# Empowering Storage Systems Research with NVMeVirt: A Comprehensive NVMe Device Emulator

SANG-HOON KIM, Ajou University, South Korea
JAEHOON SHIM, EUIDONG LEE, SEONGYEOP JEONG, ILKUEON KANG, and
JIN-SOO KIM, Seoul National University, South Korea

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. Furthermore, existing emulators lack the capability to support the advanced storage configurations that are currently in the spotlight.

In this article, 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.

CCS Concepts: • **Information systems** → **Storage management**; • **Software and its engineering** → **Operating systems**; • **Hardware** → *Simulation and emulation;*

Additional Key Words and Phrases: Virtual device, emulator, SSD, key-value SSD, ZNS SSD, NVMe device

**ACM Reference format:**
Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. 2023. Empowering Storage Systems Research with NVMeVirt: A Comprehensive NVMe Device Emulator. *ACM Trans. Storage* 19, 4, Article 32 (October 2023), 26 pages.
https://doi.org/10.1145/3625006

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).
Authors' addresses: S.-H. Kim, Ajou University, Suwon, South Korea; e-mail: sanghoonkim@ajou.ac.kr; J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, Seoul National University, Seoul, South Korea; e-mails: {mattjs, led5198, seongyeop.jeong, kangilkueon, jinsoo.kim}@snu.ac.kr.

1553-3077/2023/10-ART32 $15.00
https://doi.org/10.1145/3625006

## 1 INTRODUCTION

NAND flash memory has been significantly gaining in popularity for consumer devices and enterprise servers, and the fast advancement of semiconductor technologies has fostered the **non-volatile memory (NVM)** in building high-density, low-latency storage devices. Nowadays, we can purchase off-the-shelf storage devices that feature tens of microseconds latency and several GiB/s of bandwidth [10, 20].

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, 38, 45], **zoned namespace SSD (ZNS SSD)** [4, 15], **key-value SSD (KVSSD)** [18, 23, 27, 50], and computational storage [9, 14, 24, 33, 35, 37, 56]. 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 and efficient 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 deployed in complicated modern storage environments. 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 the NVMe interface. 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 [15, 36, 39, 59]. These approaches, however, fail to present a suitable NVMe device instance that is fully functional in diverse modern storage environments such as when the kernel is being bypassed [28, 58] or when a device directly accesses storage devices through NVMe-over-fabrics or PCI peer-to-peer communication [3, 11, 46]. Table 1 compares the previous approaches and their limitations.

This article 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 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 for supporting new device types. Unlike other emulators with similar goals, NVMeVirt does not rely on the virtualization technology, allowing comprehensive communication models

Table 1. Comparisons of Various Virtual Storage Device Approaches

| | Simulators | | Emulators | | | | |
|---|---|---|---|---|---|---|---|
| | **Trace Driven [34, 40]** | **Full System [13, 26, 53]** | **VM Based [15, 36, 59]** | **Block-driver Level [60]** | **NVMe-driver Level [39]** | **HW Platforms [25, 32]** | **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 | Yes | 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** |

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 empower 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 article:

- Provide a software framework to facilitate NVMe device research with various and stable I/O characteristics.
- Facilitate 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 article 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 article.

## 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 [22, 46]. 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, 15] and key-value SSDs [18, 23, 27, 50].

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, and so forth. 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 [11]. If the network adapter supports the target offloading feature, the NVMe commands can be processed completely at the hardware level without any involvement of the 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 is laid out in the physical PCIe bus address space, containing 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 shut down 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). 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. Usually, each submission queue is dedicated to a CPU core to deliver scalable performance by eliminating contentions among cores. Each queue has the associated *doorbell*, which indicates the index of the latest request in the queue. When the host driver writes a new value to 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 writes an NVMe completion descriptor 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, 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 submission and completion queues can be created, modified, or destroyed by submitting requests to the special queue called the *administration queue*. The administration queue pair is initialized during device initialization by specifying its physical address in the NVMe control block.

The host can ask the device to create regular queue pairs by writing a queue creation descriptor 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

Myriad studies have attempted to imitate real storage devices in software [13, 15, 25, 26, 32, 34, 36, 39, 40, 53, 59, 60]. 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 [13, 26, 34, 40, 53]. 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 [15, 25, 32, 36, 39, 59, 60]. FlexDrive [39] 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, and so forth. 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 [36] proposes an accurate and scalable virtual NVMe device using host virtualization technologies. Specifically, FEMU provides guest operating systems with a virtual NVMe device by leveraging the device virtualization feature of the QEMU [48]. 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. This requires 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 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.

We found several previous works aiming to achieve the same goal. As summarized in Table 1, however, they lack the capabilities to support complicated I/O models required by modern storage configurations such as NVMe-oF target offloading, direct access from user-space bypassing the kernel, peer-to-peer data transfer among PCIe devices, and so on. These I/O models tend to access the storage device directly bypassing the I/O stack on the operating system to minimize the I/O overhead. Therefore, emulators working at the operating system level can support none of these I/O models. Emulators leveraging the host virtualization technology can support some I/O models where I/O requests are initiated in the virtual machine. However, they cannot support the I/O models in which hardware devices directly initiate the I/O.

From the observation, we attempt to make a virtual device 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, and 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 [46]. 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. When the root complex pushes the PCIe transactions to the switch, it routes the PCIe transactions to the target device on the bus. The PCIe transactions can be also forwarded in the opposite direction, from the device to the root complex through the bus.

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.

