPU utilization exceeds the desired threshold (e.g., 85%) or the maximum number of parallel threads supported by a CPU is reached. To decide the optimal number of parallel data access threads launched by each parallel compute thread, it further monitors the runtime I/O throughput and memory utilization. It uniformly increases the number of parallel data threads launched by each compute thread of all the user-specified applications until overall I/O throughput does not decrease, and memory utilization does not exceed the desired limit, such as 85%.

The second method KV-mod is for compute-intensive applications. We adopt the *M/M/c* queuing theory [43] to solve the problem as follows. We assume that each PC thread launches the same number of PDA threads. Let the cumulative arrival rate from all the applications be λ. For example, if two applications are running in parallel and every millisecond they submit four independent compute operations, then λ = 4k operations/second. Let $\vec{\mu}$ denote the average service rate of the bottleneck resource. We use roofline model [33] to predict the bottleneck. For example, if the application is I/O bound, the average service rate is 332MB/s for an SSD that reports 83K IOPS for 4KB random I/Os in its data-sheet. Impact of the number of threads (i.e., c) is modeled by parallel servers in the queuing model. $c \in [1, c_{max}]$, c is the number of PC*PDA threads, and c_{max} is the maximum number of supported parallel threads by the system. The resultant of the changes in overall service rate from queuing model, for a range of c is captured in the vector $\vec{\mu_c}$. The objective is to find the best value of c. Once we have λ and $\vec{\mu_c}$, KV-SiPC calls the *optimizeServNum* function to decide the best value of c, which computes $\underset{c \in [1, c_{max}]}{argmin} \; [\frac{ErlangC(\lambda, \vec{\mu_c})}{c\vec{\mu_c} - \lambda} + \frac{1}{\vec{\mu_c}}]$. Different combinations of c values and corresponding estimated service rates $\vec{\mu_c}$ are tested to explore the optimal c that would minimize the total end-to-end latency. This procedure also uses the *ErlangC* function [7] to calculate the probability that an arriving job will need to queue (as opposed to immediately being served). Finally, the best value of c is periodically updated while running applications. Note that one of the above techniques can be selected to run the application. Currently,

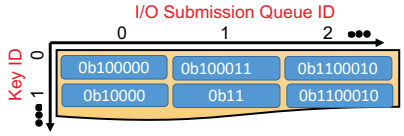


Figure 3: Illustration of bitmap.

our infrastructure does not support automatically changing from one to other at runtime.

[Key-Value Concurrency Manager.] While running on a block device, a multi-thread application usually relies upon the filesystem for memory mapping of application data and upon the block layer for its thread-safe asynchronous I/Os to the device. As there is no filesystem and block layer when KV-SSD is used, we need to explicitly maintain memory mapping and thread-safety for running a multi-threaded application on KV-SSDs. Otherwise, parallel asynchronous I/Os can lead to execution errors or incorrect data access. In the data path of our infrastructure, we develop a new key-value concurrency manager consisting of I/O completion queues and I/O submission queues that are managed by users on the host side for appropriate memory mapping of the data. The two main functions, i.e., "kvs_store" and "kvs_retrieve," are responsible for managing data flow from host memory to SSD and vice verse. "kvs_store" appends the key-value pair to the I/O submission queue of the corresponding thread for write/update to KV-SSD. "kvs_retrieve" finds and returns the key-value pair to the specified memory buffer. We use opcode to differentiate various store and retrieve actions. An opcode identifies which basic computer operation in the instruction set is to be performed.

The memory buffer allocation, which is traditionally taken care of by the filesystem layer while using block SSD, is now explicitly handled by our modified infrastructure at the user layer. For the allocation of single data value, we use a thread management command to fetch data from KV-SSD according to its corresponding key ID. The memory buffer allocation for the stream of data is done by considering queue depth and the number of parallel data threads for each I/O queue. For example, for structure `kv_data` which contains key_buf and val_buf, the following command reserves the required memory `cmd_data = valloc(sizeof(struct kv_data) * qdepth * iothread)`. In this module, we also construct and maintain appropriate counters to track the number of I/Os issued, number of I/Os on the air, and number of I/Os completed by each data thread and I/O queue to ensure thread-safety of the asynchronous I/O operations. These counters and the corresponding I/O submission and completion queues are updated if there are any completed asynchronous I/Os.

