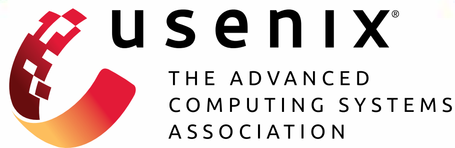


# NVMeVirt: A Versatile Software-defined Virtual NVMe Device

Sang-Hoon Kim, *Ajou University;* Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim, *Seoul National University*

https://www.usenix.org/conference/fast23/presentation/kim-sang-hoon

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

# NVMeVirt: A Versatile Software-defined Virtual NVMe Device

Sang-Hoon Kim
*Ajou University*

Jaehoon Shim
*Seoul National University*

Euidong Lee
*Seoul National University*

Seongyeop Jeong
*Seoul National University*

Ilkueon Kang
*Seoul National University*

Jin-Soo Kim
*Seoul National University*

## Abstract

There have been drastic changes in the storage device landscape recently. At the center of the diverse storage landscape lies the NVMe interface, which allows high-performance and flexible communication models required by these next-generation device types. However, its hardware-oriented definition and specification are bottlenecking the development and evaluation cycle for new revolutionary storage devices.

In this paper, we present NVMeVirt, a novel approach to facilitate software-defined NVMe devices. A user can define any NVMe device type with custom features, and NVMeVirt allows it to bridge the gap between the host I/O stack and the virtual NVMe device in software. We demonstrate the advantages and features of NVMeVirt by realizing various storage types and configurations, such as conventional SSDs, low-latency high-bandwidth NVM SSDs, zoned namespace SSDs, and key-value SSDs with the support of PCI peer-to-peer DMA and NVMe-oF target offloading. We also make cases for storage research with NVMeVirt, such as studying the performance characteristics of database engines and extending the NVMe specification for the improved key-value SSD performance.

## 1 Introduction

NAND flash memory gains significant popularity for consumer devices and enterprise servers, and the fast advancement of semiconductor technologies fosters the non-volatile memory (NVM) to build storage devices, enlightening high-density low-latency storage devices. Nowadays, we can purchase off-the-shelf storage devices, which feature tens of microsecond latency and several GiB/s of bandwidth [16, 47].

Along with the performance and data density improvement, there has been an active trend toward making storage devices smarter and more capable. For efficient and effective data processing and management, many innovative device concepts have been proposed, including but not limited to Open-Channel SSD (OCSSD) [5, 34, 41], zoned namespace SSD (ZNS SSD) [4, 12], key-value SSD (KVSSD) [14, 19, 23, 45], and computational storage [8, 11, 20, 29, 31, 33, 52]. These new types of devices are significantly diversifying the storage device landscape. In this trend, software-based storage emulators are becoming more important than ever. For instance, when academia and/or industry propose an innovative storage device concept, fully developing an actual product from the conceptual idea takes a while. Meanwhile, we can implement a new concept in an emulator and see its benefits and pitfalls while running real workloads. This can provide us invaluable insights, facilitating rapid design space exploration. Moreover, by collecting various performance metrics from the emulator, we can understand the I/O characteristics of operating systems and the applications. This information can be used to optimize both the software and hardware of the target system. Finally, each emulator has a sophisticated performance model along with many knobs that can control a certain performance characteristic of the emulated device. This can help us predict the application performance on future storage devices that exhibit different performance characteristics.

However, to the authors' best knowledge, none of the previously proposed emulators fully satisfy the requirements to be used in the modern storage environment. Many emerging device types are often optimized to their primary targeting workloads and require a customized communication model between the host and device. This requirement makes the *NVMe interface* the most preferred interface for the emerging device types due to its flexibility and extendibility. This implies that a proper storage emulator should provide a comprehensive method to customize at the NVMe interface level. However, emulating the full NVMe interface in software is challenging as the NVMe interface inherently involves the protocol defined at the hardware level. Previous work proposes to circumvent the difficulty of emulating the NVMe interface by interposing hooks in the host NVMe device driver or leveraging virtualization technologies [12, 32, 35, 55]. However, these approaches fail to present a suitable NVMe device instance that is fully functional in the diverse modern storage environments such as when the kernel is being bypassed [24, 54] or when a

| | Simulators | | Emulators | | | | |
|---|---|---|---|---|---|---|---|
| | **Trace-driven [30,36]** | **Full-system [10,22,49]** | **VM-based [12,32,55]** | **Block-driver level [56]** | **NVMe-driver level [35]** | **HW platforms [21,28]** | **NVMeVirt** |
| **Deployable in real environments** | No | Yes | Yes | Yes | Yes | Yes | **Yes** |
| **Execution speed** | Fast | Very slow | Slow | Fast | Fast | Real-time | **Real-time** |
| **NVMe Multi-queue support** | No | Yes | Yes | No | Yes | Yes | **Yes** |
| **NVMe interface modification** | Impossible | Easy | Easy | Impossible | Easy | Difficult | **Easy** |
| **Low-latency device support** | Possible | Possible | Difficult | Possible | Possible | Difficult | **Possible** |
| **Kernel bypassing with SPDK** | No | No | No | No | No | Yes | **Yes** |
| **PCI peer-to-peer DMA support** | No | No | No | No | No | Yes | **Yes** |
| **NVMe-oF target offloading** | No | No | No | No | No | Yes | **Yes** |

Table 1: Comparisons of various virtual storage device approaches.

device directly accesses storage devices through NVMe-over-fabrics or PCI peer-to-peer communication [3,9,42]. Table 1 compares the previous approaches and their limitations.

This paper presents NVMeVirt, a storage emulator facilitating software-defined NVMe devices. NVMeVirt is a Linux kernel module providing the system with a virtual NVMe device of various kinds. Currently, NVMeVirt supports conventional SSDs, NVM SSDs, ZNS SSDs, and key-value SSDs. The device is emulated at the PCI layer, presenting *a native NVMe device to the entire system*. Thus, NVMeVirt has the capability not only to function as a standard storage device, but also to be utilized in advanced storage configurations, such as the NVMe-oF target offloading, kernel bypassing, and PCI peer-to-peer communication. In addition, this level of emulation allows developers to modify the NVMe interface layer easily, making it possible to explore various design spaces over NVMe and to support new device types. Unlike other emulators with similar goals, NVMeVirt does not rely on the virtualization technology, allowing comprehensive communication models at a consistently low overhead. The performance of these devices can be controlled with several performance knobs, making the virtual device perform close to real devices. Hence, NVMeVirt opens up a new opportunity for co-designing highly intelligent storage devices over the NVMe interface and stimulates the invention of a novel storage device architecture.

In the evaluation, we demonstrate the supported features of various device types with a working prototype. We explain two case studies to demonstrate that NVMeVirt can be helpful for storage domain research and engineering. The source code of NVMeVirt is publicly available at https://github.com/snu-csl/nvmevirt. The followings are the contributions of this paper:

- Provide a software framework to facilitate NVMe device research with various and stable I/O characteristics.
- Envision the fast prototyping and development of NVMe devices and interface through a software-defined NVMe device.
- Analyze the correlation and impact between the application and storage performance using representative database benchmarks.
- Make a case for extending the NVMe interface to improve key-value SSDs.

The rest of the paper is organized as follows. Section 2 explains the background and related work. Section 3 discusses the motivation of our work and explains the internal structure of NVMeVirt. Section 4 shows the evaluation result of NVMeVirt by representing its flexibility and feasibility. Finally, Section 5 concludes the paper.

