A User-space Storage I/O Framework for NVMe SSDs in Mobile Smart Devices

Hyeong-Jun Kim, and Jin-Soo Kim, *Member, IEEE*

***Abstract*—As the application of smart devices becomes more complex, the number of file I/Os has been increased on mobile devices, making the storage performance plays an important role in ensuring better user experience. According to this trend, various researches have been performed to reduce the storage I/O or to improve the performance of the storage device itself. However, existing schemes are not a fundamental solution for improving the storage performance and limited to specific parts of the entire storage layers. In this study, a new storage I/O framework, called NVMeDirect, is proposed to improve the storage performance. The proposed framework improves the performance by allowing applications to access storage directly without any hardware modification. Also, a lightweight filesystem, operated on top of the proposed framework is provided to enable existing applications to be ported on the NVMeDirect framework easily. To evaluate the proposed I/O framework, we have conducted extensive experiments with micro-benchmark and real-world workloads. The experiment results show that, compared to the existing kernel I/O scheme, the proposed framework improves the small file I/O performance by 12.5% and the real-world mobile workload performance by up to 20%.**

***Index Terms*—Mobile devices, Non-volatile memory express, SSD, I/O framework.**

I. Introduction

In the past few years, the use of smart mobile devices has tremendously increased. Beyond the traditional functions of mobile phones, smartphones are used to run various applications such as portable multimedia players, games, web browsers, etc. Also, smart pads are replacing or breaking the boundaries of traditional laptops. In order to provide high performance, manufacturers have developed enhanced hardware components such as CPU and graphic processing unit. In addition, for improving user experience of smart devices, manufacturers are adding various features to applications and frameworks. These trends make mobile applications contain higher quality images and large amount data to increase user satisfaction. Consequently, the amount of data to be read from or written to storage has been increased significantly.

Mobile smart devices use NAND flash-based storage such as embedded multimedia cards (eMMCs). eMMCs have outstanding merits including small size form factor, low power consumption and shock resistance. These advantages contributed to the miniaturization of the device and the widespread adoption as the secondary storage of many mobile smart devices. However, because of the limitation of the underlying interface, the performance of eMMCs are being saturated. Recently, Non-Volatile Memory Express (NVMe) interface has been standardized to support high performance storage based on the PCI Express (PCIe) interconnect. Based on the NVMe standard, NVMe PCIe solid state drive (SSD) in a single ball grid array (BGA) package has been announced. The BGA form factor SSDs are currently being adopted by high-end smart phones and smart pads for providing high storage performance. In spite of this effort, storage is still often blamed as a major source of performance bottleneck in ensuring better user experience in smart devices.

In order to increase the storage performance further, industry and academia have conducted extensive researches including both hardware and software techniques. Several studies have focused on combining emerging non-volatile memory (NVM) technologies such as phase change memory (PCM), spin-torque transfer magneto-resistive memory (STT-MRAM) and transistor-less cross [1], [2] with the conventional software stack to optimize the storage stack overheads on mobile devices. These memories are designed to be byte addressable and accessible at a similar latency to DRAM. For example, some researchers focused on utilizing NVM on the legacy I/O stack of operating system, including file system layer [3], [4] and page caches [5], [6]. Others studied utilizing NVM on specific application or database engine of mobile devices [7]-[9]. These studies either employed a small size of NVM as a cache memory of storage, or replaced the whole main memory or storage devices with NVM. However, prior works are inappropriate for mobile smart devices because employing a small size of NVM as a cache for specific applications is not cost-effective. Replacing main memory or storage with NVM also incurs high cost and significantly extends time-to-market.

Other researchers tried to reduce the software overhead of storage stack by optimizing the I/O stacks of mobile devices. These include introducing mobile device-specific I/O policies

Manuscript received December 30, 2016; accepted February 28, 2017. Date of publication April 12, 2017. This work was supported partly by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2016R1A2A1A05005494). *(Corresponding author: Jin-Soo Kim.)*

Hyeong-Jun Kim is with College of Information and Communication Engineering, Sungkyunkwan University, 2066, Seobu-ro, Jangan-gu, Suwon 16419, South Korea (e-mail: hjkim@csl.skku.edu).

Jin-Soo Kim is with College of Information and Communication Engineering, Sungkyunkwan University, 2066, Seobu-ro, Jangan-gu, Suwon 16419, South Korea (e-mail: jinsookim@ skku.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCE.2017.014709

or subsystems, and reducing the number of I/O requests by optimizing the use of *fsync()* system calls [12]-[16]. However, these works concentrated on reducing the number of I/Os, instead of optimizing the fundamental I/O stack overhead.

To optimize the fundamental I/O stack overhead in kernel, many researchers have tried to reduce the kernel overhead by using the polling mechanism [17], [18] and eliminating unnecessary context switching [19], [20]. However, kernel-level I/O optimizations have a couple of limitations to satisfy the requirements of user applications. First, the kernel should be general because it provides an abstraction layer for applications, managing all the hardware resources. Thus, it is hard to optimize the kernel without loss of generality. Second, the kernel cannot implement any policy that favors a certain application because it should provide fairness among applications. Therefore, it would be desirable if a user-space I/O framework is supported for high-performance storage devices which enables the optimization of the I/O stack in the user space without any kernel intervention.