During runtime, the number of completed I/Os by each thread is instrumented by calling the function "KV Get IO-event" within the same thread that issued I/O. Specifically, a bitmap is maintained to log the mapping of the key to OpenMP thread-id and corresponding I/O submission queues for all data. A sample bitmap is shown in Figure 3, which is a two-dimensional map maintaining seven-bit thread-id for each I/O submission queue-id and key-id. This bitmap is updated when a batch of I/O requests enter the I/O submission queues. The counters maintaining the vacancy of those I/O submission queues are decreased. Upon polling the I/O completion queues, the "kvs_retrieve" function uses the bitmap to identify which I/O submission queue and thread-id the completed I/O requests belong to and then increases the vacancy of the corresponding I/O completion queue. New I/O requests can enter respective I/O submission and completion queues depending on their vacancies. The maximum number of I/Os that can be submitted to an I/O submission queue at the same time is equal to the queue depth times the number of parallel threads. An important issue in managing multiple parallel compute and data threads on the same KV-SSDs is to maintain the coherence of the shared data. For example, read-modify-write operations on the shared data by threads using different cores may result in data inconsistency. We maintain all the data into one of the five different states of the MOESI protocol [6] and implement transactions using the rules of its state transition diagram.

In our infrastructure, we directly expose NVMe driver level commands (such as *<NVME_IOCTL_GET_AIOEVENT>*) to our concurrency manager. We implement and integrate `KV-SiPC` with SNIA API [42] to initialize, open, close, perform, and verify read and write operations on KV-SSDs. These API functions substitute the functionalities of the regular POSIX API functions (such as readdir(), opendir(), and closedir()) for KV-SSDs. Thus, the key-value concurrency manager keeps track of the simultaneous input-output operations performed on NVMe KV-SSD to guarantee thread safety for executing parallel applications.

3.3 Implementation Discussion

[Processing Modules.] In order to effectively evaluate our proposed infrastructure, we implement three different processing modules in terms of compute and data access parallelism: (1) Seq - sequential compute with sequential data access, (2) PC - parallel compute with sequential data access, and (3) PDA_PC - parallel compute with parallel data access. To achieve fair comparisons, we rewrite applications to support parallelism in compute and data for both block I/O and KV access. The "Sequential (Seq)" implementation represents a straightforward application that runs with one compute thread and one data thread, while the "Parallel Compute (PC)" implementation represents traditional multi-thread applications that use multiple compute threads to compute but access data in a sequential channel. The third processing

module (i.e., PDA_PC) aims to capture a more complex parallel scenario, where multiple parallel compute threads are run, and each compute thread can launch multiple parallel data access threads. In order to enable this PDA_PC module for traditional block-based SSD, we provide handles to grant root privilege to each application thread and use private channels [46] to implement multiple I/O submission queues. In this way, we support parallel data threads at the host layer even for block-based SSD. No changes need to be made on the device driver of block-based SSD.

A parallel operating thread is an abstraction of control flow during execution. Each thread has its context and meta information for running. For the two parallel processing modules (i.e., PC and PDA_PC), it is crucial to maintain thread safety throughout the execution of an application to avoid data race and protect shared variables from undesirable mutations. However, the existing key-value storage techniques [18, 23] were designed and used for traditional key-value applications without parallel transactions. Thus, the existing key-value interface (such as wrappers) and APIs cannot handle parallel asynchronous I/Os, especially in multi-threaded environments. To address this issue, we resolve thread-safety at the layers of application, host, and device to guarantee the correctness of end-to-end operations. For example, at the application layer, the commonly used containers of the "Std C++11" library are not thread-safe. Thus, we only consider the thread-safe versions of containers (e.g., $< queue >$, $< stack >$, and $< vectors >$) and further develop new thread-safe versions of string processing functions such as "strtok()." At the host layer, we use conditional variables such as $< pthread\_mutex\_lock >$, $< pthread\_cond\_wait >$, and $< pthread\_mutex\_unlock >$ to maintain memory allocation for each thread and avoid concurrent race condition as well as memory fragmentation. Finally, at the device driver layer, to avoid concurrent data race, we manage NVMe driver layer commands (such as *<NVME_IOCTL_GET_AIOEVENT>*, *<NVME_IOCTL_SET_AIOCTX>*) separately for each I/O submission and completion queues and each thread-id. Note that in our current implementation, PC and PDA threads are forked when the application starts. Changing the number of threads dynamically during the execution of the application raises many new challenges (e.g., data ownership assignments), and we plan to work on this in the future.