# 2 Background and Related Work

## 2.1 NVM Express Standard

In modern computer architectures, peripheral devices are often connected to the processor through Peripheral Component Interconnect Express (PCIe) links [18,42]. PCIe defines the entire communication stack, from the layout of connector pins to the message protocol between the host and devices. NVM Express, or NVMe, was first aimed to extend the PCIe communication protocol for emerging non-volatile memory devices, such as solid-state drives (SSDs). As designed from the ground up for modern storage devices, NVMe provides a more efficient low-latency interface than legacy interfaces, such as Small Computer System Interface (SCSI) and Serial ATA (SATA). Later, the NVMe specifications are extended further to support various storage device types, such as zoned namespace (ZNS) SSDs [4,12] and key-value SSDs [14,19,23,45].

The latest NVMe 2.0 specifications were announced in June, 2021. They comprise multiple documents: NVMe Base Specification, Command Set Specifications (NVM Command Set specification, ZNS Command Set specification, KV Command Set specification), Transport Specifications (PCIe Transport specification, Fibre Channel Transport Specification, RDMA Transport Specification, and TCP Transport Specification) and the NVMe Management Interface Specification. The Base Specification defines the host control interface. The Command Set Specifications contain the host-to-device protocol for SSD commands used by operating systems for read/write/flush/trim operations, firmware management, error handling, etc. As observed from the Transport Specifications,

NVMe operations can be performed over various transport layers, such as PCIe, TCP/IP, and remote DMA (RDMA). Specifically, combined with RDMA-capable transport, NVMe-oF allows NVMe commands to be delivered to remote nodes and directly routed to target devices [9]. If the network adapter supports the target offloading feature, the NVMe commands can be processed completely at the hardware level without any involvement of software layers on the remote node. Thus, NVMe-oF can minimize the latency for disaggregated storage and is considered a key technology for high-performance scalable storage systems in future data centers.

## 2.2 NVMe Operation

The NVMe specifications standardize two communication interfaces for NVMe devices: *NVMe control block* and *NVMe message queues*. The NVMe control block is the primary path for setting up the NVMe device. It contains several configuration fields with which the host device driver sets up the device. Specifically, the host can specify the location of the administration queue pair, set and clear the interrupt mask, point to the address of the controller memory buffer (CMB), and shutdown and restart the device.

Meanwhile, the NVMe message queue is the interface primarily for scalable I/O. The NVMe architecture supports up to 65,535 I/O queues each with 65,535 commands (called queue depth). The queue can be created, modified, or destroyed by submitting requests to the special queue called *administration queue*. To perform I/O, the host driver builds an NVMe command according to the specification and submits it to the *submission queue* of the NVMe device. Each queue has the associated *doorbell*, which indicates the index of the latest request in the queue. When the host driver alters the doorbell, the NVMe device senses the change and starts processing the enqueued requests. The completion of the request processing is handled in a similar manner. Each submission queue has a paired *completion queue*, whereas the submission queue and the completion queues are collectively called a *queue pair*. When the I/O request processing is completed, the device places an NVMe completion message in the paired completion queue of the submitted request. The device driver on the host can sense the moment of completion by either polling the completion queue or waiting for an interrupt from the device. After processing the completion message, the device driver notifies the device of the completion by setting the doorbell of the completion queue. Accordingly, the device releases the resources associated with completed requests.

The administration queue pair is initialized during device initialization by specifying a physical address in the NVMe control block. The host can ask the device to create regular queue pairs by posting queue creation messages into the administration queue. The host can also make device management requests, such as identifying the device ID, querying supported features, and setting up an interrupt for completion notification, through the administration queue pair. The administration requests are processed in the same manner as regular I/O requests.

## 2.3 Related Work

A myriad of studies has attempted to imitate real storage devices in software [10,12,21,22,28,30,32,35,36,49,55,56]. As summarized in Table 1, we can classify these works into two categories: simulators and emulators. Simulators imitate the internal operations of real devices with a data processing model [10, 22, 30, 36, 49]. They often build the model for a target device, parameterize the performance of internal operations, and calculate desired performance metrics from the model. They enable a detailed analysis with sophisticated models. However, they are often limited as they rely on a trace collected from real systems or are extremely slow when the full system is simulated to run the real workload.

Emulators provide *device instances* to the host; hence, they can be used like a real device [12,21,28,32,35,55,56]. FlexDrive [35] proposes a software-defined NVMe device, similar to our work. By modifying the NVMe device driver, it controls the flow rate of I/O requests in the host I/O stack, allowing the exploration of the space of various device performances. Combined with a RAM disk, FlexDrive can be used for projecting the performance of future devices. However, it can only emulate the conventional SSD type and not the emerging devices, such as KVSSDs and ZNS SSDs. Also, because of its implementation as a modified device driver, it can only handle the simplest data communication where requests are coming down through the kernel I/O stack, thereby unable to support complicated I/O models, such as the NVMe-oF offloading, kernel bypassing, P2P device communication, etc. Finally, the NVMe driver is on the critical path of the host I/O subsystem, so it might be too intrusive to be applied to a working system.

FEMU [32] proposes an accurate and scalable virtual NVMe device using host virtualization technology. Specifically, FEMU provides guest operating systems with a virtual NVMe device by leveraging the device virtualization feature of the QEMU [44]. According to the split driver model, the frontend in the VM receives the NVMe commands, and forwards them to the FEMU backend running in the host operating system. Due to this organization, it requires to keep switching between the host and guest operating systems, incurring non-negligible and highly variable latency (see Section 4.2). In addition, the virtualized environment inherently limits the control of the virtualized device implementation. For example, to perform DMA (and RDMA), the PCI device should be able to access the memory in the DMA/physical address space of the host. This becomes complicated in the virtual machine environment where the guest physical memory is scattered in the host physical memory through the virtual memory schemes on the host. This prohibits the study and

exploration of device-oriented approaches in particular, such as NVMe-oF-based technologies and P2P device communication.

# 3 NVMeVirt Internals

## 3.1 Motivation

The increasing demand for high-performance and efficient storage devices has been pushing the academia and industry to develop various new storage device types, such as NVM SSD, KVSSD, ZNS SSD, and computational storage. They usually require a custom host-device interface tailored to their primary target workloads to make them work most effectively. For example, KVSSDs are for handling a huge number of small key-value pairs. They are most effective only when the host-device communication layer can handle small key-value payloads efficiently. Computational storage devices require a mechanism to deliver the code to run on the device. In this sense, we can claim that the most innovative storage research can be fostered by making it easy to modify or extend the host-device interface.

Currently, the NVMe interface is the most preferred host-device interface due to its flexibility and extendibility. The NVMe protocol itself is flexible and easy to extend; however, applying any changes to an actual system is an entirely different matter. Specifically, the NVMe interface inherently involves a protocol defined at the hardware level. To extend the interface for a new device feature, the developer should incorporate the changes not only to the device driver on the host but also to the firmware or controller logic on the real devices. This level of work usually demands a huge amount of engineering efforts and research resources, restricting the research for novel storage devices.

This motivated us to build a storage emulator that provides a comprehensive way to customize the NVMe interface and support various storage device types on top. To this end, we attempt to virtualize devices from the PCI level so that they can behave like real physical devices from the entire host's point of view. We argue this is crucial for a storage emulator that should support various device types and advanced storage configurations, such as KVSSDs with custom operations, NVM-based ultra-low-latency SSDs, the target for NVMe-oF offloading, direct access from user-space bypassing the kernel, peer-to-peer data transfer between PCIe devices, and so on. We emphasize that this is the point of difference between NVMeVirt and previous work.

## 3.2 Virtualizing a PCIe/NVMe Device

To help understand the challenges in virtualizing NVMe devices, we first detail how PCIe/NVMe devices interact with the host [42]. As shown in Figure 1, the CPU and memory subsystem are connected to peripheral devices through a hardware component called *PCIe root complex*. The root complex generates PCI transactions to the devices on behalf of the CPU when the CPU accesses the device memory-mapped regions. The root complex has multiple PCIe ports, each of which can be connected to a PCIe device (i.e., PCIe endpoint) or a PCIe switch. The PCIe switch allows the hierarchical organization of PCIe devices by implementing a *PCIe bus*, through which multiple devices can be multiplexed.