Recently, some industries announced libraries for accessing NVMe SSDs in the user space. However, these libraries only work for a single application because it moves the whole NVMe driver from the kernel to the user space and assigns it to the specific application. In addition, these libraries currently support only basic I/Os on raw block devices. We believe these libraries are inappropriate to be adopted for mobile smart devices, because many smart devices are equipped with a single storage device and applications of smart devices are run on file systems instead of the block-level storage.

In this paper, we propose a novel user-level I/O framework called NVMeDirect, which improves the performance by allowing the user applications to access the storage device directly. The previous version of this work [21] can coexist with the legacy I/O stack of the kernel, allowing applications to access NVMe SSDs in the user space. However, because the previous version only supports I/Os on the raw block devices, it was not suitable for user applications on consumer smart devices. This paper enhances the previous version of our work in terms of *usability* by providing user-space file system layer to make it easy to port user applications on the proposed user- space I/O framework. In addition, this paper includes evaluation results using real-world application workloads of smartphones. Our results show that the proposed framework and filesystem outperforms the existing kernel-based I/O by up to 23% on micro-benchmark and by 20% on real-world workload benchmark.

The remainder of this paper is organized as follows. Section II describes the overview of the NVMe interface and presents the motivation of this work. Section III presents the design of the NVMeDirect framework in detail. Section IV introduces the evaluation methodology and presents evaluation results in terms of performance with synthetic and real-world workloads. The related work is described in Section V. Finally, Section VI concludes the paper.

II. Background and Motivation

This section describes the overview of the NVMe interface and then presents the I/O stack overhead in the existing kernel software stack to motivate this work.

A. Non-volatile Memory Express (NVMe)

NVM Express (NVMe) is a high performance and scalable host controller interface for PCIe-based SSDs. Legacy storage protocols such as SCSI, SATA, and Serial Attached SCSI (SAS), were designed to support rotational storage devices such as hard disk drives (HDDs). All of these protocols communicate with the host using an integrated on-chip or an external Host Bus Adaptor (HBA). eMMC also uses some of legacy storage protocols for compatibility with existing operating systems. As NAND flash-based storage evolves, their internal bandwidth capabilities exceeded the bandwidth supported by the external interface connecting the drives to the host system. Thus, the performance improvement of NAND flash-based storage has been limited by the maximum throughput of the interface protocol itself. Moreover, since access latencies of NAND flash memory are orders of magnitude lower than that of rotational media, protocol inefficiencies and external HBA became a dominant contributor to the overall access time. These reasons led to an effort to transition from existing storage protocols to NVMe, based on a scalable, high-bandwidth, and low-latency I/O interconnect, namely PCI express (PCIe).

The notable feature of NVMe is to offer multiple queues to process I/O commands. Each I/O queue can manage up to 64K commands and a single NVMe device supports up to 64K I/O queues. When issuing an I/O command, the host system places the command into the submission queue and notifies the NVMe device using the doorbell register. After the NVMe device processes the I/O command, it writes the result to the completion queue and raises an interrupt to the host system. NVMe enhances the performance of interrupt processing by MSI/MSI-X and interrupt aggregation. In the current operating systems, the NVMe driver creates a submission queue and a completion queue per core in the host system to avoid locking overhead and cache collision.

Fig. 1 illustrates a high-level representation of the differences between the I/O stack of SATA SSD and NVMe SSD. An NVMe-based I/O request bypasses the conventional block layer request queue and uses internal hardware submission queue and completion queue pairs. Because NVMe SSD uses multiple hardware queues and each queue is bound to a CPU, NVMe driver does not need to acquire a lock to guarantee data consistency. These mechanisms allow to reduce the overhead of software, and thus improving performance and scalability.

B. Motivation

As applications become more complex and they request more storage I/Os, the storage performance plays an important role in ensuring better user experience. In particular, the latency reduction in the storage I/Os is very important on mobile smart devices, because user interaction occurs frequently on mobile devices.

As shown in Fig. 2, NVMe outperforms SATA SSD when the host reads a 4KB data, due to the enhanced storage interface and optimized software stack. However, NVMe SSD still consumes over 15% of its time on the software I/O stack.

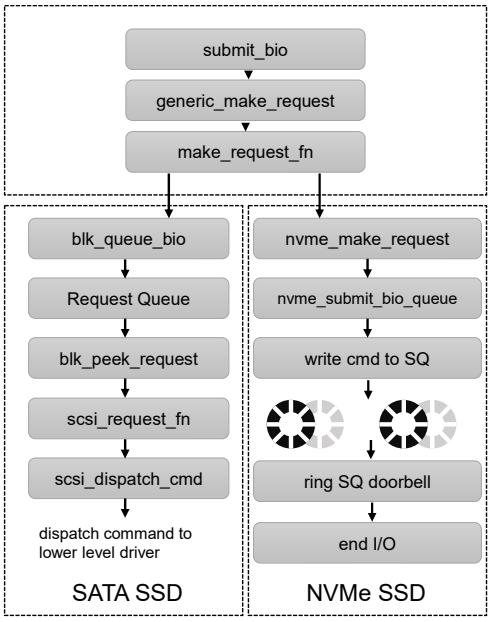


Fig. 1. High-level representation of I/O software stack for SATA SSD and NVMe SSD.