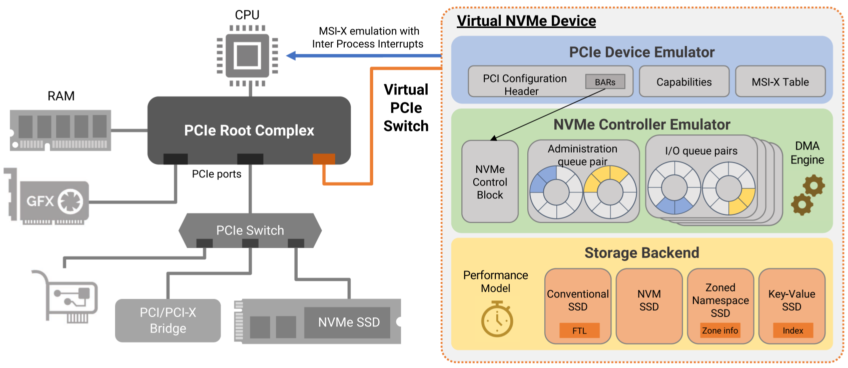


Fig. 1. The overall architecture of NVMeVirt. NVMeVirt makes a virtual NVMe device through the PCI bus and switch, so the device is seen as a native PCIe device from the host.

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 make virtual PCIe devices *indirectly* through the PCI bus. Basically, NVMeVirt tricks the OS to interpret a region of host memory as a PCIe configuration header, emulating device registers. To this end, 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 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 a 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 write 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 [58]. In addition, even other PCI devices can write NVMe commands to the NVMeVirt instance according to the PCI peer-to-peer DMA protocol. This permits NVMeVirt to foster the studies using complicated storage configurations, such as the NVMe-oF target offloading [11] 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 configuration 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 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. It also responds to zone-related requests by referring to the zone information it maintains. The KVSSD backend looks up the requested key from the index and stores or retrieves the value

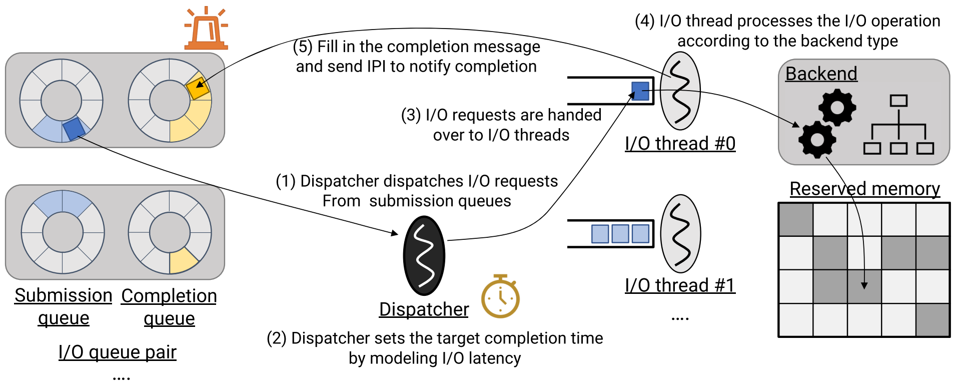

(5) Fill in the completion message
and send IPI to notify completion
(4) I/O thread processes the I/O operation
according to the backend type
Backend
(3) I/O requests are handed
over to I/O threads
I/O thread #0
(1) Dispatcher dispatches I/O requests
From submission queues
Reserved memory
I/O thread #1
....
Submission queue
Completion queue
I/O queue pair
....
Dispatcher
(2) Dispatcher sets the target completion time
by modeling I/O latency

Fig. 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) and (4)). When the desired target completion time is passed, NVMeVirt inserts a completion descriptor 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.

of the requested key. The details of the data handling in the backend memory will be discussed in Section 3.6.

The data is copied from/to the backend memory using the Intel I/OAT DMA engine [19] 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 data copy overhead becomes non-negligible when data is on the device memory for PCI 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 writes a completion descriptor on 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. If the target completion time has not been reached, the I/O worker does not generate the completion signal for the request at that time. Instead, the I/O worker goes back to check the I/O process queue to handle newly arrived requests. After processing newly arrived requests (processing skipped if no new request has arrived), the I/O worker re-examines pending requests to identify those that have exceeded their completion times. If there is a completed request, the I/O worker generates the completion signal for it.

## 3.4 Simple 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 [13, 34, 36, 39, 40, 53, 59]. 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 article, we refer to these as the *target latency* and *target bandwidth*. For example, the OptaneDC SSD can be modeled as the NVM SSD that has a target latency of 12 $\mu$s 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 [21, 55, 57], 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.

## 3.5 Advanced Performance Model

*3.5.1 Design Considerations.* The performance model for conventional SSDs is much more complex than NVM SSDs since they often incorporate various techniques and mechanisms to achieve high performance while masking the unique characteristics of NAND flash memory. We studied the specifications and performance characteristics of recently released SSDs to list the key features for the modernized SSD performance model. Also, we surveyed recent FTL research papers to get useful insights behind the performance characteristics. To follow are the key aspects we considered during the design of the performance model for NVMeVirt.

**One-shot programming.** With the significant increase in data density achieved through 3D NAND and multi-level cell technologies, reliable programming has become a growing challenge. When multi-level cell technology is used, one physical flash page holds the data of multiple logical pages, each of which consists of corresponding bits of the multi-level cells on the physical page. Traditional FTLs used to process write requests with a page-by-page strategy: programming each logical page one after another. However, recent FTLs often minimize the programming overhead through the so-called "one-shot programming" scheme; the logical pages on the same physical page are first buffered and then programmed together with a single flash page programming operation. This scheme can improve the write performance as well as the write reliability significantly, and it seems many modern SSDs employ this write scheme [12, 17]. To the best knowledge of the authors, no previous storage emulator has considered this modernized write scheme.

**Write buffer.** The increased data density per cell has inevitably led to an increase in NAND program time. Many modern SSDs opt to buffer the write data before storing them in the flash memory to hide the long NAND program time. Also, the write buffer is one of the key enablers for the one-shot programming feature. However, we could not find an emulator that accurately models the write buffering.