PCIe devices, including NVMe devices, essentially communicate with the host operating system (and the host firmware) through a memory-mapped region for their initialization. A PCIe device is supposed to present its PCI *configuration header* in the PCI configuration address space. The configuration header contains essential information to initialize the device. This information includes the device ID, vendor ID, type code of the device, status of the device, and list of resources that the device provides. The host, specifically the root complex device driver, scans the PCI configuration address space to find the configuration headers presented by installed devices. For each detected configuration header, its corresponding device driver is invoked according to its device type and IDs. This process is called a PCI bus scan. To facilitate device-specific requirements during the PCI scan, the PCI subsystem in the Linux kernel allows customizing the operations for accessing the configuration header.

With this PCI initialization protocol, the most obvious way of creating a virtual PCIe device might be injecting a forged PCI configuration header into the root complex driver. However, in this case, when the root complex recognizes the PCIe device, it will attempt to directly communicate with the device at the hardware level. This inevitably leads to a system design that requires a hardware modification, which is impractical and even too intrusive for commodity servers.

To circumvent this pitfall, we virtualize PCIe devices *indirectly* through the PCI bus. First, NVMeVirt allocates a part of the reserved memory region for the PCI configuration header of the virtual device. The configuration header is set to indicate an NVMe device with required PCI capabilities (the "PCIe Device Emulator" part in Figure 1). With the configuration header, NVMeVirt creates a virtual PCIe bus with a non-existing PCI bus ID of the system (the ID is provided as a configuration parameter) and asks the PCI subsystem to scan the bus with custom configuration header operations. When the PCI subsystem performs the PCI bus scan, it effectively detects an NVMe-type device. When the subsystem attempts to initialize it by accessing the configuration header, NVMeVirt hooks in through the custom configuration header operations and emulates requested operations. This effectively presents an NVMe-type PCIe device to the PCI subsystem, making it ask the NVMe layer to initialize the device.

The NVMe device emulation is implemented on top of the PCIe device emulation. According to the NVMe specifications, NVMe devices should present their NVMe control

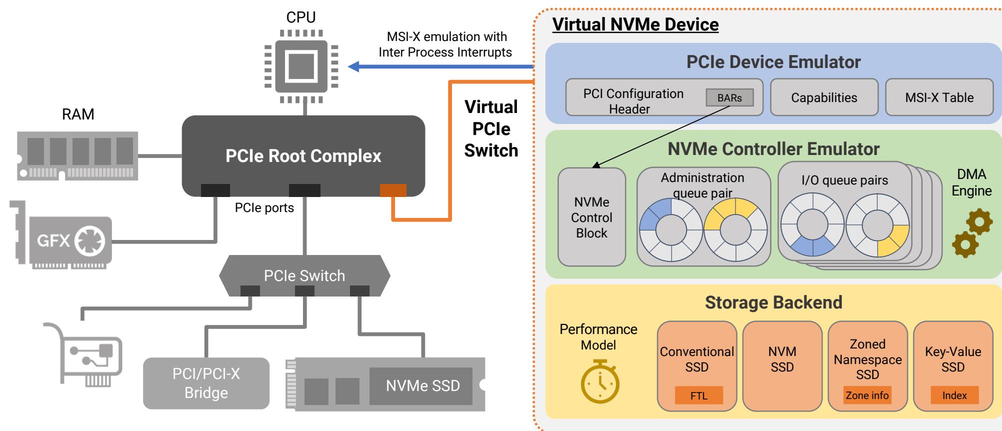


Figure 1: The overall architecture of NVMeVirt. NVMeVirt virtualizes a virtual NVMe device through the PCI bus and switch, so the device is seen as a native PCIe device from the host.

block through the base address register (BAR) fields in the PCI configuration header. Accordingly, NVMeVirt sets up the PCI configuration header so that the BARs point to a reserved memory region used for the control block. The NVMe layer on the host identifies the control block and operates on it according to NVMe standards.

In real devices, accesses to the NVMe control block are delivered to the device in the form of PCI transactions, initiating an action from the device. However, as the control block of NVMeVirt devices is only a memory region, accesses to it are processed silently without causing any event. To respond to the updates of the control block, we used the similar idea of vIOMMU [1]. Specifically, NVMeVirt runs a kernel thread called *dispatcher*. The dispatcher keeps checking the values of the control block to determine whether any changes have occurred. When the current value of the control block is changed since the last check, it implies that the host made some requests to the NVMe device. The dispatcher identifies the intention from the update and thus initiates the processing of the request.

We opt for the busy-waiting approach (i.e., keep scanning targets) over an event-driven approach (i.e., signal the dispatcher in response to incoming requests) to provide the low latency of NVM-based storage devices. The event-driven approach might save CPU cycles much; however, waking up the sleeping thread incurs non-negligible time overhead, making it unable to meet the demand for high I/O processing performance of modern storage devices.

Due to the emulation from the PCI layer level, NVMeVirt provides unique capabilities and opportunities that other emulators cannot provide. First, the emulated device operates like a real device from the perspective of the rest of the OS and even other devices. Any entity can instruct the NVMeVirt instance to perform any NVMe operations provided that it can set the control block and/or place operations in the NVMe queues under PCI and NVMe specifications. This makes it possible for a user-level application to directly access the device bypassing the kernel with SPDK [54]. In addition, even other PCI devices can place NVMe messages to the NVMeVirt instance according to the PCI peer-to-peer DMA protocol. This permits NVMeVirt to foster the studies using the NVMe-oF target offloading [9] and the direct communication between GPU and storage for AI applications [3]. Note that none of the previously presented simulators and emulators relying on device drivers or virtual machines can support these advanced storage use cases.

Another unique capability of NVMeVirt is that it allows the inspection of the NVMe message queues in detail. With real devices, it is infeasible to track the exact number of I/O requests queued in submission and completion queues from the host side since the device does not expose the processing progress to the host (i.e., the device is not supposed to report individual processing progress but only notify completions in bulk). As the dispatcher directly accesses the NVMe queue pairs and doorbells, we can track the exact state (i.e., queue depth of queue pairs, queuing delay, etc) of the device in software, allowing an in-depth understanding of the communication characteristics and behaviors. In addition, we can easily configure the maximum number of queue pairs by changing the tunable parameter, which enables the opportunity to study the implication of multi-queues on various configurations.