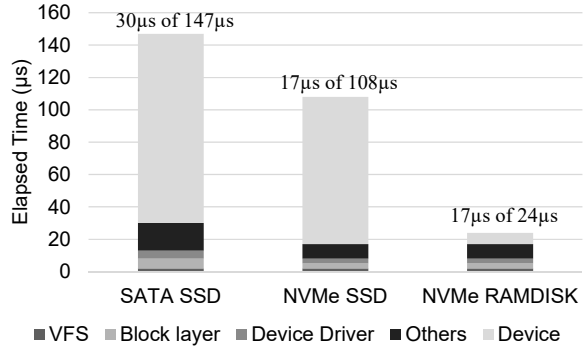


Fig. 2. Comparison of elapsed time of each layer on SATA SSD, NVMe SSD and NVMe RAMDISK.

NVMe RAMDISK in Fig. 2 represents the latency breakdown for the same condition over the NVMe device consisting of DRAMs. In this case, we can observe now the system spends 70% of its time in the software I/O stack. It is apparent that the faster the memory technology becomes, the more the overhead in the kernel software I/O stack will hinder the performance improvement.

Even if we use the state-of-the-art schemes for reducing the kernel software stack overhead, it is not possible to eliminate the kernel overhead completely due to the additional work associated with system call processing, permission check, memory allocation, and memory copy. In particular, most mobile smart devices are used by a single user and each application is executed in some form of sandbox, which is provided by the OS framework. These techniques allow only authorized applications to access storage devices. In other words, some kernel components on the storage I/O path can be eliminated or simplified. Hence, bypassing unnecessary features in the kernel and accessing the storage device directly in the user space is desirable for maximizing the performance of NAND flash-based storage devices and emerging high performance storage device in mobile smart devices. Also, under the proposed user-space I/O framework, each application can control the number of I/O queues flexibly according to its own requirement, which is not possible in the existing kernel-level I/O stack.

III. Design and Implementation

This paper proposes a novel user-level I/O framework called NVMeDirect, which enable user applications to access storage device without any kernel invention. The NVMeDirect framework consists of a light-weight user-level file system called NVMeDFS and basic block-level I/O mechanisms based on polling which can be configured freely according to the characteristics of applications. This section outlines how the NVMeDirect framework is designed in order to obtain optimal performance with NVMe SSD in mobile smart devices.

A. System Overview

Fig. 3 shows the overall structure of NVMeDirect I/O Framework. This framework avoids the overhead of the kernel I/O stack by supporting direct accesses between the user application and the NVMe SSD on mobile smart devices to achieve high performance with low latency. The framework is composed of two components: *NVMeDirect Library* and user-space library filesystem called *NVMeDFS*.

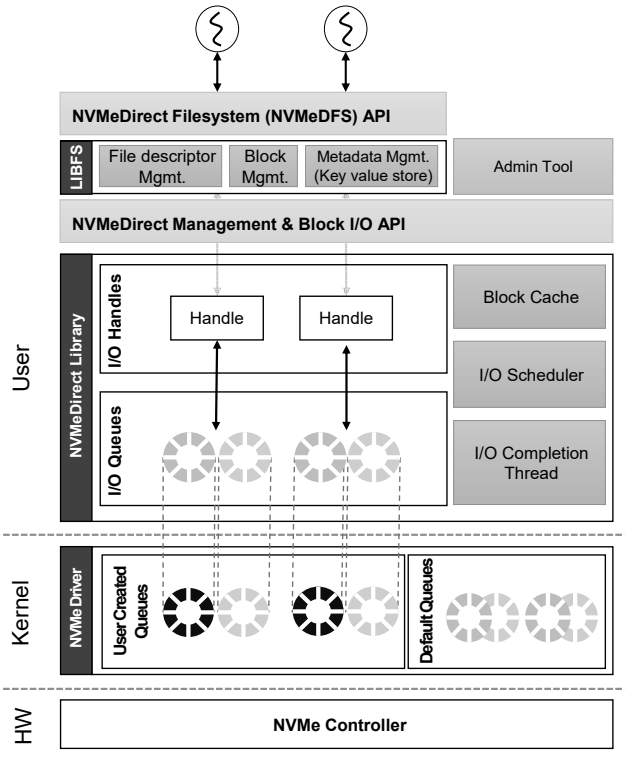


Fig. 3. The overall structure of NVMeDirect I/O Framework for accessing NVMe SSDs directly from user-level applications.

The NVMeDirect Library enables user applications to access NVMe devices as a block storage on user-space application

without kernel I/O stacks such as block device layer. In addition, The NVMeDirect Library supports several I/O features to perform various I/O policies as performed in the kernel. Because many applications in consumer devices are designed and implemented based on the file-level interface, filesystem is essential for accessing storage using the file abstraction instead of the simple block device abstraction.

NVMeDFS is a light-weight library filesystem for performing I/Os on NVMeDirect using conventional file-level APIs. This library file system enables user-space application to see the NVMe device as a file-level storage. Applications can request file I/Os to NVMeDFS with provided APIs and NVMeDFS uses NVMeDirect library APIs internally for performing I/Os on NVMe devices.