**Channel/PCIe level contention.** In an SSD, a channel serves as a data pathway connecting the NAND chips to the SSD controller. The chips can operate independently; however, since multiple chips share a single channel, contention can occur in the channel. When there is contention, operations may take longer than the base transmission time, which is determined by the channel bandwidth. Similar to the channel, a PCIe link is a data passage between the SSD controller and the host. The sum of the bandwidth of channels may exceed the PCIe bandwidth, but the overall device performance is capped by the PCIe bandwidth. We found that simulators and hardware-based emulators can easily emulate the contention due to their internal organization. However, many software-based emulators often omit modeling the contention and just bound the device performance with the data processing time of the NAND chips. In this case, the performance is solely limited by the NAND chip's capabilities, offering significantly higher bandwidth than what the actual devices can deliver (see Section 4.2).

**Multiple FTL instances.** Modern SSDs split the channels into logical groups and allocate them to each FTL core as a separate FTL [29]. Specifically, a single I/O request can be split into smaller operations and distributed to multiple FTL instances, which process the operations in parallel. In this manner, SSDs can effectively utilize device-level parallelism and achieve improved scalability. However, it also introduces additional constraints during I/O completion and garbage collection.

*3.5.2 Performance Model.* We carefully designed an FTL performance model to incorporate the features obtained from the previous observations. Basically it models *a set of* page-mapped FTL instances. Each FTL instance has a number of channels, each of which is connected to a number of NAND flash chips. An I/O request is assigned to one or more FTL instances that cover the LBA range of the request.

To process a write request, the data should be buffered in a write buffer first. NVMeVirt assumes that the write buffer is shared by all FTL instances. When the write buffer is full, the processing of the write request is stalled until enough write buffer is secured to accommodate the data. To incorporate the one-shot programming in the model, NVMeVirt does not process the buffered data immediately. Instead, it waits until other logical pages on the same physical page are buffered. When all logical pages for the physical page are buffered, their data is transferred to the corresponding FTL. The data transfer time is modeled considering the channel contention that we will explain below. The space in the write buffer is reclaimed later, once the FTL completes the processing of the request. The write completion to the host can be notified as soon as the data is written to the write buffer (i.e., early completion) or when the data is completely written to the NAND, according to the operation types and configurations.

Read requests are processed without going through the write buffer. Instead, the FTL instance locates the data and transfers it to the buffer specified in the NVMe command. During the data

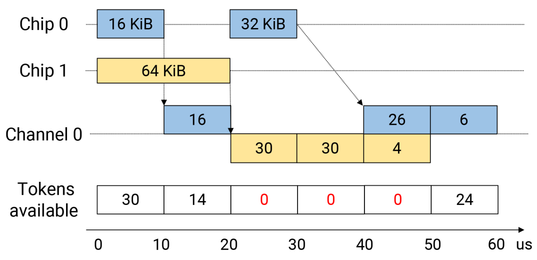


Fig. 3. The token-based contention model. Channel 0 is given with 30 tokens, each of which implies the right to transfer 1 KiB of data for 10 $\mu$s. This configuration effectively emulates the bandwidth of 3,000,000 KiB/second. Chips should take the required amount of tokens to transfer data over the channel. If tokens are not available, the operation is postponed until the tokens get available.

transfer, NVMeVirt determines the completion time by modeling the contention in the channel and the PCIe link.

The NAND chips in NVMeVirt are modeled to operate independently. Each chip has its next available time to process inbound operations. When a chip receives an operation, its completion time is calculated from the next available time of the chip and the time it takes to perform the operation. When multiple chips are involved in an operation, the completion is determined by the chip that takes the longest time to complete the operation.

Data within an SSD is transferred through the channels and the PCIe link, where contention may arise among the chips and channels, respectively. We model the contention in a bandwidth-limited channel with the concept of *tokens*. Each token represents the right to utilize a part of the channel to transfer the specified amount of data during the specified time slot. Thus, a token can be defined, for example, as 1 KiB for 10 $\mu$s or 4 KiB for 20 $\mu$s. Each channel is given with a number of tokens considering their maximum bandwidth. For instance, we can define a token of 1 KiB/10 $\mu$s and model a channel with a bandwidth of 3,000,000 KiB/s using 30 tokens. An entity can grab a number of tokens to transfer data during the time slot.

For example, as illustrated in Figure 3, when transferring 16 KiB of data over Channel 0, Chip 0 utilizes 16 tokens to complete the entire data transfer within 10 $\mu$s. Other chips may grab some of the remaining tokens and utilize the channel simultaneously. If the entity wants more tokens than what is available, it grabs all the remaining tokens but can only transfer data in proportion to the number of tokens grabbed. In Figure 3, to transfer 64 KiB, Chip 1 can acquire 30 tokens during the time period of 20 $\mu$s to 30 $\mu$s, and another 30 tokens during 30 $\mu$s to 40 $\mu$s. During the transfer, Channel 0 does not have any available token, and thus Chip 0, which needs to transfer 32 KiB of data, should wait until the time of 40 $\mu$s. At this point, Chip 1 holds 4 tokens, allowing Chip 0 to acquire the remaining 26 tokens for transferring 26 KiB of data during that time. Afterward, Chip 1 can complete the data transfer by acquiring 6 tokens at the time of 50 $\mu$s. We have employed this token-based contention model in emulating the channels and the PCIe link. Each channel and the PCIe link have their own tokens and time slots, which are configured based on their performance characteristics.

Currently, NVMeVirt performs **garbage collection (GC)** when the number of free blocks gets below a threshold. The victim block is selected according to the greedy policy. NVMeVirt maintains a separate GC update block so that the cold data migrated during GC is not intermixed with hot inbound data. To perform GC, NVMeVirt schedules internal operations (e.g., read and write-back pages) to the model. NVMeVirt is capable of emulating background GC as well as foreground GC.