[Functionalities of kvs_map.] The "kvs_map" function maintains the "application data <-> keys" mapping table implemented using the hash table data structure. During application writes/updates, "kvs_create" first generates key-value pairs. Then "kvs_map" updates the application data to the key mapping table. While during application reads, this mapping table is used to identify the keys dedicated to the required application data. "kvs_map" also computes checksum to consider the application level cache effects for data

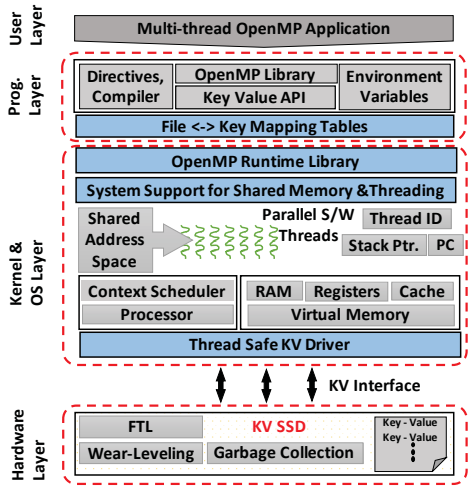


Figure 4: System Architecture.

using cyclic redundancy checks (CRCs) to protect against readback errors. Thus, any multi-threaded OpenMP application can be translated to a key-value based application by mainly replacing file calls of `<File Initialization>` with `<kvs_create>`, `<File Open>` with `<kvs_map>`, and `<File Scan>` with `<kvs_retrieve>`.

4 EVALUATION

In this section, we present the implementation details of *KV-SiPC* and our evaluation results. We mainly analyze various performance parameters comparing block SSD and KV-SSD and study resource utilization. We use a wide range of dataset sizes from 20 KB to 1 TB to better understand applications' execution. Here, we present our understanding on using our infrastructure to resolve critical system collapse occurred by over-usage of memory resources such as thrashing [1]. Finally, we discuss the sensitivity analysis of our infrastructure with respect to its configured parameters.

4.1 Interface Implementation

We construct a real system infrastructure to implement three processing modules of *KV-SiPC*. We build the prototype using the Samsung SSD. In particular, Figure 4 shows the components that we design or modify (i.e., thread safe KV driver, system support for shared memory threading, OpenMP runtime library, and file <-> key mapping tables) in the compute path and data path to implement *KV-SiPC*. We use the same hardware (i.e., Samsung's PM983 SSD) with different firmware to support either traditional block interface

[1] In computer systems, thrashing occurs when a computer's memory resources are overused, leading to a constant state of paging and page faults, inhibiting most application-level processing.

or KV interface, for fair performance comparison. The routine operations of SSD, such as wear leveling [2] and Garbage Collection (GC) [3], are same for block SSD and KV-SSD while the Flash Translation Layer (FTL)[4] of KV-SSD translates key-value pairs to physical addresses of flash storage.

At the program layer, we maintain file <-> key mapping tables to translate a file-based multi-thread application to key-value based operations. This comprises of the "kvs_map" function and maintains a log of the application data to key mapping. At the kernel and OS system layers of the host system, we further modify the OpenMP runtime library that integrates OpenMP pragma to the functions of the key-value API at the program layer with the key-value related interfaces of the KV driver. We bind the modified OpenMP library functions to the GNU Compiler Collection (GCC) for implementation during the compilation of the program by using the OpenMP flag directive. Additionally, the modified library functions implement the dynamic threads manager, which launches parallel compute and data threads by monitoring different system resources such as CPU and memory. We also build system support for shared memory and threading to implement some new memory management data structures such as "Thread Safe Queues" for key-value concurrency manager to safely handle concurrent shared data. Using these data structures, we ensure lock-free execution of parallel threads through the kernel and OS layers. Our *KV-SiPC* infrastructure supports both synchronous and asynchronous I/Os.

4.2 Evaluation Setup

We use one node of the "Mission Peak" platform [38], which is a 1U server with high-speed NF1 form factor NVMe SSDs. The "Mission Peak" platform is developed with the partnership of the primary storage industry leaders such as Samsung, AIC, Mellanox, E8 Storage, and Memorysolution. We limit the shared memory of our testbed to 100GB. Samsung PM983 Block-based SSDs and KV-SSDs are used in our evaluation. They both have the same flash hardware, capacity, and internal management algorithms, such as garbage collection policy and wear-leveling policy. For our experiments with block SSDs, we use the ext4 filesystem with the block size of 4096 Bytes. Our kernel version is Linux 4.9.5-040905-generic with Ubuntu 16.04.4 LTS (xenial) OS.

[2]Wear leveling is a technique to arrange data, so that write/erase cycles are distributed evenly among all of the blocks in SSD. Wear leveling helps to prolong the service life of erasable computer storage media, such as SSD.
[3]Garbage Collection (GC) is a process to systematically identify which memory cells contain unneeded data and clear the blocks of unneeded data to reclaim the empty space.
[4]The Flash Translation Layer (FTL) maintains an abstraction of read and write requests on logical blocks from the host to turn them into low-level read, erase, and program commands on the underlying physical storage chips.