B. NVMeDirect Library

The NVMeDirect library is designed to fully utilize the performance of NVMe SSDs while meeting the diverse requirements from user applications. The library exports NVMe storage device to user application directly as a block-level storage. Fig. 3 illustrates the overall architecture of the NVMeDirect I/O framework and components of the NVMeDirect library. The library consists of three components: Management I/O, Basic I/O and Extended I/O.

The management I/O component is composed of *Admin tool* and *kernel module*. Admin tool controls the kernel module with the root privilege to manage the access permission of I/O queues with *ioctl*(). When an application requests to create a queue, the kernel module checks whether the application is allowed to perform user-level I/Os. And then it creates the required NVMe I/O queue pairs, and maps their memory regions and the associated doorbell registers to the user-space memory region of the application. After creating queues and mapping memory region is completed, the application can issue I/O commands directly to the NVMe SSD using Basic I/O component of NVMeDirect without any hardware modification or help from the kernel I/O stack.

The Basic I/O components consists of I/O handles and I/O queues. I/O queues are memory-mapped address space for NVMe I/O queues, which is created in the kernel address space. NVMeDirect library offers the notion of I/O handles to send I/O requests to NVMe queues. I/O handle can be configured to use different features such as caching, I/O scheduling, and I/O completion according to the characteristics of the I/O requests. As shown in Fig. 4, a thread can create one or more I/O handles to access the queues and each handle can be bound to a dedicated queue or a shared queue. For example, Applications

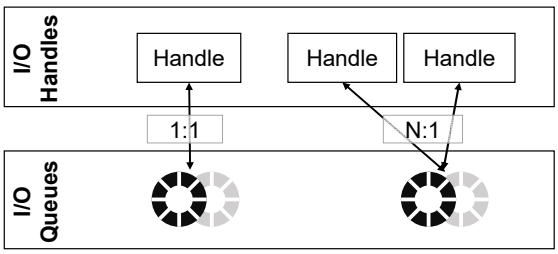


Fig. 4. An example of I/O queues and I/O handles binding. I/O handles and I/O queues can map flexible for utilizing I/O framework effeciency.

can share a single queue of NVMeDirect with multiple handles for asynchronous I/Os and dedicate a single queue with a handles for synchronous I/Os. These mapping of handles and queues provide an efficient and fast I/O service to synchronous I/O requests which make applications wait for I/O to complete.

The last component of the NVMeDirect library is Extended I/O component which provides various I/O helper functions such as block cache, I/O scheduler, and I/O completion thread. As mentioned in the Basic I/O component, applications can use these components depending on the characteristics of I/O. Block cache is similar to the page cache in the kernel, which manages the cache memory in 4KB unit size. The NVMe interface uses the physical memory addresses for I/O commands. Thus, Block cache maintains pre-translated physical addresses for each memory unit to avoid address translation overhead during the actual I/Os. I/O scheduler issues I/O commands for asynchronous I/O operations or when an I/O request is dispatched from the Block cache. The I/O completion thread is implemented as a standalone thread to check the completion status of the I/Os using polling. Since the interrupt-based I/O completion incurs context switching and additional software overhead, it is not suitable for high performance and low latency I/O devices [18]. Polling mechanism is especially suitable for mobile smart devices, because the mobile smart devices occur user interaction frequently and the majority of file accesses is known as small random I/Os.

However, when fewer storage I/Os are requested from application, the polling mechanism wastes most of CPU cycles in busy waiting. In particular, polling is not efficient in a large sequential access such as reading or writing multimedia files, which is sensitive to bandwidth rather than latency. To address this problem, the NVMeDirect library utilizes a dedicated polling thread with dynamic polling period control based on the number of I/O requests and the size of I/O request to avoid unnecessary CPU usage.

The NVMeDirect library exposes various APIs such as *nvmed_read*(), *nvmed_write*(), *nvmed_flush*(), *nvmed_discard*(), etc. that can be used by any user applications. These APIs interact with various components of the library and eventually issue NVMe commands such as read, write, flush, and discard that correspond to each API. Also, the NVMeDirect framework supports APIs for managing queues and handles such as *nvmed_create_queue*(), *nvmed_create_handle*(), and *nvmed_set_param*().

C. NVMeDirect File System (NVMeDFS)

NVMeDirect File System, called *NVMeDFS*, is designed to enable file-level I/Os on high performance NVMe devices. The existing user-level I/O libraries (including the previous version of NVMeDirect) lack the support for file systems. They merely present a storage device as a block device (i.e. an array of fixed-size blocks) to applications. This makes it very difficult for most applications in mobile smart devices to take advantage of user-level I/O libraries without extensive modifications to applications and and/or execution framework. To resolve this problem, NVMeDFS is developed in the form of a lightweight

shared library which provides POSIX-like interfaces to any applications at the user level.

The internal architecture of NVMeDFS is illustrated in Fig. 5. NVMeDFS divides the storage device into two regions: metadata region and data block region. Filesystem metadata such as superblock, block bitmaps and inodes are managed using an embedded key-value store with an index structure called HB+Trie [22]. Because the key-value store maintains data using logging and checkpoint, it can guarantee durability and consistency of metadata effectively without any additional journaling mechanism. Since the key-value store itself is initially designed to run on a file system, we have modified the storage engine of the key-value store so that it operates on top of NVMeDirect library. Due to the nature of log-structured design, the original key-value store requires periodic compaction to reclaim the space occupied by deleted or updated data. In order to remove additional I/O overhead incurred during compaction, NVMeDFS reuses invalid blocks without making free space thorough compaction. In addition, NVMeDFS uses fixed-size metadata to reduce internal fragmentation while blocks are reused.