It is noteworthy that NVMeVirt only calculates the timing of the operations, without actually moving the data during the emulation. NVMeVirt is highly configurable, allowing the user to customize the number of FTL instances, the size of the write buffer, and other operation times. We can estimate the timing parameters based on the device specification and/or by observing the performance behavior of the device, similar to the approaches proposed in the literature [29, 50, 57].

*3.5.3 ZNS SSD.* Basically, the ZNS SSD backend uses the same advanced data processing model as the conventional SSD. However, it does not need the full FTL since the host explicitly controls data placement according to the ZNS SSD specification. Thus, the ZNS SSD backend primarily focuses on maintaining the status of zones and reporting the information when queried by the host.

While analyzing the performance characteristics of real ZNS SSDs, we discovered that manufacturers seem to employ different zone provisioning schemes for ZNS SSDs. Specifically, some manufacturers appear to organize a zone by grouping only a few multiples of NAND flash blocks from the same channel together. This type of *small-zone* ZNS SSDs can provide a completely isolated performance and management scheme, albeit at the expense of lower performance due to limited parallelism.

On the other hand, other vendors form a zone using multiple NAND flash blocks that span multiple channels and NAND chips. As I/O operations for a zone can be distributed to multiple channels and chips in parallel, these *large-zone* ZNS SSDs can offer better performance compared to the small-zone ZNS SSDs. The small-zone and large-zone ZNS SSDs require different zone management policies. We designed the ZNS SSD backend in NVMeVirt to support both zone provisioning schemes.

## 3.6 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. 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 [36]. 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 allocating the data storing space dynamically using `alloc_pages()` or `vmalloc()`. In these cases, 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 at the moment. 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 and imposes a performance variance on the processing. For these reasons, we opt to use the linear mapping scheme, rather than the RAM disk or the dynamic space allocation 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). The space for the metadata is small, so it is fully allocated during the module initialization to minimize the potential performance variations.

**KVSSD.** The page-level address mapping is sufficient for conventional SSDs because the allocation unit is a fixed size, which is either larger than or equal to the page size. However, most of the keys and values in KVSSDs are much smaller than a single page (often tens to hundreds of bytes long) in general, and their sizes are highly variable. This necessitates a proper memory management scheme to prevent/control the external fragmentation of the address space yet provide a stable performance. Accordingly, we divide the first half of the reserved memory into 1 KiB chunks and the second half into 4 KiB ones. NVMeVirt maintains a bitmap to track the availability of each chunk. When a request requires a small chunk, we allocate the space from the 1 KiB chunk pool and mark the corresponding bitmap entry. When a chunk is reclaimed, its corresponding bitmap entry is cleared and the space is recycled. The large chunk pool is managed similarly.

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 [42] 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 bytes 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 192 GiB of memory, which provides 72 cores and a total of 384 GiB of memory. The system is also equipped with commercial storage devices for performance comparison and analysis: the 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 article. To evaluate KVSSD, we used the Samsung KVSSD [27]. 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

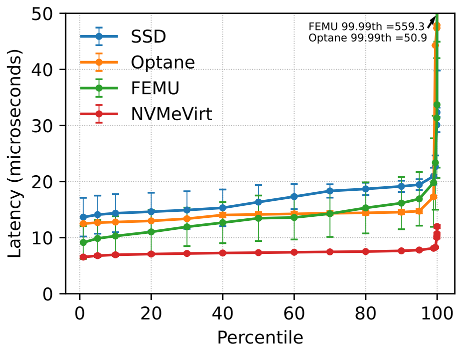


Fig. 4. Performance variance of real and emulated devices for 4 KB random write operations

Mellanox SX6012 switch supporting 56 Gbps FDR bandwidth. We implemented NVMeVirt based on the Linux kernel 5.10.37, and the implementation takes approximately 10,200 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 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 processor 1. In the meantime, applications and benchmarks are set to run on the cores on processor 0 with `taskset`. The memory for the NUMA 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 primarily focuses on facilitating various NVMe device types, and FEMU [36] 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 while processing inbound requests. Figure 4 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. Note that the small performance variance of NVMeVirt does not imply that NVMeVirt cannot emulate the performance variance of real devices. Rather, it indicates that NVMeVirt can effectively control the performance variance by providing stable and consistent performance. This is a very promising feature for emulating future high-performance storage devices.

On the other hand, the FEMU-emulated device 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

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 $\mu$s | 154 $\mu$s |
| Read bandwidth | 2.4 GiB | 2.6 GiB |
| Write latency | 14 $\mu$s | 56 $\mu$s |
| Write bandwidth | 2.0 GiB | 1.3 GiB |

| **Advanced Model** | | |
|---|---|---|
| | SSD | ZNS |
| Partitions | 4 | 1 |
| Write buffer | 1 MiB | 48 MiB |
| PCIe link bandwidth | 3.4 GiB | 3.2 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 $\mu$s | 40 $\mu$s |
| Write unit size | 32 KiB | 192 KiB |
| NAND write time | 185 $\mu$s | 1,913 $\mu$s |

These parameters are used to *characterize* the observed performance of the target device, not to describe their actual internals.

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 $\mu$s and a standard deviation of 462 $\mu$s. Considering the influences of the performance variance on the applications' tail latencies, FEMU will operate with a very high non-realistic tail latency.

## 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 Section 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 [10, 20, 54]. Note that the size of the write buffer is determined so that the SSD can double-buffer all data for ongoing programming. For example, the Samsung 970 Pro has 16 chips, where each chip is capable of independently performing 32 KiB page programming. In this case, we set the write buffer size to be 16 × 32 KiB × 2 = 1,024 KiB in order to double-buffer incoming payloads. We used various benchmark tools and configurations for evaluating various device types. For Optane and SSD, we used FIO [2] to measure the read and write bandwidth while varying the request size from 4 KiB to 256 KiB. We also used FIO for measuring the bandwidth 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 the write performance by measuring the aggregated bandwidth of different numbers of threads, each generating 192 KiB requests in accordance with the ZNS zone restriction.