We use different real multi-thread OpenMP applications such as K-means clustering, Page-Rank, and matrix-matrix multiplication for evaluating the overall benefit of `KV-SiPC`. Later, for an in-depth study of the impact of our infrastructure on different performance parameters, we use the K-means clustering application as a representative multi-thread application. To thoroughly evaluate our infrastructure, we generate workloads by using different numbers of data files such that a small workload has around 1000+ data files and 30 MB in total while a large workload has around 10 million data files and 0.2 TB in total. We compare performance under three processing modules (i.e., sequential (Seq), parallel compute (PC), and parallel data access along with parallel compute (PDA_PC)) as introduced in Section 3.3.

K-means Clustering: Our multi-threaded OpenMP K-means clustering implementation [8] takes color images as the input. Specifically, we use images from NASA Image Galleries [32] as our workloads. To avoid network and web delays, we store databases consisting of 'jpeg,' 'jpg,' and 'png' images on persistent storage. We then cluster pixels in an image based on five features, including three *RGB* channels and the position (x, y) of each pixel. The parameters of K-means clustering include the number of desired clusters (K), the number of iterations (I), and the size of the input dataset (N). The resultant cluster centroids and a list of pixels belonging to each cluster are stored into persistent storage. In our experiments, we consider the total end-to-end runtime, including (1) pre-processing of changing the format of images from 'jpeg,' 'jpg,' and 'png' to vectors with five features (or elements), (2) clustering of data (image pixels) using the K-means clustering algorithm, and (3) storing the results in persistent storage.

4.3 Performance Overview

We measure the parallel I/O performing capabilities of `KV-SiPC` by comparing KV-SSD with block SSD using different number of parallel OpenMP threads. The workload used consists of ten thousand 225KB size different files for block SSD and ten thousand 225KB key-value pairs for KV-SSD. *Figure 5 shows that KV-SSD can saturate the available device bandwidth reported in the datasheet which is represented by a green horizontal line [40] and scale better upon increasing the number of parallel threads due to its lean I/O stack.* On average, we see that KV-SiPC can obtain a speed-up of 2x while performing load and store compared to block SSD.

Next, we measure the overall performance of real OpenMP applications by comparing throughput using traditional I/O stack with block SSD and KV-SiPC with KV-SSD. In Figure 6 we compare the throughput using PDA_PC module where we use maximum available threads for block SSD, and `KV-SiPC` dynamically identifies an optimal number of threads for KV-SSDs using our runtime instrumentation module (KV-ins),

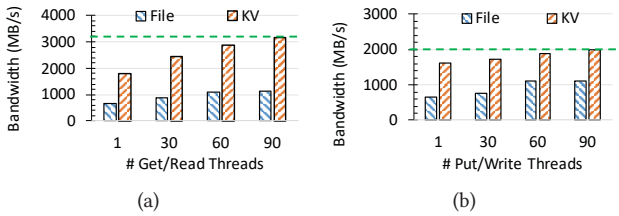


Figure 5: Bandwidth while performing (a) Get/Read and (b) Put/Write with different number of OpenMP threads for ten thousand operations of 225KB load/store

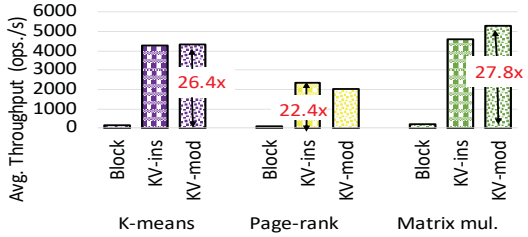


Figure 6: Average application throughput of file-based handling using block SSD and KV based handling using KV-SiPC with KV-SSD for different input data.

and M/M/c modeling module (KV-mod), both as explained in Sec. 3. The average throughput is measured as the number of operations completed per second, where "operation" refers to the end-to-end process of each data point. *We obtain around 25x throughput improvement using KV-SiPC with KV-SSDs when compared to traditional I/O stack with block SSD (see Figure 6).*