## 3.3 Supporting Various NVMe Device Types

While the dispatcher focuses on processing *the control requests* to the device, time-consuming *I/O requests* are handled by a set of kernel threads called *I/O workers*. As illustrated in Figure 2, each I/O worker maintains an *I/O process queue*, which lists the pending NVMe requests. When the dispatcher detects a doorbell ringing, it fetches the I/O requests from the corresponding submission queue. The dispatcher estimates

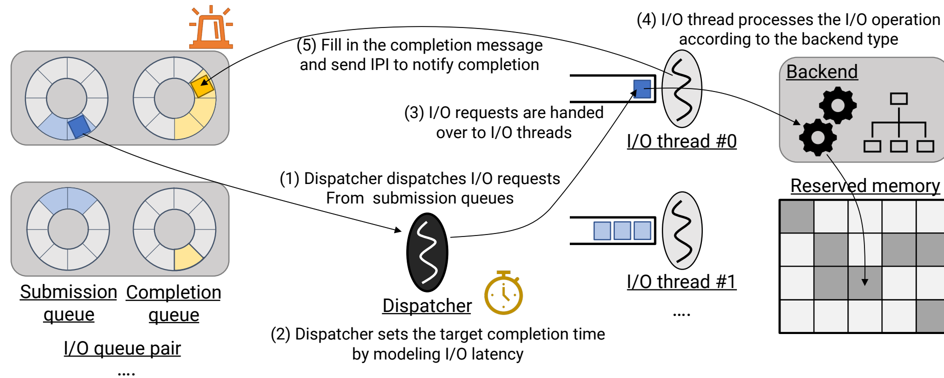


Figure 2: Processing I/O requests in NVMeVirt. The host inserts an I/O request into the NVMe submission queue and rings the associated doorbell. The dispatcher in NVMeVirt dispatches the I/O request (1), and computes the target completion time according to the latency model of the deployed backend (2). Then the request is handed over to an I/O worker, which processes the operation of the configured backend type (3)(4). When the desired target completion time is passed, NVMeVirt inserts a completion message into the NVMe completion queue and signals an Inter-Processor Interrupt (IPI) to the core that made the I/O request (5). Finally, the host I/O stack eventually wakes up the context that waits for the completion of the I/O request.

the target completion time of each I/O request (Section 3.4 discusses the timing model) and hands over the request to an I/O worker by placing it in the corresponding I/O process queue. The target I/O worker can be selected in a round-robin manner or as desired, and requests are placed in the queue sorted by their completion time.

The I/O worker processes the requests depending on the device type that it emulates. Currently, NVMeVirt supports the conventional SSD, NVM SSD, ZNS SSD, and KVSSD, and the device type can be specified at compile time. NVMeVirt initializes the NVMe control block to advertise itself as the selected device type, making the corresponding host device driver interact with the NVMeVirt instance. To process an I/O operation for a device type, the I/O worker invokes the I/O processing routine of the corresponding *backend* of the device type. For example, the conventional SSD backend copies the data payload to the backend memory for write requests or copies data from the memory to a specified I/O buffer. The ZNS backend checks whether the request is valid according to the ZNS specification and processes it similarly to the SSD backend if the request is valid. The KVSSD backend looks up the requested key from the index and stores or retrieves the value of the requested key. The details of the data handling in the backend memory will be discussed in Section 3.5.

The data is copied from/to the backend memory using the Intel I/OAT DMA engine [15] instead of the traditional `memcpy` for improving the I/O processing performance and reducing the CPU overhead. When an application initiates an I/O request, the requested data is on some pages in the host (i.e., in the I/O buffer or in the backend memory for write and read requests, respectively). Thus, the CPU can copy data within the memory of the host at a low overhead. However, the overhead might be non-negligible when data is on the device memory for P2P I/O requests. To support the inter-device communication, the PCI root complex and MMU collaborate to present an illusion of device memory in the physical address space of the host. When a PCI device moves the data in the device memory through DMA, the PCI root complex routes the accesses to the target device at the PCI level, providing low latency for data moving. However, when the CPU accesses the device memory-mapped address range for `memcpy`, the accesses are translated into PCI transactions and processed by the PCI device at a small I/O unit. This inevitably and significantly impairs the data copy performance, making NVMeVirt unable to guarantee the high performance of real devices. As the DMA engine allows low-overhead data transfer from/to device memory-mapped memory regions, NVMeVirt can achieve compelling I/O processing rates regardless of the I/O configurations.

After performing the actual I/O operation, the I/O worker compares the current and target completion times of the request. If the current time passes the target completion time, the I/O worker places a completion message into the corresponding completion queue. To notify the host OS of the processing completion, NVMeVirt sends an Inter-Process Interrupt (IPI) to the waiting processor bound to the queue pair. NVMeVirt supports Message Signaled Interface-X (MSI-X) for a scalable high-performance completion path. Each queue pair has its own dedicated IRQ vector; hence, NVMeVirt can efficiently signal to the target core with the specified IRQ vector. The I/O stack on the signaled core will eventually wake up the user context that initiated the request.

## 3.4 Performance Model

For a device emulator, the capability to imitate the performance of real devices is important. To that end, there has

been no shortage of studies attempting to replicate the latency and bandwidth characteristics of real storage devices with emulators and simulators [10,30,32,35,36,49,55]. We employed a similar approach as implemented in those works. Basically, an I/O request is divided into smaller chunks, which are independently processed in parallel by multiple data processing units. The data processing unit drives underlying storage media to read from or write to it. The chunk and data processing operation are, for example, considered the flash page and its read operation and programming in SSDs, respectively. The I/O completion time for an I/O request is determined by the completion time of the last operation for the request. The time for processing each chunk can be controlled with tunable parameters, which can be set based on the I/O bandwidth and latency of the target device. Further, the size of the chunk and the number of data processing units are configurable. With a set of parameter values we can expect a latency for small requests and the maximum bandwidth with large requests for the device. The latency is mainly determined by the operation time of the data processing operations, whereas the maximum bandwidth is bounded by the aggregated performance of the data processing units. For the remainder of the paper, we refer to these as the *target latency* and *target bandwidth*. For example, the OptaneDC SSD could be successfully modeled as the SSD that has a target latency of 12 us and a target bandwidth of 2,400 MiB for read requests. We refer to this model as the *simple model*.

We can produce the performance characteristics of real NVM SSD and KVSSD using the simple model as presented in Section 4.3. In general, the most complicated performance characteristics of flash-based storage devices are originated from garbage collection. However, as many studies have analyzed earlier [17,51,53], NVM SSDs are expected to allow in-place updates, enabling them to operate without garbage collection. For KVSSD, the size of key-value pairs is small; hence its performance tends to be bound by the host-device interfacing performance and the key indexing time, rather than the performance of the storage media. Thus, the simple model was sufficient for those device types.

Meanwhile, the model for conventional SSD is much more complex. FTL (Flash Translation Layer) controls data placement and performs garbage collection if necessary. In addition, modern SSDs aggressively use parallel processing techniques to maximize their performance. Accordingly, we had to redesign the performance model from the ground up to mimic the performance characteristics of real SSDs.

First, we implemented a page-mapped FTL based on that of FEMU [32]. The FTL determines data placement on the fly, and performs garbage collection when the number of free blocks gets below a threshold. Currently, the FTL selects victim blocks according to the greedy policy. To consider the on-device buffer, it is complemented with a small amount of memory as the write buffer. A write request is signaled to be completed as soon as the time to copy the payload into the write buffer is modeled. The buffer is processed in the background later with flash page writes.

In addition to the FTL, we revamped the data processing model to incorporate the parallel architectures widely used in modern SSDs. The storage space of modern SSDs is often divided into several partitions, and each partition is handled by one FTL instance [25]. Therefore, in the advanced model called the *parallel model*, multiple FTL instances exist in one SSD, and the instances share one PCIe link connected to the host. Data transfer from/to the host are serialized through the PCIe link. However, the rest of the FTL operations can be processed in parallel. Each partition comprises multiple NAND channels, which are connected to multiple dies in turn. Again, data transfer through the NAND channel are serialized, but the dies can operate independently.

In this architecture, FTL orchestrates the dies and channels to process I/O requests. For example, FTL instructs multiple dies to perform a read operation. When the dies are ready to transfer data, FTL schedules data transfer from the dies to the PCIe link through the shared NAND channels. The performance of the device is calculated based on the parameters of the FTL and model. We can speculate the parameters from the device specification and/or by observing the performance behavior of the device, similar to those approaches proposed in the literature [25,45,53]. With the parallel model, we can tune the parameters to provide a certain target latency and bandwidth independently similar to the simple model.

The ZNS SSD backend uses the same data processing model as the conventional SSD. However, it does not need the FTL since the host explicitly controls data placement according to the ZNS SSD specifications.