To evaluate KVSSD, we used OpenMPDK KVBench [51], which is an open-source benchmark based on the ForestDB benchmark suite [6]. We report the aggregated bandwidth from various

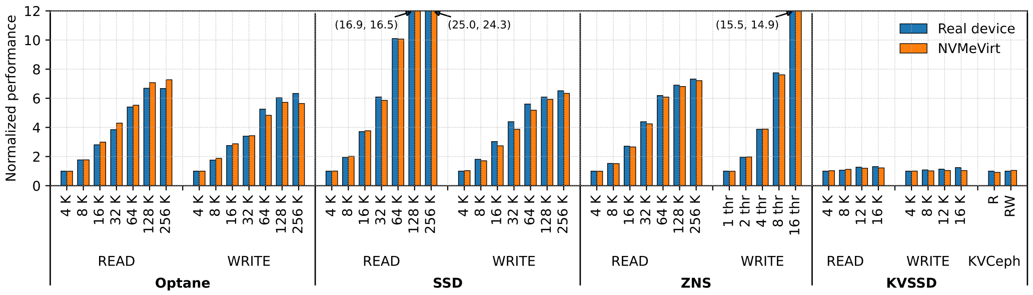


Fig. 5. The performance comparison of the real devices and virtual devices. The performance is normalized to the left-most value within each category.

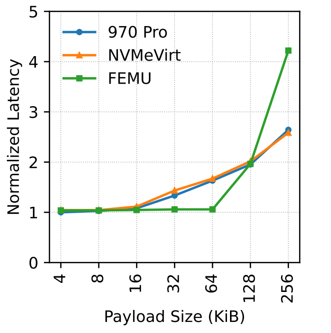
payload sizes. In addition, we report the aggregated bandwidth measured with KVCeph [31] that generates realistic key-value operation workloads.

Figure 5 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 virtual devices is small. The performance difference is by up to 11.6%, 11.8%, 3.3%, and 3.8% for Optane, SSD, ZNS SSD, and KVSSD, respectively.

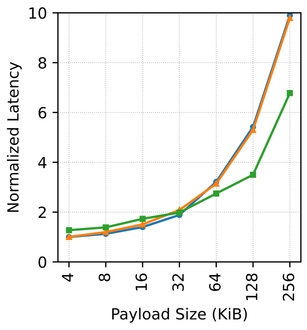
Next, we demonstrate the effectiveness of the token-based contention model by comparing the performance trend between NVMeVirt and FEMU. We configured FEMU to emulate the performance of the Samsung 970 Pro SSD, based on the observed 4 KiB read/write latency. Additionally, FEMU is set up with eight channels, each containing two LUNs according to the device specification of the reference device.

Figure 6 compares the latencies of random read and write operations with various payload sizes. The latency is shown normalized to that of the 4 KiB operation in the reference device. We can verify that NVMeVirt exhibits very similar performance trends to the reference device across the entire range of payload sizes. However, FEMU exhibits high latency deviation, particularly for large payload sizes. We attribute this to the absence of contention emulation in FEMU. In real devices, each LUN can operate independently, but they contend with each other while transferring data through the channels and the PCIe link. This contention inevitably leads to increased latency, particularly for large payloads. However, the performance model of FEMU does not consider such contention at all, and all 16 LUNs are allowed to operate independently without any limitations. This gives the flat read latency trend up to a payload size of 64 KiB for read operations, showing much higher performance than the real device. Even worse, FEMU attempts to compensate for the lower-than-real latency by introducing extra latency, which backfires when the payload size exceeds the range that FEMU is targeting. Write operations also exhibit a similar performance deviation. While FEMU can match the performance of a real device when handling small payloads, it struggles to accurately emulate contention. As a result, FEMU tends to exhibit significantly higher performance than the real device when dealing with larger payloads. From the evaluation, we can conclude that the proposed performance model can emulate resource contention effectively, which is particularly crucial when modeling the performance of modern SSDs that are inherently designed with high internal parallelism.

Finally, we analyzed various I/O characteristics. First, Figure 7(a) summarizes the latency distribution for processing 16 KiB requests in the Samsung 970 Pro and its NVMeVirt counterpart.

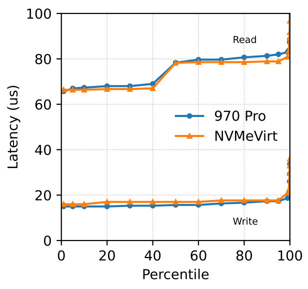


(a) Read latency on various payload sizes

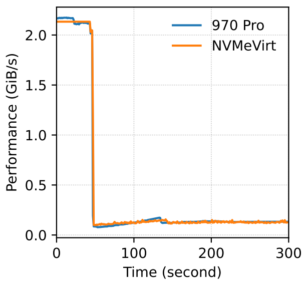


(b) Write latency on various payload sizes

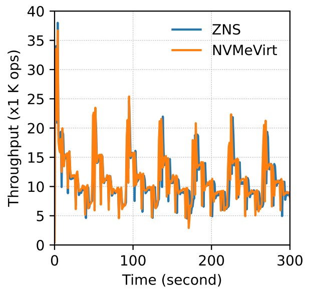
Fig. 6. The comparison of performance characteristics of emulators.



(a) Latencies



(b) The effect of GC



(c) RocksDB YCSB-A

Fig. 7. The comparison of various performance characteristics.

We can see the similar latency distributions between the setups. In particular, the sophisticated performance model can emulate the bimodal read latencies originated from different LSB- and MSB-page read times in MLC flash memory. Second, 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 7(b) shows the performance change, and we can see that NVMeVirt exhibits the realistic performance change when GC is involved. Lastly, Figure 7(c) 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.