Interestingly, we observe that applications with higher temporal big O complexity show a bit better improvement than applications with low complexity. See matrix multiplication with the complexity of $O(n^3)$) (for nxn matrix data) shows 27.8x improvement and page-rank with the complexity of $O(n)$) (for a graph with n nodes) shows 22.4x improvement. We anticipate that this is because the applications with higher complexity usually have more nested loops. Thus, the execution of each parallel compute thread may depend upon multiple data pages as well as the completion of other compute threads. Within such dependencies, the overhead and bottlenecks are higher in a bulky software stack of block SSD than the thin software stack of KV-SSD. Finally, we see that KV-mod is better for the applications with higher complexity, as the runtime instrumentation method (i.e., KV-ins) to configure an optimal number of threads incurs additional contention on CPU resources. We also observe that the throughput improvement, which we obtain by running real applications with KV-SiPC (see Figure 6) is higher than that obtained by simple read and write workload (see Figures 2(a)). We understand that this is because apart from reducing the storage stack overhead, KV-SiPC also simplifies the dependency and increases the concurrency between

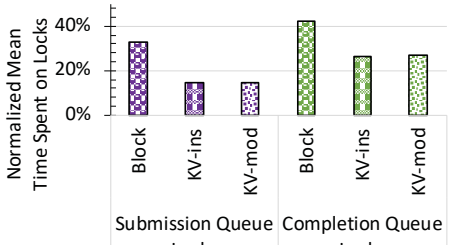


Figure 7: Normalized time spent of locks in all queues with respect to the total time.

Figure 8: Time towards different KV-SiPC operations while running K-means.

parallel compute threads and parallel data threads when compared to block SSD. As real applications can take simultaneous advantage of reduced I/O stack, simplified dependency, increased concurrency, thus they show better performance improvement than just synthetic I/O workloads.

To further find the evidence of the observed performance gain, we analyze time spent waiting for locks. Particularly, we analyze the locks within the "Adapter Queue" [45] of the Linux kernel. Adapter Queue gathers jobs from applications translates the request to `ScsiCommand()` [1] and sends it to the "Device Queue" in the driver. We implement the kernel patch [45, 46] with Lockmeter to compare the time of the locks initialized in the submission and completion queues in the Adapter Queue. The statistic results in Figure 7 show that temporal wastes on these locks with traditional I/O stack is at least 35% higher than KVSiPC. As a result, the bottleneck between the Adapter Queue and the Device Queue is reduced with KVSiPC to be able to better utilize the available 64K queues in the NVMe controller.

Figure 8 shows the breakdown of the time taken by various operations while running K-means clustering for 7,244,954 data files with root mean square error tolerance of 0.01. From Figure 8, we see that PC and PDA_PC mainly reduce the time consumed towards identifying the closest centers in the iterations of K-means. This is the most compute-intensive operation in K-means, so it gets the maximum benefit from parallelism. We also observe that the time consumed to handle KV_Create and KV_Map is very less.

Finally, we study the tail performance of our key-value interface compared to the block device. Figure 9 shows the cumulative distribution functions (CDFs) of the latency under the sequential module with a workload of 40,000 files. We observe that the 90 percentile of the latency (i.e., marked

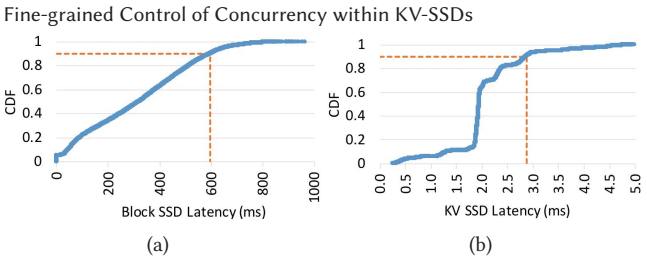


Figure 9: Tail Latency of, (a) Block SSD, and (b) KV-SSD.

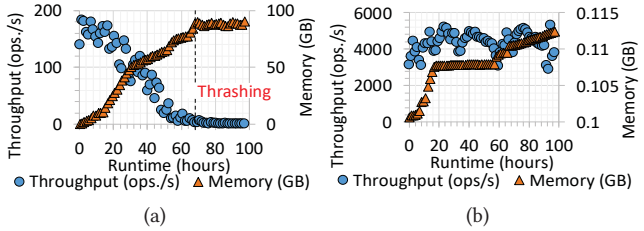


Figure 10: Throughput and Memory Utilization w.r.t. runtime using, (a) Block SSDs, and (b) KV-SSDs.

with dotted lines in Figure 9) is significantly reduced by using KV-SSD with our key-value interface, which is only 3 ms while using block SSD is 600 ms. Similar latency trends are observed under the other programming modules with different workloads and applications. We anticipate that such a reduction in end-to-end tail latency is because *KV-SiPC* does not rely much on the complex host side resource manager (e.g., task scheduler, memory management) which could result in higher performance deviation per thread per I/O, and hence high tail latency.