Note that NVMeVirt actually does not limit the performance model. Indeed, any performance models can be implemented and integrated into NVMeVirt as required.

### 3.5 Data Storage and Handling

NVMeVirt should store requested data somewhere in the system, and retrieve them later for read requests. Similar to many other storage simulators and emulators, NVMeVirt stores the data in the main memory. As running as a kernel module, NVMeVirt cannot luxuriate in comfortable functions from user space, such as virtual memory. However, the memory management overhead should be low and consistent to emulate future-generation devices such as PRAM- and MRAM-based SSDs. Different device types require different memory management policies and mechanisms. We explain the approaches for each device type that NVMeVirt currently supports.

**Common.** Regardless of the device type, NVMeVirt requires an extensive amount of memory for data storage. NVMeVirt obtains the required memory by reserving a part of the physical address space with the booting parameter during the system initialization. We configured the NUMA setting not to

interleave memory, and the memory is reserved from a dedicated NUMA node where the NVMeVirt threads are pinned down. Therefore, the reserved memory is physically contiguous. At the beginning of the reserved memory area are the NVMe control block and PCI resources, such as the MSI-X table and PCI capabilities, followed by the bulk memory region used for storing data.

**SSD and ZNS SSD.** Basically the backends for SSD and ZNS SSD use a simple linear mapping for data placement. For a physical block/page number in the flash address space, its in-memory location is calculated by adding the starting address of the reserved memory region. The FTL for conventional SSDs maintains the logical-to-physical flash address space mapping on top of the linear mapping as in FEMU [32]. Once the target address is calculated from the block number, NVMeVirt moves data from/to the in-memory location. One might suggest reusing the RAM disk facility or using `alloc_pages()` or `vmalloc()` for allocating the data storing pages. In this case, however, we cannot control the location of pages from NUMA domains. Even if we can use `alloc_pages_node()` to specify the NUMA domain to allocate pages from, the time to process page allocation may vary significantly according to the status of the memory subsystem. When the system has free pages for a core in its per-process free page list, the page allocation can occur fast. However, if the list is empty, the page should be allocated by dividing a large memory chunk through the buddy system allocator, which is time-consuming. For these reasons, we opt to use the linear mapping scheme, rather than the RAM disk or the dynamic mapping scheme. Further, the ZNS SSD backend maintains the metadata to track the status of zones (i.e., open zone list and the write pointers within the open zones).

**KVSSD.** The page-level address mapping is sufficient for SSDs since the allocation unit is a fixed size, larger than (or equal to) the page size. However, generally most the keys and values in KVSSDs are much smaller than a single page (often tens to hundreds of bytes long), and their sizes are highly variable. This necessitates a proper memory management scheme to prevent/control the external fragmentation of the address space yet to provide a stable performance. Accordingly, we divide the first half of the reserved memory into 1 KiB chunks and the second half to 4 KiB ones. NVMeVirt maintains a bitmap to track the availability of each chunk.
KVSSD also requires an index for key-value pairs. For simplicity, we implemented it with a hash table. During the device initialization, we allocate a slice of memory and initialize it as a table. Each entry in the table contains the actual key and location of data as the chunk index. To process a key-value operation, the key is hashed with the Fowler-Noll-Vo hash function [38] to produce an integer index. This integer index is used as the index of the table, and the hash collision is handled with the linear probing scheme. Currently, the maximum size is set to 16-byte and 4 KiB for the key and value, respectively.

# 4 Evaluation

In this section, we provide the evaluation results to demonstrate the features and versatility of NVMeVirt. We aimed to discover the following with the evaluations;

1. What device types and their features does NVMeVirt provide?
2. How precisely can NVMeVirt emulate the performance of various storage devices, including off-the-shelf SSDs, NVM SSDs, ZNS SSDs, and key-value SSDs?
3. What type of advanced storage configurations can NVMeVirt support?
4. How can NVMeVirt contribute to storage-domain research?

## 4.1 Evaluation Setup

**Evaluation Environment.** To evaluate NVMeVirt, we used two identical servers with the same hardware and software configurations. Each server is equipped with two Intel Xeon Gold 6240 processors running at 2.60 GHz in a NUMA configuration. Each processor has 36 cores and 196 GiB of memory, which provides 72 cores and a total of 392 GiB of memory. The system is also equipped with commercial storage devices for performance comparison and analysis: Samsung 970 Pro SSD and Intel P4800X SSD based on the OptaneDC persistent memory technology. The devices are 512 GB and 350 GB in size, and represent an off-the-shelf high-end SSD and NVM SSD, respectively. We refer to them as "SSD" and "Optane" in the rest of the paper. To evaluate KVSSD, we used the Samsung KVSSD [23]. We also used a ZNS SSD provided by a company as an evaluation prototype. This ZNS SSD comprises 96 MiB zones that can be written only at 192 KiB units. The servers are also equipped with one Mellanox ConnectX-5 VPI HCA and connected through Mellanox SX6012 switch supporting 56 Gbps FDR bandwidth. We implemented NVMeVirt based on the Linux kernel 5.10.37, and the implementation takes approximately 9,000 lines of the kernel module code.

**Configuration.** To minimize the cross-interference between the applications and operations of NVMeVirt, we set them to run on different processors. Specifically, one processor (processor 1) is completely dedicated to NVMeVirt, whereas applications and benchmarks run on the other processor (processor 0). During the system initialization, the entire memory for the processor 1 is reserved with kernel booting parameters and used for storing data. The dispatcher and I/O workers are pinned down to different cores on the dedicated processor. The virtual PCIe bus is registered to the system as if it is attached to the processor 1. The memory for the NUMA

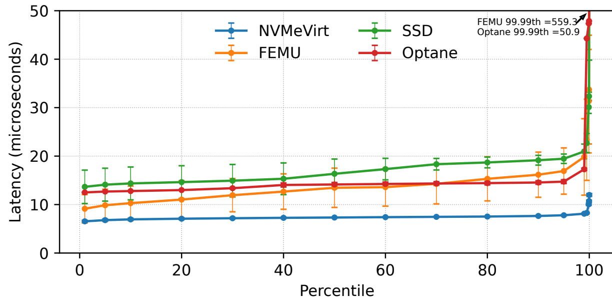


Figure 3: Performance variance of real and emulated devices for 4 KB random write operations

node 0, which is dedicated to applications, is configured to 32 GiB considering the size ratio between the memory and storage devices. In the evaluation, NVMeVirt is configured to use one I/O worker and a maximum of 72 queue pairs so that each core has its own queue pair. Note that we can easily change the maximum number of queue pairs of the virtual device.

## 4.2 Emulation Quality

NVMeVirt is primarily focusing on facilitating various NVMe device types, and FEMU [32] is the most relevant work sharing a similar goal. We compare the emulation quality by measuring the latency distribution of random write operations by repeating the same test 10 times. In each test, FIO writes 128 GiB of data with 4 KiB requests at random locations. We set FEMU and NVMeVirt to operate at the maximum speed without adding any latency to the incoming request. Figure 3 summarizes the percentile distributions of the 10 runs. The error bars indicate the standard deviation of the runs; thus, the longer the bars are, the more performance fluctuation the system exhibits.

Compared to the performance of real devices, NVMeVirt exhibits much lower latency with a very stable performance over the entire percentile range. These performance characteristics are very promising for emulating future storage devices. The FEMU-emulated device, however, barely meets the required performance to emulate modern storage devices. The maximum performance of FEMU is slightly faster than Optane. Hence, we are unable to utilize FEMU to project the performance implication of future low-latency storage devices. In addition, we observe high run-to-run performance variance from FEMU. Its standard deviations range from 28.7% to 39.7% of its average performance. The 99.99th percentile of FEMU goes off-the-chart, showing an average of 559.3 us and a standard deviation of 462 us. Considering the influences of the performance variance on the applications' tail latencies, FEMU will operate with a very high non-realistic tail latency.