Traditional filesystem accesses data blocks by pathname traversal though several metadata and data blocks such as directory entry and inodes blocks. This conventional structures occurs a significant amount of storage I/O operations and becomes overhead on accessing file data. Unlike the traditional filesystem, NVMeDFS maintains this information using a key-value pair, where key is the full pathname and value is the corresponding inode number. Whenever a file is created, this information is inserted into the key-value store. This scheme enables applications to access metadata and data blocks of the file with low latency.

(*KEY*) Pathname → (*VALUE*) size, timestamp, etc…

Because of the common prefix pathname such as parent directory, indexing based on the full pathname may incur storage space overhead. However, the key-value store in NVMeDFS uses the HB+Trie structure which provides common prefix compression. Using this feature, NVMeDFS avoids storage overhead by duplicated parts of the pathname (Fig 6).

Each file is broken up into data blocks of fixed size. The data block size is 4KB and the contiguous blocks can be represented by a single extent as in the Ext4 filesystem.

NVMeDFS supports POSIX-like APIs with *nvmedfs* prefix. Applications can access files using APIs such as *nvmedfs_open*(), *nvmedfs_read*(), and *nvmedfs_write*(). Every function of filesystem uses NVMeDirect library functions internally to perform I/Os on NVMe SSDs. NVMeDFS supports various options for file I/O including sync / async, and direct / buffered I/O as in the kernel I/O. These options use the features of the I/O handles in the NVMeDirect library. This approach enables applications to efficiently utilize the NVMe storage devices according to the characteristics of each I/O.

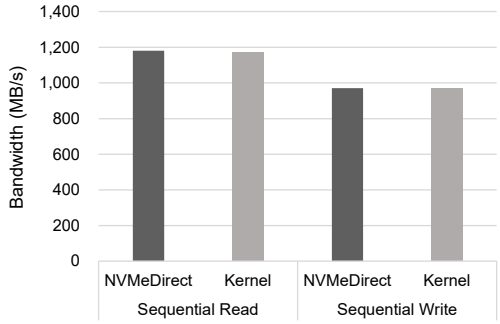
IV. Evaluation

The performance of the proposed NVMeDirect framework has been evaluated using several micro-benchmarks and real-world mobile workloads on a commercial NVMe SSD. In the experiments, we compare the performance of the proposed framework with that of the legacy kernel I/O stack.

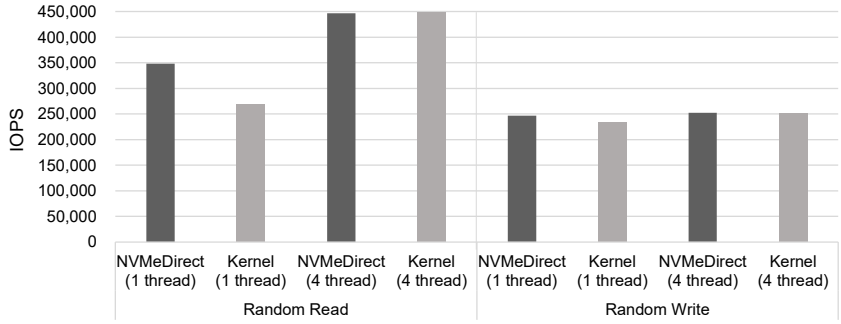
A. Baseline I/O Performance

First, we use a micro-benchmark to measure the baseline performance of the NVMeDirect framework. Fig. 5 depicts the baseline performance of sequential I/O (Fig. 5a) and random I/O (Fig. 5b) on NVMeDirect and Kernel I/O, respectively. For measuring sequential read or write performance, the micro-benchmark issues I/O requests of 128KB to the storage device using a single thread. When performing sequential read or write on the NVMe SSD, the performance meets or exceeds the maximum performance of the device on both NVMeDirect and Kernel I/O regardless. Because the performance is high enough to meet the requirement of mobile applications, such as storing or streaming 4K or higher-quality video, NVMe SSDs are effective in providing high user experience on mobile smart devices.

Fig. 5b shows the random read and write performance on NVMeDirect and Kernel I/O. In term of user responsiveness on mobile smart devices, random I/O performance is more important than sequential I/O performance. For measuring random I/O performance, the micro-benchmark issues I/O requests of 4KB to the storage device with the queue depth of 32. Because the majority of mobile devices has at least quad-core CPUs, the test is repeated not only for a single thread, but also for four threads. When the number of I/O requests is enough, the performance of random reads and writes



(a) Sequential I/O performance comparison



(b) Random I/O performance comparison

Fig. 5. Comparison of the baseline performance of NVMeDirect framework and Kernel I/O.

meets the performance specification of the device as shown in the result of four threads. However, NVMeDirect achieves higher IOPS compared to Kernel I/O on the single thread test in both reads and writes. This is because NVMeDirect avoids the overhead of the kernel I/O stack by supporting direct accesses between the user application and the NVMe SSD. On the single thread test, NVMeDirect framework records 23% higher read IOPS and 5% higher write IOPS than kernel I/O.