Based on these evaluations, we can conclude that NVMeVirt is capable of modeling various performance aspects of the target devices faithfully.

## 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 its performance with the FIO benchmark. The NVMeVirt instance is configured to emulate Optane with the target performance listed in Table 2. Figure 8 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

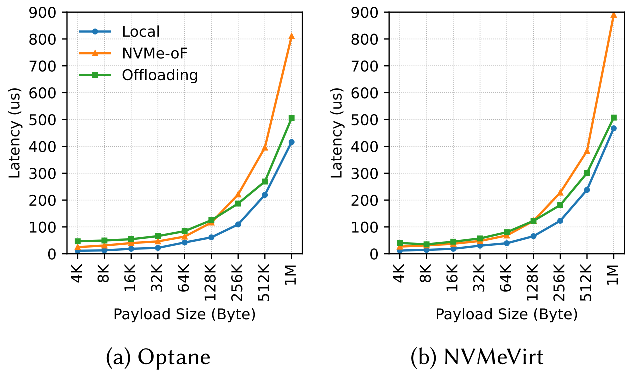


Fig. 8. The write performance as the NVMe-oF target.

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.

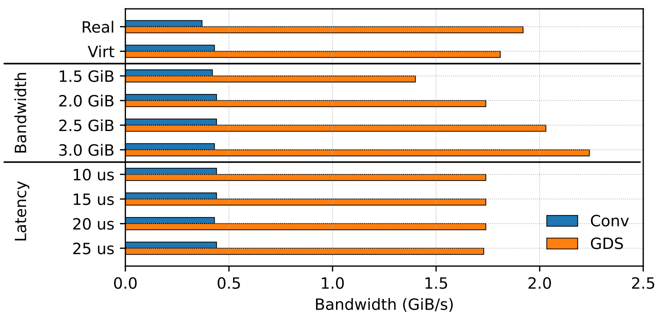
We can observe that NVMeVirt emulates the performance of Optane over NVMe-oF closely. Specifically, the baseline configuration without the target offloading outperforms that with the target offloading when the payload is smaller than 128 KiB. We attribute the performance trend to the NVMe command processing 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 [43, 44]. GPU directly accesses NVMe devices through PCI peer-to-peer DMA protocol, enabling decreased latency and CPU utilization on the host. We analyze the implication of storage performance on the GDS environment with NVMeVirt.

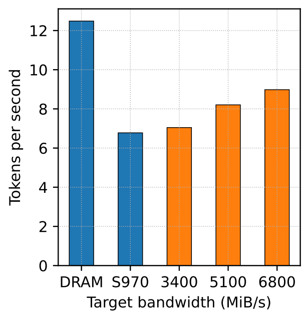
We measure the performance of checkpointing while running Megatron DeepSpeed [41] with NVIDIA A100 GPU. As shown in Figure 9(a), with the Samsung 970 Pro SSD, the checkpointing occurs at the rate of 0.37 GiB/s with the conventional storage configuration, where checkpointing data is stored through the 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 $\mu$s (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 9(a), 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



(a) Checkpointing performance of Megatron DeepSpeed



(b) Inference performance of OPT-30B

Fig. 9. Performance of AI workloads on various storage configurations and performance.

performance over a wide range of latencies, we can infer that the workload is likely to process I/O with a high queue depth.

We investigate the performance implications of the storage device on an AI inference workload. Using the DeepSpeed ZeRO-3 framework [8], we executed the OPT-30B model [49] and measured the inference performance under various storage performance. The model requires approximately 60 GB of space for storing the entire model parameters in half precision, which is too big to fit into the GPU memory (40 GB) of our NVIDIA A100 card. Thus, the framework divides the model parameters into smaller chunks, saves the chunks on the storage, and loads them in a pipelined fashion considering the stage they are required for during the inference process. We measure the rate of generating 10 tokens from 4 input tokens using the Hugging Face token generation pipeline with the batch size of 256.

Figure 9(b) illustrates the inference performance across different storage bandwidths. "DRAM" refers to the performance when the model parameters are loaded entirely into the host memory. In this case, inferences do not require any I/O operations for storage, exhibiting the highest level of performance. "S970" indicates the performance when the model parameters are initially stored on a Samsung 970 Pro SSD. As the pipelined parameter loading keeps issuing read requests to storage, parameter loading time is influenced by the device bandwidth. Thus, it shows a lower inference performance than the "DRAM" configuration.

We investigated further how much the inference performance is influenced by the storage bandwidth. We configured a baseline configuration with an NVMeVirt instance modeling the Samsung 970 Pro. Then we changed the storage bandwidth by increasing the channel and the PCIe link bandwidth simultaneously. Specifically, the baseline configuration uses a 1:4.25 (800 MiB:3,400 MiB) bandwidth ratio between the channel and the PCIe link. Accordingly, when we increase the channel bandwidth by 400 MiB/s, the PCIe link bandwidth is increased by 1,700 MiB to keep their bandwidth ratio.

The orange bars in Figure 9(b) show the inference performance at the configured PCIe link bandwidth (i.e., 3400 implies the performance when the storage provides the bandwidth of 3,400 MiB/s). We can clearly see the impact of storage bandwidth in the AI inference workload; the higher bandwidth the device provides, the higher inference rate the model can generate. However, the increase is not fully proportional to the bandwidth; when the bandwidth is doubled from 3,400 MiB/s to 6,800 MiB/s, the performance is only increased by 27.3%. We found that the framework does not utilize the full bandwidth of the storage, and we attribute the limited bandwidth usage to the under-optimized parameter loading scheme and the heavy I/O stack on the host. Therefore, we can conclude that there is room for improvement in the storage subsystem of AI application frameworks.