4.4 Resource Utilization

We further investigate the case of memory over-utilization that causes thrashing and then study the CPU utilization under three different programming modules. When memory resources are overused, the system enters a constant state of paging and page faults, which is called thrashing. Thrashing is undesirable because it can drastically slow down or stop the processing of applications. Thus, *KV-SiPC* strives to address this issue by performing fine-grained resource management to control the number of parallel compute and data access threads. Figure 10 shows memory utilization (right axis) as well as the throughput (left axis) over 100 hours (i.e., four days) under the PDA_PC programming module, where the workload consists of 50 million data files with the size of 1TB in total. In this experiment, we limit the shared memory of the testbed to 100 GB. From Figure 10 (a), we observe that with the block interface, the throughput of processing is initially about 200 ops/s, and memory is under-utilized. However, as the time elapses, the memory utilization keeps increasing until saturation (i.e., almost all 100GB memory space is occupied). Thrashing then happens after 70 hours,

Table 1: CPU Utilization.

	Seq.	PC	PDA_PC
Block SSD	1.11%	99.21%	64.54%
KV-SSD	14.50%	98.56%	21.02%

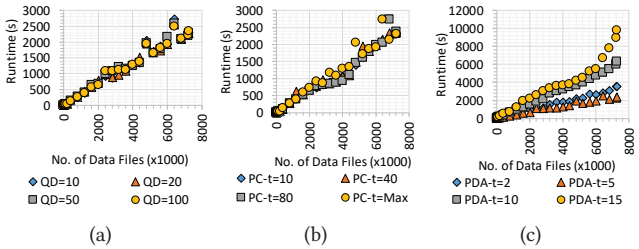


Figure 11: Sensitivity analysis w.r.t., (a) Queue Depth, (b) No. of Compute Threads, and (c) No. of Data Threads.

which dramatically lowers the throughput and then causes the system to crash. This is because block-based SSD, by default, uses block caching that fetches the entire block (including unnecessary pages of data) to memory. As a result, it becomes highly likely that in parallel workloads where data locality of pages accessed by each thread is not good, prefetching such unnecessary pages can occupy a lot of memory space. In contrast, this undesirable thrashing does not occur when using our *KV-SiPC* (Figure 10 (b)). We observe that our key-value interface always achieves the throughput as high as 4,000 ops/s throughout the execution. Meanwhile, the increasing rate of memory utilization is not as fast as that under block SSD. Neither saturation nor thrashing appears even after 100 hours when using our key-value infrastructure. *KV-SiPC* enables an application to store data as values instead of fixed-size blocks in KV-SSD. Thus, only the required data is fetched into memory by each parallel thread, which saves the memory space and conserves CPU cycles for fetching and evicting unnecessary data. Thus, *KV-SiPC* can well utilize the same amount of resources to further accelerate an application with larger workloads.

The average CPU utilization under three programming modules using the block or KV-SSD is further shown in Table 1. We observe that the CPU is underutilized for the sequential module, especially when the block interface is used. On the other hand, CPU resources are almost saturated, making CPU the bottleneck for the parallel compute programming module. The best usage of CPU resources is obtained under the PDA_PC programming module because PDA_PC balances the number of parallel compute threads along with the number of parallel data threads. Additionally, we notice that for PDA_PC module with block SSD still over-utilize the CPU, due to the inherent translation operations of the intermediate layers in block I/O stack.

4.5 Sensitivity Analysis

Finally, we present the sensitivity analysis of *KV-SiPC* with respect to I/O queue depth of submission and completion queues, the number of parallel compute threads, and the number of parallel data access threads. Figure 11 shows the results (i.e., the total runtime as the function of the number of data files) under different parameter settings. Specifically, we initialize the I/O queue depth as 10, the number of parallel compute threads as the maximum (e.g., 96 in our evaluation), and the number of parallel data threads as 2. Then, we vary the values for one of the three parameters in each set of experiments. First, we observe that different queue depths (ranging from 10 to 100) do not affect the performance in our experiments, see Figure 11 (a). This is because the I/O latency is very low when using KV-SSDs so that the I/O queue is never entirely filled with I/Os. Figure 11 (b) shows that different numbers of parallel compute threads obtain almost the same performance for all the workloads. A similar performance trend is also found for other applications. Figure 11 (c) shows the sensitivity analysis of the number of parallel data access threads. The performance gap under different numbers of parallel data access threads becomes visible now. For example, the minimum total runtime is achieved when our testbed uses five concurrent parallel data access threads, i.e., PDA = 5. When the number of parallel data access threads increases, the total runtime decreases initially until resources are saturated and begin to increase later. This is because extra latency is caused by maintaining thread safety for a larger number of data threads, which might be more than the performance benefits. Thus, we think that it is necessary to tune the optimal number of parallel data access threads by monitoring the I/O throughput and memory utilization during the execution.