B. *Key-value Store Performance*

Key-value store is a widely-used data storage, which is designed for storing, retrieving, and managing unstructured data. In comparison to the RDBMS, which was used as a data repository for structured data, key-value store is easier to use and provides higher performance on unstructured data. Key-value store is being widely used in mobile smart devices as well, as the storage for storing various types of data generated from the applications.

To measure the performance of the key-value store on the NVMeDirect framework, a synthetic micro-benchmark is used whose key size and value size follow the zipf distribution. The average key size and the average value size of the workload is set to 32 bytes and 512 bytes, respectively. Initially, one million key-value pairs are created and each reader and writer threads perform *Get()* or *Put()* operations for 10 minutes. To compare the results from the existing kernel-based I/O stack, the benchmark is performed both on the raw block device and on the ext4 filesystem.

Fig. 7 illustrates the operations per second in Ext4, raw block device, and NVMeDirect while running the micro-benchmark. As shown in Fig. 7, key-value store on the NVMeDirect framework outperforms the other cases with Ext4 and raw block device. The Ext4 filesystem and the raw block device show almost similar results, even though the Ext4 filesystem includes software overhead, such as block allocation, metadata management, and journaling. Because the micro-benchmark performs on a single large file, the overhead of the Ext4 filesystem is negligible. Compared to the kernel-based I/O stack, the NVMeDirect framework improves the throughput by about 12% on *Get()* operations and by 17% on *Put()* operations. This is because NVMeDirect reduces the latency by eliminating the software overhead of the kernel.

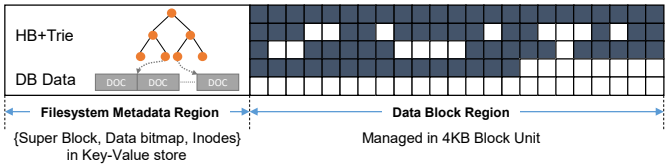


Fig. 6. The overall structure of NVMeDirect File System. Filesystem manage its metadata using key-value store to minimize the management overhead.

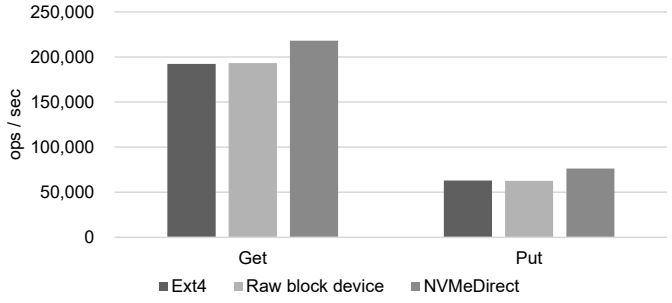


Fig. 7. Performance comparison of key-value store on Ext4, Raw block device and NVMeDirect framework.

Most smart device users are sensitive to the latency of the read operations because they usually sit idle while the result is delivered. Therefore, a longer delay time than a certain threshold results in degraded user experience on mobile smart devices. TABLE I shows the tail latency of *Get()* operations at the 90th, 95th, and 99th percentile while performing the micro-benchmark. For the 99th percentile latency, performing a *Get()* operation on the Ext4 filesystem or on the raw block device shows about 18~20x higher latency than the 90th percentile latency, while the NVMeDirect framework shows only 2.6x higher latency. We can see that the NVMeDirect framework shows more stable performance, being very effective in reducing the tail latency.

TABLE I
TAIL LATENCY OF GET OPERATION BY STORAGE ENGINE

Percentile	Ext4 Filesystem	Raw Block Device	NVMeDirect Framework
90	24 µs	23 µs	23 µs
95	74 µs	75 µs	25 µs
99	438 µs	473 µs	59 µs

C. *Filesystem Performance*

According to our analysis on mobile smartphones, about 50% of files in mobile storage are less than or equal to 4KB in size. In other words, the filesystem performance for small files is very important on mobile smart devices. Based on these observation, we have conducted an experiment for measuring the filesystem performance especially for small files. Our synthetic micro-benchmark measures the elapsed time of creating, deleting, reading, and writing operation over a large number of small files. The micro-benchmark performs 5 million operations on one million files. Each operation is randomly chosen from create, delete, read, and append operations. The file sizes range from 4KB to 16KB, and the data size of each operation is set to 4KB. We compare the performance of our NVMeDFS with that of Ext4, which is one of the most widely-used filesystems in mobile smart devices.

Fig. 8 shows the total elapsed time of performing the entire micro-benchmark (denoted by bars) and the amount of bytes written (denoted by lines) on each filesystem. As shown in the line graphs in Fig. 8, the amount of bytes written is reduced by 23% when we use NVMeDFS. Because the Ext4 filesystem uses journaling for metadata consistency and crash recovery, metadata is written twice in the filesystem, which requires more time and space than actually needed. However, NVMeDFS can avoid duplicated write for consistency of filesystem metadata, because NVMeDFS uses the logging scheme of the underlying key-value store for ensuring filesystem metadata consistency. This leads to a significant reduction in the total elapsed time. As shown in the bar graphs in Fig. 8, NVMeDFS is 12.5% faster than the Ext4 filesystem by 12.5% on small file accesses.