Table 2: Presumed model parameters to emulate a real device performance.

| Simple model | Optane | KVSSD |
|---|---|---|
| Page size | 4 KiB | 4 KiB |
| # of I/O units | 1 | 10 |
| Read latency | 12 us | 154 us |
| Read bandwidth | 2.4 GiB | 2.6 GiB |
| Write latency | 14 us | 56 us |
| Write bandwidth | 2.0 GiB | 1.3 GiB |

| Parallel model | SSD | ZNS |
|---|---|---|
| PCIe link bandwidth | 3.6 GiB | 3.3 GiB |
| # of NAND channels | 8 | 8 |
| NAND channel bandwidth | 800 MiB | 800 MiB |
| Dies per channel | 2 | 16 |
| Read unit size | 32 KiB | 64 KiB |
| NAND read time | 36 us | 40 us |
| Write unit size | 32 KiB | 192 KiB |
| NAND write time | 185 us | 1,913 us |

## 4.3 Emulating a Real Device Performance

One of the primary goals of NVMeVirt is to emulate the performance of real devices. To verify this, we measured various performance metrics from real devices and configured NVMeVirt devices, as we discussed in 3.4. Table 2 summarizes the key parameters that we used for emulating the target devices. The values are obtained empirically from in-house microbenchmarks and device specification documents [16, 47, 50]. We used various benchmark tools and configurations for evaluating various device types. For Optane and the SSD, we used FIO [2] to measure the read and write latency while varying the request size from 4 KiB to 256 KiB. We also used FIO for measuring the performance of the ZNS SSD. We evaluated the read performance in the same manner. However, the ZNS SSD only allows 192 KiB writes to opened zones. Thus, we evaluated write performance with various numbers of threads that generate 192 KiB requests in accordance with the ZNS zone restriction. To evaluate KVSSD, we used OpenMPDK KVBench [46] which is an open-source benchmark based on the ForestDB benchmark suite [6]. We report the aggregated bandwidth from various payload sizes. In addition, we report the aggregated bandwidth measured with KVCeph [27] that generates realistic key-value operation workloads.

Figure 4 compares the performance of virtual devices to the real devices on various configurations. In each category, the values are normalized to the left-most entry value of the category (i.e., 4 KiB performance of the real device) Throughout the evaluation, we can confirm that the virtual devices provided by NVMeVirt faithfully reflect the configured target performance. We observe that the performance difference between the real and the virtualized devices is small. The

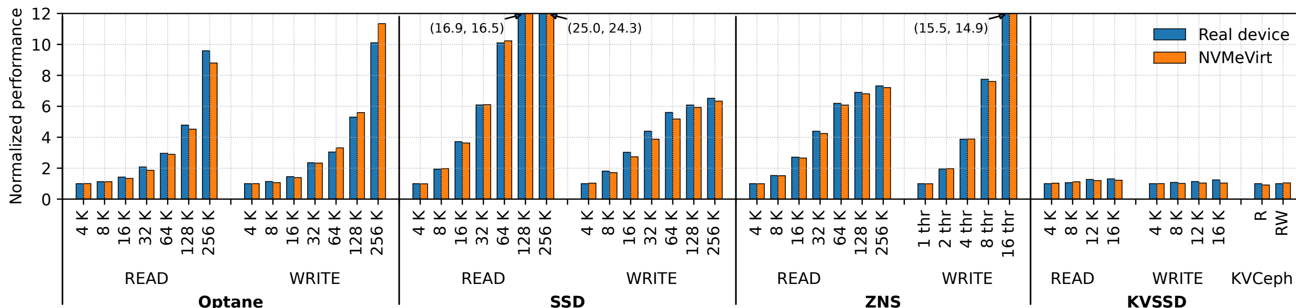

Figure 4: Performance comparison of the real devices and virtual devices. The performance is normalized to the left-most entry value of the category.

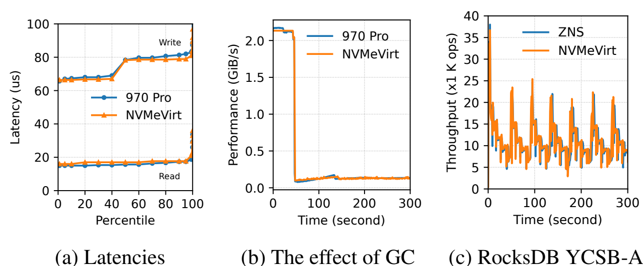

Figure 5: The comparison of various performance characteristics

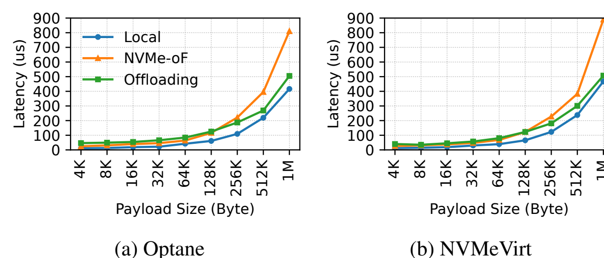

Figure 6: The write performance as the NVMe-oF target

performance difference is by up to 12.2%, 11.8%, 3.3%, and 3.8% for Optane, SSD, ZNS SSD, and KVSSD, respectively.

Next, we analyzed various I/O characteristics. Firstly, Figure 5a summarizes the latency distribution for processing 16 KiB requests in Samsung 970 Pro and its NVMeVirt counterpart. We can see the similar latency distributions between the setups. Secondly, we evaluated the performance change when the garbage collection is performing. We built a microbenchmark that fills in the storage space with sequential write and keeps performing random writes. These writes will eventually trigger GC, which will cause performance drops. Figure 5b shows the performance change, and we can see that NVMeVirt exhibits the realistic performance change when GC is involved. Lastly, Figure 5c shows the performance of RocksDB running on ZNS SSD. We measured the throughput over a period of time while running the YCSB-A benchmark. We can observe repeated performance changes from ZNS SSD, and NVMeVirt can model the performance changes very closely. From these evaluations, we conclude that NVMeVirt can model various performance aspects of the target devices.

### 4.4 Supporting Various Storage Environments

In this evaluation, we elaborate on the versatility of NVMeVirt in various storage configurations. First, we demonstrate the feasibility of the NVMe-oF target offloading. We configured an NVMeVirt instance as the NVMe-oF target and measured their performance with the FIO benchmark. The NVMeVirt instance is configured to emulate Optane with the target performance listed in Table 2. Figure 6 compares their performance under various payload sizes and different NVMe-oF configurations. Note that 'NVMe-oF' indicates the performance of the default NVMe-oF configuration without the target offloading, and 'Offloading' is with the target offloading enabled. We present the results from write operations only since read operations showed the same trend.

We can observe that NVMeVirt emulates the performance of Optane over NVMe-oF closely. Specifically, the baseline configuration without the target offloading outperforms that with the target offloading when the payload is smaller than 128 KiB. We attribute the performance trend to the NVMe command processing overhead in the adapter that outweighs the performance gain from the optimized data path. However, when the payload is larger than 128 KiB, the performance gain outweighs the overhead, making the target offloading-enabled configuration show much lower latency.

