# NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs

Hyeong-Jun Kim
*Sungkyunkwan University*
*hjkim@csl.skku.edu*

Young-Sik Lee
*KAIST*
*yslee@calab.kaist.ac.kr*

Jin-Soo Kim
*Sungkyunkwan University*
*jinsookim@skku.edu*

## Abstract

The performance of storage devices has been increased significantly due to emerging technologies such as Solid State Drives (SSDs) and Non-Volatile Memory Express (NVMe) interface. However, the complex I/O stack of the kernel impedes utilizing the full performance of NVMe SSDs. The application-specific optimization is also difficult on the kernel because the kernel should provide generality and fairness.

In this paper, we propose a user-level I/O framework which improves the performance by allowing user applications to access commercial NVMe SSDs directly without any hardware modification. Moreover, the proposed framework provides flexibility where user applications can select their own I/O policies including I/O completion method, caching, and I/O scheduling. Our evaluation results show that the proposed framework outperforms the kernel-based I/O by up to 30% on microbenchmarks and by up to 15% on Redis.

## 1 Introduction

The new emerging technologies are making a remarkable progress in the performance of storage devices. NAND flash-based Solid State Drives (SSDs) are being widely adopted on behalf of hard disk drives (HDDs). The next-generation non-volatile memory such as 3D XPoint [8] promises the next step for the storage devices. In accordance with the improvement in the storage performance, the new NVMe (Non-Volatile Memory Express) interface has been standardized to support high performance storage based on the PCI Express (PCIe) interconnect.

As storage devices are getting faster, the overhead of the legacy kernel I/O stack becomes noticeable since it has been optimized for slow HDDs. To overcome this problem, many researchers have tried to reduce the kernel overhead by using the polling mechanism [5, 13] and eliminating unnecessary context switching [11, 14].

However, kernel-level I/O optimizations have several 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. Lastly, the frequent update of the kernel requires a constant effort to port such application-specific optimizations.

In this sense, 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. In particular, such a user-space I/O framework can have a great impact on modern data-intensive applications such as distributed data processing platforms, NoSQL systems, and database systems, where the I/O performance plays an important role in the overall performance. Recently, Intel released a set of tools and libraries for accessing NVMe SSDs in the user space, called SPDK [7]. However, SPDK only works for a single user and application because it moves the whole NVMe driver from the kernel to the user space.

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. Our approach leverages the standard NVMe interface and works on commercial NVMe SSDs without any hardware modification. Since the user-space I/O framework does not go through the kernel during actual I/Os, it allows for many optimization opportunities. Unlike SPDK, NVMeDirect can co-exist with the legacy I/O stack of the kernel so that the existing (kernel-based) applications can use the same NVMe SSD with NVMeDirect-enabled applications simultaneously on different disk partitions. Another advantage of NVMeDirect over SPDK is that the proposed framework provides flexibility in queue management, I/O completion method, caching, and I/O scheduling where each user application can select its own I/O policies according to its I/O characteristics and requirements. For example, if an application requires a super-fast latency, it can allocate a dedicate queue and use the polling mechanism after issuing I/O commands directly to the NVMe SSDs. One may implement a differentiated I/O service inside of an application by isolating a queue for high-priority I/O requests. This can be useful for database servers where boosting the I/O performance of logging is known to be important to improve the overall performance [9, 10].

Our experimental results on a commercial NVMe SSD indicate that the proposed scheme outperforms the

kernel-based I/O by up to 30% on microbenchmarks and by up to 15% on Redis. We also show that the proposed NVMeDirect framework can provide the differentiated service successfully by boosting the prioritized I/O requests.

## 2 Background

NVM Express (NVMe) is a high performance and scalable host controller interface for PCIe-based SSDs [1]. 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 notify the NVMe device using the *doorbell* register. After the NVMe device processes the I/O command, it writes the results 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 Linux kernel, the NVMe driver creates a submission queue and a completion queue per core in the host system to avoid locking and cache collision.

## 3 NVMeDirect I/O Framework

### 3.1 Design

We develop a user-space I/O framework called NVMeDirect to fully utilize the performance of NVMe SSDs while meeting the diverse requirements from user applications. Figure 1 illustrates the overall architecture of our NVMeDirect framework.