5 RELATED WORK

In the past decade, a fair amount of developments in the direction of - designing key-value stores to run HPC applications on block storage [17, 22]; key value-based memory management [18, 28]; and development of the storage device firmware supporting key-value I/Os [23, 31], which shows the emerging interest in this direction. The concept of designing application frameworks to overlap the computing and I/O carefully is becoming popular [30, 47]. On the other hand, we begin to realize that storage devices are no longer the bottleneck. Rather minimizing the operating system I/O stack overhead is becoming the key research priorities [26, 27]. Recently, KV-SSDs with the capabilities of running key-value store inside the SSDs has shown to accelerate the processing of key-value objects [10, 20, 23, 34, 36, 48]. Simultaneously, taking advantage of the parallel compute and data accesses, thin OS I/O stack, and content discovery capable smart SSDs such as KV-SSDs may yield even better performance using the same amount of computing and memory resources. However, we are not aware of any prior work that develops such end-to-end infrastructure. In this section, we present state of the art by discussing related existing works and place our *KV-SiPC* in context with them. First, there are lots of persistent key-value stores designed for local systems such as ForestDB [4], and LevelDB [15], which use B+ or LSM tree-based [44] designs to improve performance. Researchers are adopting the key-value management techniques such as LSM tree-based design to built HPC compatible key-value stores [17, 22]. However, all these key-value database techniques are designed for block SSDs, resulting in stacked logs and low-level processing inefficiency.

Second, many existing works have reconsidered the cache system design and directly opened the device-level details of underlying flash storage for key-value caching. Specifically, the DIDACAche [41] provides a key-value interface using an open-channel SSD, mapping values directly to flash blocks. KAML [18] provides a generic caching layer and separate namespaces for different key-value stores. These existing key-value memory management interfaces were not designed considering the requirements of multi-thread applications. Thus, they lack to provide proper concurrency and consistency. Third, towards developing new storage devices that support direct key-value I/Os. The first attempt on this front was Seagate's Kinetic direct-access-over-Ethernet HDDs [31] that incorporate a LevelDB key-value store inside each drive and present a direct key-value interface over ethernet. A recent attempt is being made by storage alliance SNIA, with Samsung [42] being the forerunner among major vendors, to introduce Key-Value Solid State Drive (KV-SSD), along with a standardized Application Programming Interface (API) for key-value storage. However, fundamental research gaps continue to exists regarding software infrastructure to use KV-SSDs to run multi-thread applications.

6 CONCLUSION

We develop a novel storage infrastructure *KV-SiPC* that utilizes the key-value interface to enable execution of multi-threaded applications on KV-SSDs. *KV-SiPC* can simplify application data management by removing intermediate layers (e.g., the filesystem and block layers from OS kernel) and allow the fine-grained control over interactions of parallel compute and parallel data threads to improve system concurrency. Our evaluation demonstrates that *KV-SiPC* can reduce application-to-storage I/O completion latency and better utilize shared memory resources. In the future, we plan to investigate the impact of *KV-SiPC* on the endurance of SSD to address the wear-out issue due to limited PE cycles.

REFERENCES

[1] SCSI Command, 2017. https://en.wikipedia.org/wiki/SCSI_command.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[3] Mohd Abdul Ahad and Ranjit Biswas. Comparing and analyzing the characteristics of hadoop, cassandra and quantcast file systems for handling big data. *Indian J. Sci. Technol.*, 10(8):1–6, 2017.

[4] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers*, (1):1–1, 2016.

[5] Brian Aker and Mark Atwood. MEMSTAT Linux instrumentation tool, 2019. https://linux.die.net/man/1/memstat.

[6] Hesham Altwaijry and Diyab S Alzahrani. Improved-moesi cache coherence protocol. *Arabian Journal for Science and Engineering*, 39(4):2739–2748, 2014.

[7] Ian Angus. An introduction to erlang b and erlang c. *Telemanagement*, 187:6–8, 2001.

[8] Janki Bhimani, Miriam Leeser, and Ningfang Mi. Accelerating K-Means clustering with parallel implementations and GPU computing. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6. IEEE, 2015.

[9] Janki Bhimani, Jingpei Yang, and Changho Choi. Parallel key value based multithread machine learning leveraging kv-ssds, July 16 2020. US Patent App. 16/528,492.