### 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]. We created an NVMeVirt instance configured as an NVM SSD and configured the database instance on it with recommended configurations from optimization tools [16, 47]. The database instance is then populated with sysbench [52] 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 10 summarizes the results. We only report the trends of the first 5 minutes since the performance became stable afterward.

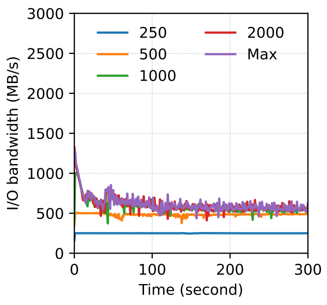
Overall, we verified that MariaDB and PostgreSQL react very differently to the storage performance. Figures 10(a) and 10(b) compare the I/O bandwidth utilization of MariaDB and PostgreSQL over time when the target bandwidth is set to the given value (i.e., 250 implies the storage bandwidth is limited 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 10(c) 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 the 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 10(d) 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 $\mu$s. However, as the latency decreases, the TPS of PostgreSQL keeps increasing, becoming comparable to that of MariaDB when the latency is minimum. Figures 10(e) and 10(f) show 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.

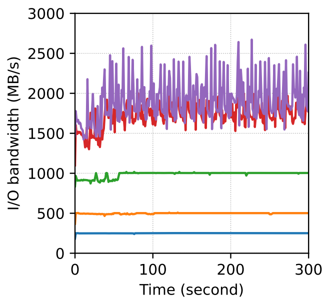
From the evaluation, we can conclude that PostgreSQL is more promising on modern storage devices, whereas MariaDB is more efficient when the storage is slow. We can verify that NVMeVirt allows us to estimate the performance of applications on future storage devices.

### 4.6 Case for NVMe Interface Study

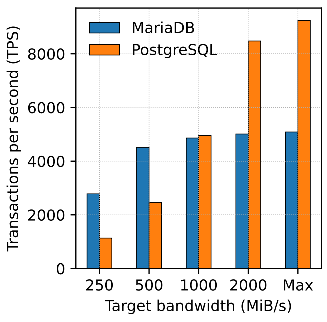
As NVMeVirt handles inbound NVMe operations in software, it opens up the opportunity to extend the host-device interface easily. To demonstrate this, we made a case with one of the recent studies whose evaluation is limited by the host-device interface modification. Specifically, Kim et al. [30] proposed 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



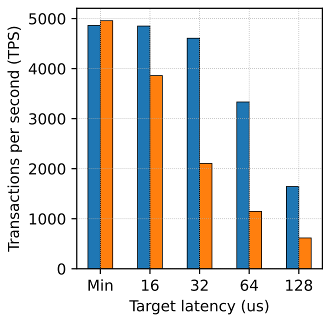
(a) MariaDB



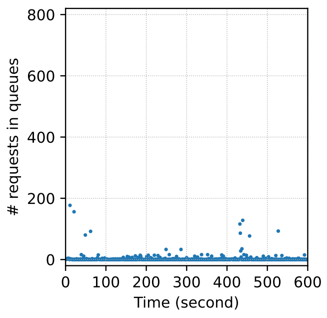
(b) PostgreSQL



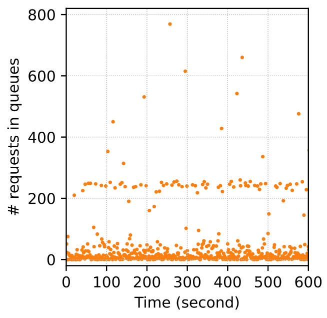
(c) TPS vs. target bandwidth



(d) TPS vs. target latency



(e) Queue depth (MariaDB)



(f) Queue depth (PostgreSQL)

Fig. 10. Performance characteristics of the MariaDB and PostgreSQL database engines on various storage configurations.

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

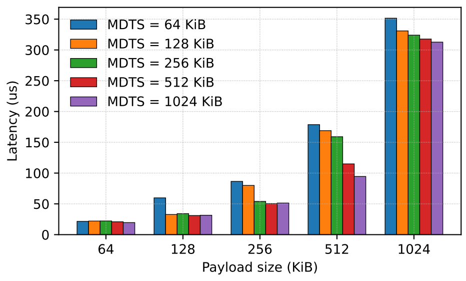


Fig. 11. The effect of the MDTS value on device performance.

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 $\mu$s, which is reduced to 86 $\mu$s with the compound command, giving a 5.4x performance gain. This improvement is higher than the value reported in the original work (92.0 $\mu$s to 41.5 $\mu$s), and we attribute the extra improvement to the conservative estimation in the original work.

## 4.7 Case for Evaluating Device Configurations

We can easily evaluate the effect of various device configuration parameters with NVMeVirt, as it is purely implemented in software. In this case study, we evaluate the impact of changing the **maximum data transfer size (MDTS)** value of the NVMe device on the latency. According to the NVMe standard, the device declares the MDTS value it can process. If the size of a request exceeds the MDTS value, the host device driver should process it by dividing it into smaller requests. Based on the definition of MDTS, we can anticipate that a device with a larger MDTS will be able to perform faster on large requests. To verify this hypothesis, we measured the latency of random reads on various payload sizes while changing the MDTS value of an NVMeVirt instance emulating the Optane NVM SSD. We used the FIO benchmark to measure the latency.

Figure 11 summarizes the result, which is the average of five runs. When the payload size is smaller than the MDTS value, the latency is hardly affected since each request is processed as it is, without being divided into smaller parts. However, as the payload size becomes larger than the MDTS value, the latency is significantly influenced because both the device driver and the device itself need to handle increased number of requests. For example, increasing the MDTS value from 64 KiB to 1,024 KiB results in a reduction in latency ranging from 11.0% to 48.0% when processing large requests.