The *Admin tool* controls the kernel driver with the root privilege to manage the access permission of I/O queues. When an application requests a single I/O queue to NVMeDirect, the user-level library calls the kernel driver. The kernel first checks whether the application is allowed to perform user-level I/Os. And then it creates the required submission queue and the completion queue, and maps their memory regions and the associated doorbell registers to the user-space memory region of the application. After this initialization process, the application can issue I/O commands directly to the NVMe SSD without any hardware modification or help from the kernel I/O stack. NVMeDirect offers the notion of *I/O handles* to send I/O requests to NVMe queues. 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. According to the characteristics of the workloads, each handle can be configured to use different features such as caching, I/O scheduling, and I/O completion. The major APIs supported by the NVMeDirect framework are summarized in Table 1. NVMeDirect also provides various wrapper APIs that correspond to NVMe commands such as read, write, flush, and discard.

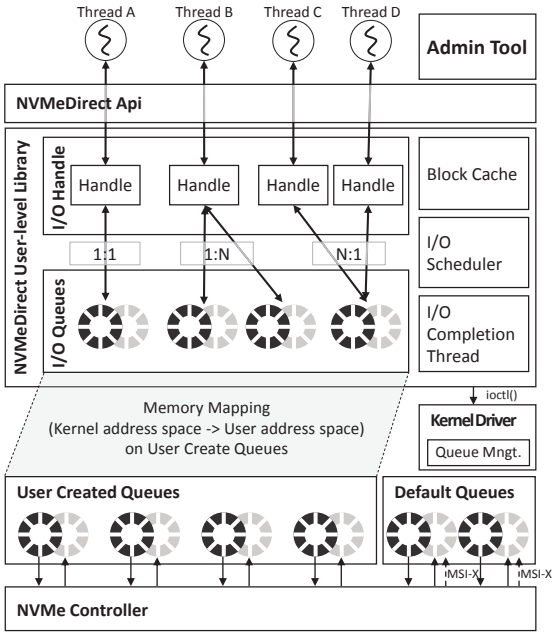



Figure 1: The overall architecture of NVMeDirect.

Separating *handles* from *queues* enables flexible grouped I/O policies among multiple threads and makes it easy to implement differentiated I/O services. Basically, a single I/O queue is bound to a single handle as Thread A in Figure 1. If a thread needs to separate read requests from write requests to avoid read starvation due to bulk writes, it can bind multiple queues to a single handle as Thread B in Figure 1. It is also possible to share a single queue among multiple threads as Thread C and D in Figure 1.

NVMeDirect also offers *block cache*, *I/O scheduler*, and *I/O completion thread* components for supporting diverse I/O policies. Applications can mix and match these components depending on their needs. *Block cache* manipulates the memory for I/O in 4KB unit size similar to the page cache in the kernel. Since the NVMe interface uses the physical memory addresses for I/O commands, the memory in the block cache utilizes pre-translated physical addresses to avoid address translation overhead. *I/O scheduler* issues I/O commands for asynchronous I/O operations or when an I/O request is dispatched from the block cache. 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 [13]. However, if an application is sensitive to bandwidth rather than to latency, polling is not efficient due to the significant increase in the CPU load. Thus, NVMeDirect utilizes a dedicated polling thread with dynamic polling period control based on the I/O size or a hint from applications to avoid unnecessary CPU usage.

| API | Description |
|---|---|
| `nvmed = nvmed_open(dev_path)` | Open and get the information of device |
| `result = nvmed_close(nvmed)` | Close the NVMe device |
| `queue = nvmed_create_queue(nvmed)` | Create an I/O queue and map it to user-space |
| `result = nvmed_destroy_queue(queue)` | Delete an I/O queue |
| `handle = nvmed_create_handle(queue)` | Create an I/O handle bound to a specific I/O queue |
| `handle = nvmed_create_mq_handle(queues)` | Create an I/O handle bound to multiple I/O queues |
| `result = nvmed_destroy_handle(handle)` | Delete an I/O handle |
| `result = nvmed_set_param(handle, param, val)` | Set a parameter for I/O queue or handle |
| `buffer = nvmed_get_buffer(num_pages)` | Allocate a buffer |
| `result = nvmed_put_buffer(buffer)` | Deallocate a buffer |

Table 1: Major APIs defined in the NVMeDirect framework.