We also demonstrate the PCI peer-to-peer DMA support for AI workloads. Specifically, GPUDirect Storage (GDS) is one of the promising techniques to accelerate GPU-intensive workloads [39, 40]. GPU directly accesses NVMe devices through PCI peer-to-peer DMA protocol, enabling decreased latency and CPU utilization on the host. We analyze the im-

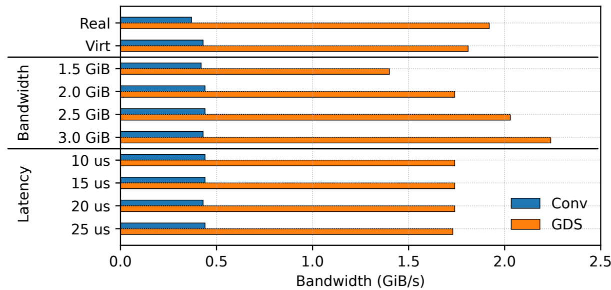


Figure 7: Checkpointing performance of Megatron DeepSpeed.

plication of storage performance on the GDS environment with NVMeVirt.

We measure the performance of checkpointing while running Megatron DeepSpeed [37] with NVIDIA A100 GPU. As shown in Figure 7, with Samsung 970 Pro SSD, the checkpointing occurs at the rate of 0.37 GiB/s with the conventional storage configuration, where checkpointing data is stored through the host's filesystem (the data is labeled with 'Real'). With GDS enabled, the checkpointing data goes to the NVMe device directly, showing a substantial performance gain of 5.2x. The NVMeVirt instance configured for the SSD exhibits the same performance trend, as labeled with 'Virt'.

We set up an NVMeVirt SSD instance in the same environment and evaluate the implication of storage device performance. We measured the checkpointing performance on various target bandwidths, but with a fixed latency of 10 us (grouped in 'Bandwidth'). We also evaluated the performance from various latencies while the target bandwidth is fixed to 2.0 GiB (grouped in 'Latency').

As shown in Figure 7, the storage performance does not much influence the AI application when the checkpointing occurs through the conventional storage configuration. The checkpointing performance remains consistent at a rate of 0.43 GiB/s regardless of changes in bandwidth and latency. However, the performance is affected by bandwidth in the GDS configuration. This confirms that the direct storage access is promising in that it can circumvent the inherent performance bottleneck in the conventional I/O path. Also, it implies that to fully exploit the reduced overhead through GDS, the storage performance should be improved further. From the consistent performance over a wide range of latencies, we can infer that the workload is likely to process I/O with a high queue depth.

## 4.5 Case for the Database Engine Analysis

To promote the tunable performance of NVMeVirt, we analyze the implication of storage performance on database engine performance. We selected MariaDB and PostgreSQL, two database engines that are very popular in the industry [7].

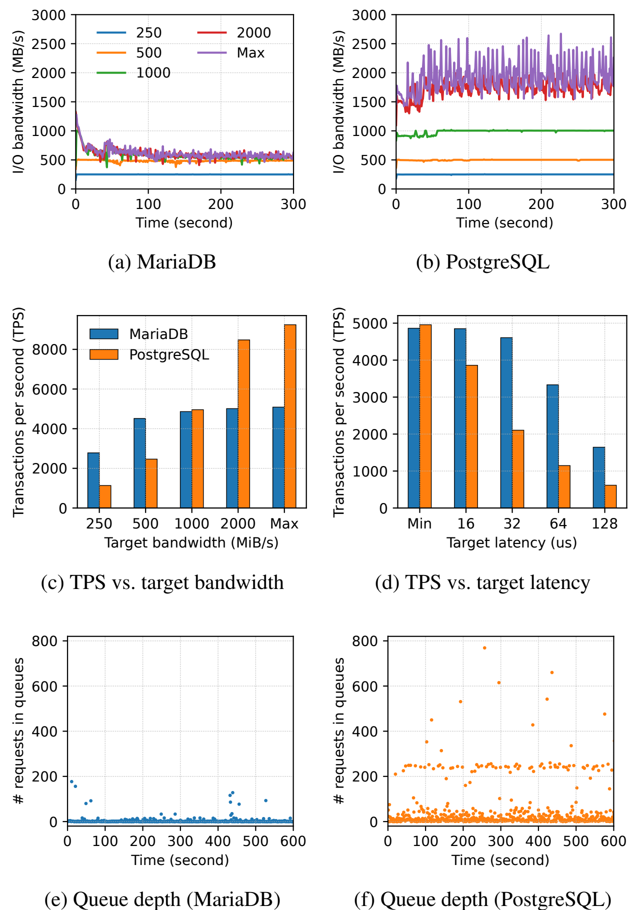


Figure 8: Performance characteristics of the MariaDB and PostgreSQL database engines on various storage configurations

We created an NVMeVirt instance configured as an NVM SSD, and configured the database instance on it with recommended configurations from optimization tools [13, 43]. The database instance is then populated with sysbench [48] to have 10 tables of 50,000,000 bytes in size, taking approximately 120 GiB of space in total. Then we run the OLTP workload with sysbench with 72 threads for 60 minutes. We measured various performance metrics while running the benchmark, and Figure 8 summarizes the results. We only report the trends of the first 5 minutes since the performance became stable afterwards.

Overall, we verified that MariaDB and PostgreSQL react very differently to the storage performance. Figure 8a and 8b compare the I/O bandwidth utilization of MariaDB and PostgreSQL over time when the target bandwidth is set to the given value (i.e., 250 implies the storage bandwidth is limited to 250 MiB/s). In this evaluation, the I/O latency was set to a minimum (i.e., does not impose any delay while processing I/O operations). For MariaDB, it fully utilizes the I/O

bandwidth up to 500 MiB/s, but does not utilize it further. Even though the storage device provides a higher bandwidth, the I/O bandwidth utilization remains low, approximately at 600 MiB/s. On the other hand, PostgreSQL fully utilizes the I/O bandwidth up to 1,000 MiB/s and exhibits a saturated performance at around 1,800 MiB/s. This hints that PostgreSQL is designed to utilize the storage device more eagerly than MariaDB. However, this does not necessarily mean that PostgreSQL outperforms MariaDB. Figure 8c compares the processing performance measured in transactions per second (TPS) on various bandwidth limits. The I/O bandwidth limit influences both database engines, but PostgreSQL is much more sensitive than MariaDB. Specifically, MariaDB exhibits a higher TPS than PostgreSQL when the bandwidth is low, but the TPS is not improved much when the device has more bandwidth. Meanwhile, PostgreSQL shows a lower performance when the bandwidth is low. However, the performance improves as the device supports more bandwidth. When the bandwidth limit is low, MariaDB outperforms PostgreSQL by 2.45x at 250 MiB/s bandwidth limit. However, PostgreSQL outperforms MariaDB by 1.82x at the unlimited bandwidth.

The engines exhibit a similar trend with respect to the latency. Figure 8d compares the performance when the bandwidth is fixed to 1,000 MiB/s, and both read and write latencies are set to the values on the y-axis. When the device exhibits a high latency, MariaDB outperforms PostgreSQL by up to 2.67x when the latency is 128 us. However, as the latency decreases, the TPS of PostgreSQL keeps increasing, becoming comparable to that of MariaDB when the latency is minimum. Figure 8e and 8f shows the number of queued requests in the submission queues of the device over time. The device is configured to have the minimum latency and a target bandwidth of 2.0 GiB. MariaDB operates with a low queue depth, whereas PostgreSQL utilizes a higher queue depth with a noticeable unique pattern near qd=200.

From the evaluation, we can conclude that PostgreSQL is more promising on modern storage devices, whereas MariaDB is more efficient when the storage is slow. We can verify that NVMeVirt allows us to estimate the performance of applications on future storage devices.

### 4.6 Case for NVMe Interface Study

As NVMeVirt handles inbound NVMe operations in software, it opens up the opportunity to extend the host-device interface easily. To demonstrate this, we made a case with one of the recent studies whose evaluation is limited by the host-device interface modification. Specifically, Kim et al. [26] proposed to extend the NVMe command set so that one NVMe command can batch multiple key-value operations, thereby amortizing the interface overhead for small key-value operations. To realize this so-called 'compound command' concept, the KVSSD firmware should be modified to understand and process the extended NVMe command. However, the authors were unable to modify the firmware and ended up estimating the performance gain from the single operation performance.