From the evaluation results, we can confirm the significant impact of MDTS, indicating that storage devices would benefit from supporting larger MDTS values to efficiently handle large requests. Note that the MDTS value of the real device cannot be changed, while it can be easily configured in the NVMeVirt instance by adjusting the value during compilation. This demonstrates the versatility of NVMeVirt as a valuable tool for conducting research on storage systems.

## 5 CONCLUSION

We presented NVMeVirt, a virtual, software-only NVMe device. As it operates in software, users can easily utilize it for performing various research on sophisticated storage configurations such as user-level I/O bypassing the kernel I/O stack, PCIe peer-to-peer communication, and RDMA over NVMe-oF. It also opens a new opportunity for developers to co-design highly intelligent storage devices over the NVMe interface. In the evaluation, we demonstrated the usefulness of NVMeVirt for storage research.

## ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for invaluable feedback. We also thank Hyeong-Jun Kim who developed the initial prototype of NVMeVirt.

## REFERENCES

[1] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU emulation. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC'11)*. USENIX Association.

[2] Jens Axboe. n. d. 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. 2021. FlashNeuron: SSD-enabled large-batch training of very deep neural networks. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association.

[4] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the block interface tax for flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC'21)*. USENIX Association.

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 359–375.

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

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

[8] DeepSpeed. n. d. ZeRO-Inference: Democratizing massive model inference. https://www.deepspeed.ai/2022/09/09/zero-inference.html

[9] 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. 2020. Cost-effective, energy-efficient, and scalable storage computing for large-scale AI applications. *ACM Transactions on Storage* 16, 4, Article 21 (2020), 37 pages.

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

[11] NVM Express. n. d. NVMe-oF Specification. https://nvmexpress.org/developers/nvme-of-specification/

[12] Congming Gao, Min Ye, Chun Jason Xue, Youtao Zhang, Liang Shi, Jiwu Shu, and Jun Yang. 2022. Reprogramming 3D TLC flash memory based solid state drives. *ACM Transactions on Storage* 18, 9 (May 2022), 133 pages.

[13] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. 2018. 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'18)*. 469–481.

[14] 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. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA'16)*. 153–165.

[15] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. 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'21)*.

[16] Major Hayden. n. d. MySQLTuner. https://github.com/major/MySQLTuner-perl

[17] Chien-Chung Ho, Yung-Chun Li, Yuan-Hao Chang, and Yu-Ming Chang. 2018. Achieving defect-free multilevel 3D flash memories with one-shot program design. In *Proceedings of the 55th ACM/ESDA/IEEE Annual Design Automation Conference (DAC'18)*. 6 pages.

[18] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*, 173–187.

[19] Intel. n. d. Intel I/O Acceleration Technology. https://www.intel.co.kr/content/www/kr/ko/wireless-network/accel-technology.html

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

[21] 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. n. d. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 http://arxiv.org/abs/1903.05714

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

[23] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 373–384.

[24] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45rd ACM/IEEE International Symposium on Computer Architecture (ISCA'18)*. Los Angeles, CA.

[25] Myoungsoo Jung. 2020. OpenExpress: Fully hardware automated open research framework for future fast NVMe devices. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*.

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

[27] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR'19)*. 144–154.

[28] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. 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'16)*.

[29] Joonsung Kim, Kanghyun Choi, Wonsik Lee, and Jangwoo Kim. 2021. Performance modeling and practical use cases for black-box SSDs. *ACM Transactions on Storage* 17, 2, Article 14 (June 2021), 38 pages.

[30] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. 2019. 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'19)*.

[31] kvceph. n. d. Open memory platform development kit. http://github.com/OpenMPDK

[32] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage* 16, 3, Article 15 (Aug. 2020), 35 pages.

[33] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. 2022. 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'22)*. USENIX Association.

[34] Parallel Data Lab. n. d. The DiskSim Simulation Environment v4.0. https://www.pdl.cmu.edu/DiskSim/index.shtml

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

[36] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. 2018. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*.

[37] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. 2019. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*. USENIX Association.

[38] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. 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'13)*. USENIX Association, 257–270.

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

[40] Krishna T. Malladi, Mu-Tien Chang, Dimin Niu, and Hongzhong Zheng. 2017. FlashStorageSim: Performance modeling for SSD architectures. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS'17)*. 1–2.

[41] Microsoft. n. d. Megatron-DeepSpeed. https://github.com/microsoft/Megatron-DeepSpeed

[42] Landon Curt Noll. n. d. FNV Hash. http://isthe.com/chongo/tech/comp/fnv/

[43] NVIDIA. n. d. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. https://developer.nvidia.com/blog/gpudirect-storage

[44] NVIDIA. n. d. The NVIDIA Magnum IO GPUDIrect Storage Overview Guide. https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html

[45] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. 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'14)*.

[46] PCI SIG. n. d. PCI SIG. https://pcisig.com

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

[48] QMEU. n. d. QEMU: A Generic and Open Source Machine Emulator and Virtualizer. https://qemu.org

[49] Facebook Research. n. d. Metaseq. https://github.com/facebookresearch/metaseq

[50] Manoj P. Saha, Adnan Maruf, Bryan S. Kim, and Janki Bhimani. 2021. KV-SSD: What is it good for? In *Proceedings of the 58th ACM/IEEE Annual Design Automation Conference (DAC'21)*. 1105–1110.

[51] Samsung Electronics. n. d. OpenMPDK KVSSD. https://github.com/OpenMPDK/KVSSD/

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

[53] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. 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'18)*.

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

[55] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. 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'19)*.

[56] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carolejean Wu, David Michael Brooks, and Guyeon Wei. 2021. 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'21)*.

[57] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an unwritten contract of Intel Optane SSD. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19)*.

[58] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. 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'17)*. 154–161.

[59] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. 2013. VSSIM: Virtual machine based SSD simulator. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST'13)*.

[60] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association.

Received 19 May 2023; accepted 12 September 2023