## 3.2 Implementation

The NVMeDirect framework is composed of three components: queue management, admin tool, and user-level library. The queue management and the admin tool are mostly implemented in the NVMe kernel driver, and user-level library is developed as a shared library.

We implement the queue management module in the NVMe driver of the Linux kernel 4.3.3. At the initialization stage, the admin tool notifies the access privileges and the number of available queues to the queue management module with `ioctl()`. When an application requests to create a queue, the queue management module checks the permission and creates a pair of submission and completion queues. The module maps the kernel memory addresses of the created queues and the doorbell to the user's address space using `dma_common_mmap()` to make them visible for the user-level library. The module also exports the memory addresses via the `proc` file system. Then, the user-level library can issue I/O commands by accessing the memory addresses of queues and doorbell registers.

The block cache in the user-level library allocates the memory when an application requests to create a buffer for the buffered I/Os. The memory in the block cache is indexed by the radix tree and allocated by the `nvmed_get_buffer()` function.

The I/O completion thread is implemented as a standalone thread to check the completion of I/O using polling. Multiple completion queues can share a single I/O completion thread or a single completion queue can use a dedicated thread to check the I/O completion. The polling period can be adjusted dynamically depending on the I/O characteristics of applications. Also, an application can explicitly set the polling period of the specific queue using `nvmed_set_param()`. The I/O completion thread uses `usleep()` to adjust the polling period.

# 4 Evaluation

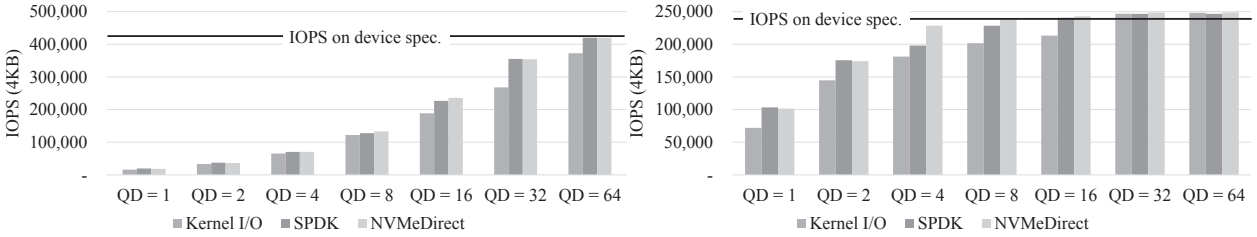
We compare the I/O performance of NVMeDirect with the original kernel-based I/O with asynchronous I/O support (Kernel I/O) and SPDK using the Flexible IO Tester (fio) benchmark [3]. For all the experiments, we use a Linux machine equipped with a 3.3GHz Intel Core i7 CPU and 64GB of memory running Ubuntu 14.04. All the performance evaluations are performed on a commercial Intel 750 Series 400GB NVMe SSD.

## 4.1 Baseline Performance

Figure 2 depicts the IOPS of random reads (Figure 2a) and random writes (Figure 2b) on NVMeDirect, SPDK, and Kernel I/O varying the queue depth with a single thread. When the queue depth is sufficiently large, the performance of random reads and writes meets or exceeds the performance specification of the device on both NVMeDirect, SPDK, and Kernel I/O. However, NVMeDirect achieves higher IOPS compared to Kernel I/O until the IOPS is saturated. 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. As shown in Figure 2a, we can see that our framework records the near maximum performance of the device with the queue depth of 64 for random reads, while Kernel I/O has 12% less IOPS in the same configuration. In Figure 2b, when NVMeDirect achieves the maximum device performance, Kernel I/O shows 20% less IOPS. SPDK shows the same trend as NVMeDirect because it also accesses the NVMe SSD directly in the user space.

## 4.2 Impact of the Polling Period

Figure 3 shows the trends of IOPS (denoted by lines) and CPU utilization (denoted by bars) when we vary the polling period per I/O size. The result is normalized to the IOPS achieved when the polling is performed without delay for each I/O size. We can notice that a significant performance degradation occurs in a certain point for each I/O size. For instance, if the I/O is 4KB in size, it is better to shorten the polling period as much as possible because the I/O is completed very quickly. In case of 8KB and 16KB I/O sizes, there is no significant slowdown, even though the polling is performed once in 70µs and 200µs, respectively. At the same time, we can re-




(a) Random read performance comparison.