[10] Tim Bisson, Ke Chen, Changho Choi, Vijay Balakrishnan, and Yang-suk Kee. Crail-KV: A High-Performance Distributed Key-Value Store Leveraging Native KV-SSDs over NVMe-oF. In *Performance Computing and Communications Conference (IPCCC), 2018 IEEE 37th International*, pages 1–8. IEEE, 2018.

[11] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[13] Akon Dey, Alan Fekete, and Uwe Röhm. Scalable distributed transactions across heterogeneous stores. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 125–136. IEEE, 2015.

[14] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. Nosql database systems: a survey and decision guidance. *Computer Science-Research and Development*, 32(3-4):353–365, 2017.

[15] Sanjay Ghemawat and Jeff Dean. LevelDB, A fast and lightweight key/value database library by Google, 2014.

[16] Sebastien Godard. MPSTAT Linux instrumentation tool, 2019. https://linux.die.net/man/1/mpstat.

[17] Hugh Greenberg, John Bent, and Gary Grider. {MDHIM}: A parallel key/value framework for {HPC}. In *7th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[18] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 373–384. IEEE, 2017.

[19] Jingpei-Yang. OpenMPDK v0.7.0, 2018. https://github.com/OpenMPDK/KVSSD/.

[20] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel DG Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 144–154, 2019.

[21] Yang Seok Ki. Key Value SSD, September 20 2018. US Patent App. 15/876,028.

[22] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[23] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction support using compound commands in key-value ssds. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[24] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the i/o path: a holistic approach for application performance. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 345–358, 2017.

[25] Los Alamos National Laboratory. Providing order to the world: Range query for KV-SSD, 2020. https://www.lanl.gov/projects/ultrascale-systems-research-center/student-symposiums.php.

[26] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pages 603–616, 2019.

[27] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write dependency disentanglement with {HORAE}. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 549–565, 2020.

[28] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13. ACM, 2011.

[29] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *HotStorage*, pages 8–8, 2014.

[30] Kshitij Mehta and Edgar Gabriel. Multi-threaded parallel i/o for openmp applications. *International Journal of Parallel Programming*, 43(2):286–309, 2015.

[31] Manas Minglani, Jim Diehl, Xiang Cao, Binghze Li, Dongchul Park, David J Lilja, and David HC Du. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In *Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on*, pages 501–510. IEEE, 2017.

[32] NASA. NASA Image Galleries, 2018. https://www.nasa.gov/multimedia/imagegallery/index.html.

[33] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85. IEEE, 2014.

[34] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pınar Tözün. Open-channel ssd (what is it good for). *CIDR, January*, 2020.

[35] Gustavo Pinto, Anthony Canino, Fernando Castor, Guoqing Xu, and Yu David Liu. Understanding and overcoming parallelism bottlenecks in forkjoin applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 765–775. IEEE, 2017.

[36] Rekha Pitchumani and Yang-suk Kee. Hybrid data reliability for emerging key-value storage devices. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 309–322, 2020.

[37] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, San Jose, CA, 2013. USENIX. https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren.

[38] Samsung. Mission Peak: High-performance Storage Solution with NF1 SSD, 2018. https://www.samsung.com/semiconductor/insights/tech-leadership.

[39] Samsung. Samsung Key Value SSD, 2019. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[40] Samsung, 2020. https://samsungsemiconductor-us.com/labs/pdfs/Samsung_PM983_Product_Brief.pdf.

[41] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching. In *FAST*, pages 391–405, 2017.

[42] SNIA. SNIA's key value storage API spec, 2019. https://www.snia.org/sites/default/files/technical_work/PublicReview/Key%20Value%20Storage%20API%200.25-DRAFT.pdf.

[43] Jianfu Wang, Opher Baron, and Alan Scheller-Wolf. M/m/c queue with two priority classes. *Operations Research*, 63(3):733–749, 2015.

[44] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.

[45] Peter Wai Yee Wong, Badari Pulavarty, Shailabh Nagar, Janet Morgan, Jonathan Lahr, Bill Hartner, Hubertus Franke, and Suparna Bhattacharya. Improving linux block i/o for enterprise workloads. In *Ottawa Linux Symposium*, page 390, 2002.

[46] Zhengyu Yang, Morteza Hoseinzadeh, Ping Wong, John Artoux, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. H-NVMe: a hybrid framework of NVMe-based storage system in cloud computing environment. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International*, pages 1–8. IEEE, 2017.

[47] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 45–58, 2015.

[48] Itai Ben Zion. Key-value ftl over open channel ssd. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 192–192, 2019.