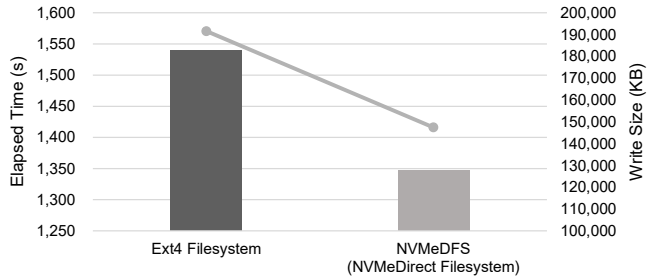


Fig. 8. Filesystem performance comparison of Ext4 vs. NVMeDFS. Bar graph denotes elapsed time, and line graph denotes write size.

D. Real-World Mobile Workload Performance

The last experiment evaluates the performance of three different mobile applications by replaying the database query traces. These traces are collected by a mobile embedded database on real mobile smart devices with running popular mobile applications for about a week. The messenger trace was obtained from a well-known smartphone messenger application that sends and receives a lot of text messages. The social network service (SNS) trace was collected by reading news feeds and writing comments on those feeds. Finally, the mail trace is obtained by smartphone mail client application that sends and receives a lot of mails. Because each of these applications stores and reads most of data on the embedded database, the collected workloads represent the entire storage I/O of the application and the reduction in the execution time means the increased storage performance for the application.

These real-world workloads are fed to the the mobile embedded database which works on Ext4 or on NVMeDFS with the NVMeDirect framework. Fig. 9 shows the execution time of each workload on Ext4 and NVMeDFS. The execution time is normalized to the execution time on the Ext4 filesystem. Compared to the Ext4 filesystem, the execution time is reduced by up to 20% on NVMeDFS. Note that NVMeDFS avoids storage space overhead needed by journaling by using key-value stores for metadata consistency. However, as shown in the previous study [12], journaling on the Ext4 filesystem incurs a lot of storage I/O requests. Table II compares the difference in the number of write requests issued for each workload. From Fig. 9 and Table II, we can see that the performance gain of NVMeDFS mainly comes from reducing the number of write requests and the fast processing of those requests by bypassing the kernel I/O stack.

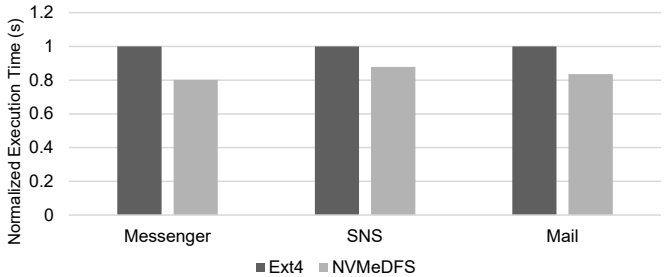


Fig. 9. Normalized execution time of real-world mobile application database workloads on Ext4 and NVMeDFS

TABLE II
THE NUMBER OF STORAGE WRITE REQUESTS

Workload	Ext4 Filesystem	NVMeDFS	Difference
Messenger	22,987	18,492	24%
SNS	13,666	12,549	9%
Mail	147,263	121,419	21%

V. Related Work

There have been several studies for improving the storage performance of mobile smart devices. This section briefly describes the related work.

Kim *et al*. [14] presents the evidence that storage performance indeed affects application performance on smart devices and suggests several solutions for improving application performance, such as RAIDs, log-structured file system, and application modification. Jeong *et al.* [12] investigates the relationship between embedded database systems and filesystem journaling, highlighting the "journaling of journaling (JoJ)" problem. Shen *et al.* [13] also focused on the JoJ problem and suggests an adaptive journaling mode for resolving the problem. Ngyen et al. [11] investigates the I/O behavior on smartphones and proposes the SmartIO approach to minimize application I/O delay. Jeong *et al.* [16] studies the effectiveness of asynchronous I/O on mobile devices and suggested QASIO scheme to reduce the latency some important asynchronous I/O requests. These works are focused on resolving abnormal behavior between OS and application or optimizing the kernel I/O path on mobile smart devices.

In order to improve the performance of mobile devices, some researchers suggest employing NVM in mobile devices. Kim *et al*. [7] proposes a database engine, which reduces the number of synchronous writes by using NVM as a dedicated logging area. Oh *et al*. [8] also suggests a database engine of mobile devices to gain the advantages of NVM. Kim *et al*. [10] proposes the delta journaling scheme that reduces the journaling overhead of the filesystem layer using a small-sized NVM. Lin *et al*. [5] proposes a page cache replacement policy based on hierarchical memory structure. However, prior work is inefficient because employing NVM is restricted to page cache or logging area of filesystem or database. Also, adopting NVM in consumer devices is not cost-effective and extends time-to-market.

Caufield *et al*. [17] proposes a flexible file-system architecture that exposes the storage-class memory to user applications to access storage without kernel interaction. This approach is somewhat related to the proposed NVMeDirect framework. However, their studies require special hardware while the NVMeDirect framework can be utilized on any consumer devices equipped with commercial NVMe SSDs.