(b) Random write performance comparison.

Figure 2: Asynchronous I/O performance of Kernel I/O, SPDK, and NVMeDirect varying the queue depth (QD).

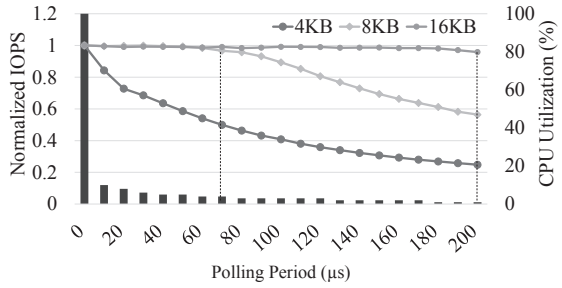



Figure 3: IOPS and CPU Utilization by the polling period per I/O size normalized to the best performance

duce the CPU utilization due to the polling thread to 4% for 8KB I/Os and 1% for 16KB I/Os. As mentioned in Section 3.1, we use the adaptive polling period control based on this result to reduce the CPU utilization associated with polling.

## 4.3 Latency Sensitive Application

The low latency is particularly effective on the latency sensitive application such as key-value stores. We evaluate NVMeDirect on one of the latency sensitive applications, Redis, which is an in-memory key value store mainly used as database, cache, and message broker [2]. Although Redis is an in-memory database, Redis writes logs for all write commands to provide persistence. This makes the write latency be critical to the performance of Redis. To run Redis on NVMeDirect, we added 6 LOC (lines of code) for initialization and modified 12 LOC to replace POSIX I/O interface with the corresponding NVMeDirect interface with the block cache. We use the total 10 clients with workload-A in YCSB [6], which is an update heavy workload.

Table 2 shows the throughput and latency of Redis on Kernel I/O and NVMeDirect. NVMeDirect improves the overall throughput by about 15% and decreases the average latency by 13% on read and by 20% on update operations. This is because NVMeDirect reduces the latency by eliminating the software overhead of the kernel.

## 4.4 Differentiated I/O Service

I/O classification and boosting the prioritized I/Os is important to improve the application performance such as

| | Throughput | Latency | |
|---|---|---|---|
| | (ops/s) | Read (µs) | Update (µs) |
| Kernel I/O | 46,563 | 209.18 | 217.61 |
| NVMeDirect | 53,570 | 183.05 | 173.32 |

Table 2: Redis performance comparison on 10 clients.

writing logs in database systems [9, 10]. NVMeDirect can provide the differentiated I/O service easily because the framework can apply different I/O policies to the application using I/O handles and multiple queues.

We perform an experiment to demonstrate the possible I/O boosting scheme in NVMeDirect. To boost specific I/Os, we assign a prioritized thread to a dedicated queue while the other threads share a single queue. For the case of non-boosting mode, each thread has its own queue in the framework. Figure 4 illustrates the IOPS of Kernel I/O and two I/O boosting modes of NVMeDirect while running the fio benchmark. The benchmark runs four threads including one prioritized thread and each thread performs random writes with a queue depth of 4. As shown in the result, the prioritized thread with a dedicated queue on NVMeDirect outperforms the other threads remarkably. In the case of Kernel I/O, all threads have the similar performance because there is no mechanism to support prioritized I/Os. This result shows that NVMeDirect can provide the differentiated I/O service without any software overhead.

# 5 Related Work

There have been several studies for optimizing the storage stack as the hardware latency is decreased to a few milliseconds. Shin et al. [11] present a low latency I/O completion scheme based on the optimized low level hardware abstraction layer. Especially, optimizing I/O path minimizes the scheduling delay caused by the context switch. Yu et al. [14] propose several optimization schemes to fully exploit the performance of fast storage devices. The optimization includes polling I/O completion, eliminating context switches, merging I/O, and double buffering. Yang et al. [13] also present that the polling method for the I/O completion delivers higher

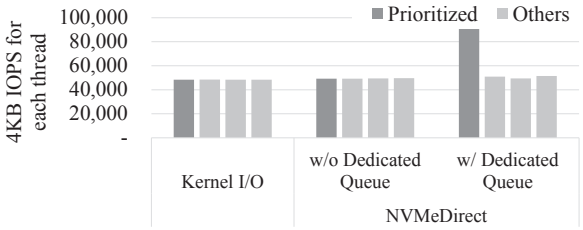



Figure 4: Random write performance with I/O priority.

performance than the traditional interrupt-driven I/O.