We attempted to verify the benefit of the compound command by using the KVSSD instance with NVMeVirt. Specifically, we modified the KVSSD backend to understand the compound command and process packed operations in a batch. Each I/O operation in a compound command is processed as an individual key-value operation in the backend. This modification took less than a week for one of the authors, and we argue that this manifests the advantage of the software-level NVMe abstraction that NVMeVirt provides. To evaluate the performance, we built a user-level microbenchmark tool that builds the compound command with multiple requests, and submits the command to the device through the NVMe pass-through interface. Note that the extended KVSSD backend uses the same performance model and configuration as explained in Section 4.3.

From the evaluation, we can verify the significant performance improvement with the compound command. Without the compound command, processing eight 4 KiB key-value put operations takes approximately 469 us, which is reduced to 86 us with the compound command, giving a 5.4x performance gain. This improvement is higher than the value reported in the original work (92.0 us to 41.5 us), and we attribute the extra improvement to the conservative estimation in the original work.

## 5 Conclusion

We presented NVMeVirt, a virtual, software-only NVMe device. It opens a new opportunity for developers to co-design highly intelligent storage devices over the NVMe interface. With NVMeVirt, we demonstrated the usefulness of NVMeVirt for storage research.

Currently, NVMeVirt only supports a single instance of one device type. We are planning to support multiple NVMe device instances with various storage device types. Furthermore, we are working on a separate study to present the in-depth methodology and analysis to model the performance characteristics of various NVMe-based SSDs using NVMeVirt.

## Acknowledgments

We would like to thank our shepherd, Robert Ross, and anonymous reviewers for their invaluable feedback. We also thank Hyeong-Jun Kim who developed the initial prototype of NVMeVirt. This work was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (23ZS1310), the National Research Foundation of Korea (NRF) grant (No. 2019R1A2C2089773), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363) funded by the Korea government (MSIT).

## References


[1] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: Efficient IOMMU emulation. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, Portland, OR, June 2011.

[2] Jens Axboe. fio: flexible I/O tester. https://github.com/axboe/fio.

[3] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. FlashNeuron: SSD-enabled large-batch training of very deep neural networks. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, February 2021.

[4] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: avoiding the block interface tax for flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, Virtual, July 2021.

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 359–375, Santa Clara, California, USA, February–March 2017.

[6] Couchbase Labs. ForestDB benchmark. https://github.com/couchbaselabs/ForestDB-Benchmark.

[7] DB-Engines ranking. https://db-engines.com/en/ranking.

[8] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstien, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, M. G. França, and Vladimir Alves. Cost-effective, energy-efficient, and scalable storage computing for large-scale AI applications. *ACM Transactions on Storage*, 16(4):1–37, 2020.

[9] NVM Express. NVMe-oF specification. https://nvmexpress.org/developers/nvme-of-specification/.

[10] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber: Enabling precise full-system simulation with detailed modeling of all SSD resources. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481, Fukuoka, Japan, October 2018.

[11] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 153–165, Seoul, South Korea, June 2016.

[12] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: advanced zoned namespace interface for supporting in-storage zone compaction. In *Proceedings of the 15nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[13] Major Hayden. MySQLTuner. https://github.com/major/MySQLTuner-perl.

[14] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 173–187, Virtual, July 2020.

[15] Intel. Intel I/O acceleration technology. https://www.intel.co.kr/content/www/kr/ko/wireless-network/accel-technology.html.

[16] Intel. Intel Optane SSD 9 series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[17] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module.

[18] Mike Jackson and Ravi Budruk. *PCI Express Technology*. MindShare Technology, 2012.

[19] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384, Austin, TX, USA, February 2017. IEEE.

[20] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, June 2018.

[21] Myoungsoo Jung. OpenExpress: Fully hardware automated open research framework for future fast nvme devices. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, Virtual, July 2020.

[22] Myoungsoo Jung, Jie Zhang, Ahmed Abulila, Miryeong Kwon, Narges Shahidi, John Shalf, Nam Sung Kim, and Mahmut Kandemir. SimpleSSD: Modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters*, 17:37–41, September 2017.

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

[24] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Denver, CO, June 2016.

[25] Joonsung Kim, Kanghyun Choi, Wonsik Lee, and Jangwoo Kim. Performance modeling and practical use cases for black-box SSDs. *ACM Transactions on Storage*, 17(2), jun 2021.

[26] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction support using compound commands in key-value SSDs. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Renton, WA, July 2019.

[27] Open memory platform development kit. http://github.com/OpenMPDK.

[28] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage*, 16(3), aug 2020.

[29] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/software co-programmable framework for computational SSDs to accelerate deep learning service on large-scale graphs. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2022.

[30] Parallel Data Lab. The DiskSim simulation environment v4.0. https://www.pdl.cmu.edu/DiskSim/index.shtml.

[31] Young-Sik Lee, Luis Cavazos Quero, Sang-Hoon Kim, Jin-Soo Kim, and Seungryoul Maeng. ActiveSort: Efficient external sorting using active SSDs in the MapReduce framework. *Future Generation Computer Systems*, 65:76–89, December 2016.

[32] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The case of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[33] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[34] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, Santa Clara, California, USA, February 2013.

[35] Krishna T. Malladi, Manu Awasthi, and Hongzhong Zheng. FlexDrive: A framework to explore NVMe storage solutions. In *Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications*, page 1115–1122, Dec 2016.

[36] Krishna T. Malladi, Mu-Tien Chang, Dimin Niu, and Hongzhong Zheng. FlashStorageSim: Performance modeling for SSD architectures. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS)*, page 1–2, August 2017.

[37] Microsoft. Megatron-DeepSpeed. https://github.com/microsoft/Megatron-DeepSpeed.

[38] Landon Curt Noll. FNV hash. http://isthe.com/chongo/tech/comp/fnv/.

[39] NVIDIA. GPUDirect Storage: A direct path between storage and GPU memory. https://developer.nvidia.com/blog/gpudirect-storage/.

[40] NVIDIA. The NVIDIA Magnum IO GPUDIrect Storage overview guide. https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html.

[41] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th ACM International Conference*

*on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, Utah, USA, March 2014.

[42] PCI SIG. PCI SIG. https://pcisig.com.

[43] PGTune: configuration for PostgreSQL based on the maximum performance for a given hardware configuration. https://pgtune.leopard.in.ua/.

[44] QMEU. QEMU: A generic and open source machine emulator and virtualizer. https://qemu.org.

[45] Manoj P. Saha, Adnan Maruf, Bryan S. Kim, and Janki Bhimani. KV-SSD: What is it good for? In *Proceedings of the 58th ACM/IEEE Annual Design Automation Conference (DAC)*, pages 1105–1110, 2021.

[46] Samsung Electronics. OpenMPDK KVSSD. https://github.com/OpenMPDK/KVSSD/.

[47] Samsung Electronics. Samsung Enterprise SSD. https://www.samsung.com/semiconductor/ssd/enterprise-ssd/.

[48] Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[49] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: a framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[50] Tech Power Up. Samsung 970 Pro 512 GB. https://www.techpowerup.com/ssd-specs/samsung-970-pro-512-gb.d54.

[51] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC '19, November 2019.

[52] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carolejean Wu, David Michael Brooks, and Guyeon Wei. RecSSD: Near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[53] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, June 2019.

[54] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: a development kit to build high performance storage applications. In *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[55] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual machine based SSD simulator. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST)*, May 2013.

[56] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, February 2012.