VI. Conclusion

In mobile devices, storage device plays an important role in providing better user experience, as applications become more complex and I/O-intensive. According to these trends, researchers have strived to improve the storage performance of smart consumer devices. However, the previous approaches are

limited to specific application or kernel layer and practically unacceptable on consumer smart devices.

In order to improve the performance of mobile smart devices, this paper suggests the new NVMeDirect framework, which makes the application access to the NVMe storage directly in user space without any hardware modification. The framework achieves 23% higher random read IOPS and 5% higher random write IOPS on the NVMe SSD compared to the existing kernel I/O. This paper also proposes a light-weight user-space library filesystem, called NVMeDFS, which manages metadata consistency and durability using the key-value store. This filesystem makes it easy to port existing applications over the NVMeDirect framework. Evaluation results on NVMeDFS confirm that NVMeDFS and NVMeDirect framework improves small file I/O performance by 12.5%, and real-world mobile workload by up to 20% over the existing Ext4 filesystem and kernel I/O.

REFERENCES

[1] S. Mittal, and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1537-1550, May 2016.

[2] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Proc. USENIX Conference Hot Topics in Operating Systems*, April 2011, pp. 2:1-2:5.

[3] H. G. Lee and S. Ryu, "High-performance NAND and PRAM hybrid storage design for consumer electronics," *IEEE Transactions on Consumer Electronics*, vol. 56, no. 1, pp. 112-118, Feb. 2010.

[4] K. Zhong, T. Wang, X. Zhu, L. Long, D. Liu, W. Liu, Z. Shao and E.-H.-M. Sha, "Building high-performance smartphones via nonvolatile memory: the swap approach," in *Proc. ACM International Conference on Embedded Software*, New Delhi, India, pp. 1-10, Oct. 2014.

[5] Y.-J. Lin, C.-L. Yang, H.-p. Li, and C.-Y. M. Wang, "A buffer cache architecture for smartphones with hybrid DRAM/PCM memory," in *Proc. IEEE Non-Volatile Memory System and Applications Symposium*, Hong Kong, China, pp. 1-6, Aug. 2015.

[6] J. Park, E. Lee, and H. Bahn, "DABC-NV: A buffer cache architecture for mobile systems with heterogeneous flash memories," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 4, pp. 1237-1245, Nov. 2012.

[7] D. Kim, E. Lee, S. Ahn, and H. Bahn, "Improving the storage performance of smartphones through journaling in non-volatile memory," *IEEE Transactions on Consumer Electronics*, vol. 59, no. 3, pp. 556-561, Aug. 2013.

[8] G. Oh, S. Kim, S.-W. Lee, and B. Moon, "SQLite optimization with phase change memory for mobile applications," in *Proc. International Conference on Very Large Database*, Kohala Coast, Hawaii, pp. 1454-1465, Aug. 2015.

[9] H. Luo, L. Tian, and H. Jiang, "qNVRAM: Quasi non-volatile RAM for low overhead persistency enforcement in smartphones," in *Proc. USENIX Workshop on Hot Topics in Storage and File Systems*, Philadelphia, USA, pp. 1-5, Jun. 2014.

[10] J. Kim, C. Min, and Y. I. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 2, pp. 217-224, May 2014.

[11] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang, "Reducing smartphone application delay through read/write isolation," in *Proc. ACM Annual International Conference on Mobile Systems, Applications, and Services*, Florence, Italy, pp. 287-300, May 2015.

[12] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proc. USENIX Annual Technical Conference*, San Jose, USA, pp. 309-320, Jun. 2013.

[13] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free," in *Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 287-293, Feb. 2014.

[14] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 1-14, Feb. 2012

[15] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," In *Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 273-286, Feb. 2015.

[16] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting quasi-asynchronous I/O for better responsiveness in mobile devices," in *Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 191-202, Feb. 2015.

[17] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," In *Proc. international conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, pp. 387-400, Mar. 2012.

[18] J. Yang, D.B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proc. USENIX Conference on File and Storage Technologies*, Santa Clara, USA, pp. 1-7, Feb. 2015.

[19] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "OS I/O path optimizations for flash solid-state drives," in *Proc, USENIX Annual Technical Conference*, Philadelphia, USA, pp. 483-488, Jun. 2014.

[20] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, and H. Y. Yeom, "Optimizing the block I/O subsystem for fast storage devices," *ACM Transactions on Computer Systems*, vol. 32 no. 2, pp. 6:1-6:48, June 2014.

[21] H.-J. Kim, Y. S. Lee, J.-S. Kim, "NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs," In *Proc. USENIX Workshop on Hot Topics in Storage and File Systems*, Denver, USA, pp. 1-5, Jun. 2016.

[22] J. S. Ahn, C. Seo, R. Mayuram, R. Yaseen, J.-S. Kim, and S. Maeng "ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 902-915. 2016.

Hyeong-Jun Kim received the B.S. degree in Electrical and Computer Engineering from Sungkyunkwan University (SKKU), Korea in 2010 and M.S. degree in Electrical Computer Engineering from SKKU in 2012. He is currently a Ph.D. candidate in the Department of IT Convergence at SKKU.

His current research interests include storage systems, and operating systems and embedded systems.

Jin-Soo Kim (M'89) received the B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He is currently a professor in Sungkyunkwan University (SKKU).

Before joining SKKU, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Prof. Kim is a member of the IEEE and the IEEE Computer Society.