Since the kernel still has overhead in spite of several optimizations, researchers have tried to utilize direct access to the storage devices without involving the kernel. Caulfield et al. [5] present user-space software stacks to further reduce storage access latencies based on their special storage device, Moneta [4]. Likewise, Volos et al. [12] propose a flexible file-system architecture that exposes the storage-class memory to user applications to access storage without kernel interaction. These approaches are similar to the proposed NVMeDirect I/O framework. However, their studies require special hardware while our framework can run on any commercial NVMe SSDs. In addition, they still have complex modules to provide general file system layer which is not necessary for all applications.

NVMeDirect is a research outcome independent of SPDK [7] released by Intel. Although NVMeDirect is conceptually similar to SPDK, NVMeDirect has following differences. First, NVMeDirect leverages the kernel NVMe driver for *control-plane* operations, thus existing applications and NVMeDirect-enabled applications can share the same NVMe SSD. In SPDK, however, the whole NVMe SSD is dedicated to a single process who has all the user-level driver code. Second, NVMeDirect is not intended to be just a set of mechanisms to allow user-level direct accesses to NVMe SSDs. Instead, NVMeDirect also aims to provide a large set of I/O policies to optimize various data-intensive applications according to their characteristics and requirements.

## 6 Conclusion

We propose a user-space I/O framework, NVMeDirect, to enable the application-specific optimization on NVMe SSDs. Using NVMeDirect, user-space applications can access NVMe SSDs directly without any overhead of the kernel I/O stack. In addition, the framework provides several I/O policies which can be used selectively by the demand of applications. The evaluation results show that NVMeDirect is a promising approach to improve application performance using several user-level I/O optimization schemes.

Since NVMeDirect does not interact with the kernel during the I/Os, it cannot provide enough protection normally enforced by the file system layer. In spite of this, we believe NVMeDirect is still useful for many data-intensive applications which are deployed in a trusted environment. As future work, we plan to investigate ways to protect the system against illegal memory and storage accesses. We are also planning to provide user-level file systems to support more diverse application scenarios. NVMeDirect is available as opensource at `https://github.com/nvmedirect`.

## Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Nisha Talagala, for their valuable comments. This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1503-03.

## References

[1] NVM Express Overview. `http://www.nvmexpress.org/about/nvm-express-overview/`.

[2] Redis. `http://redis.io/`.

[3] AXBOE, J. Flexible IO tester. `http://git.kernel.dk/?p=fio.git;a=summary`.

[4] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proc. MICRO* (2010), pp. 385–395.

[5] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. In *Proc. ASPLOS* (2012), pp. 387–400.

[6] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proc. SOCC* (2010), pp. 143–154.

[7] INTEL. Storage performance development kit. `https://01.org/spdk`.

[8] INTEL, AND MICRON. Intel and Micron Produce Breakthrough Memory Technology. `http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology`, 2015.

[9] KIM, S., KIM, H., KIM, S.-H., LEE, J., AND JEONG, J. Request-oriented durable write caching for application performance. In *Proc. USENIX ATC* (2015), pp. 193–206.

[10] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A case for flash memory SSD in enterprise database applications. In *Proc. SIGMOD* (2008), pp. 1075–1086.

[11] SHIN, W., CHEN, Q., OH, M., EOM, H., AND YEOM, H. Y. OS I/O path optimizations for flash solid-state drives. In *Proc. USENIX ATC* (2014), pp. 483–488.

[12] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. EuroSys* (2014).

[13] YANG, J., MINTURN, D. B., AND HADY, F. When poll is better than interrupt. In *Proc. FAST* (2012), p. 3.

[14] YU, Y. J., SHIN, D. I., SHIN, W., SONG, N. Y., CHOI, J. W., KIM, H. S., EOM, H., AND YEOM, H. Y. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 6.