FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization

Dongup Kwon, *Department of Electrical and Computer Engineering, Seoul National University / Memory Solutions Lab, Samsung Semiconductor Inc.;* Junehyuk Boo and Dongryeong Kim, *Department of Electrical and Computer Engineering, Seoul National University;* Jangwoo Kim, *Department of Electrical and Computer Engineering, Seoul National University / Memory Solutions Lab, Samsung Semiconductor Inc.*

https://www.usenix.org/conference/osdi20/presentation/kwon

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization

Dongup Kwon[1,2], Junehyuk Boo[1], Dongryeong Kim[1], Jangwoo Kim[1,2,*]

[1]*Department of Electrical and Computer Engineering, Seoul National University*
[2]*Memory Solutions Lab, Samsung Semiconductor Inc.*

Abstract

Emerging big-data workloads with massive I/O processing require fast, scalable, and flexible storage virtualization support. Hardware-assisted virtualization can achieve reasonable performance for fast storage devices, but it comes at the expense of limited functionalities in a virtualized environment (e.g., migration, replication, caching). To restore the VM features with minimal performance degradation, recent advances propose to implement a new software-based virtualization layer by dedicating computing cores to virtual device emulation. However, due to the dedication of expensive general-purpose cores and the nature of host-driven storage device management, the proposed schemes raise the critical performance and scalability issues with the increasing number and performance of storage devices per server.

In this paper, we propose FVM, a new hardware-assisted storage virtualization mechanism to achieve high performance and scalability while maintaining the flexibility to support various VM features. The key idea is to implement (1) a storage virtualization layer on an FPGA card (FVM-engine) decoupled from the host resources and (2) a device-control method to have the card directly manage the physical storage devices. In this way, a server equipped with FVM-engine can save the invaluable host-side resources (i.e., CPU, memory bandwidth) from virtual and physical device management and utilize the decoupled FPGA resources for virtual device emulation. Our FVM-engine prototype outperforms existing storage virtualization schemes while maintaining the same flexibility and programmability as software implementations.

1 Introduction

Storage virtualization is one of the most important components to determine the cost-effectiveness of modern datacenters, which improves the utilization of the storage devices and makes resource management much easier. For example,

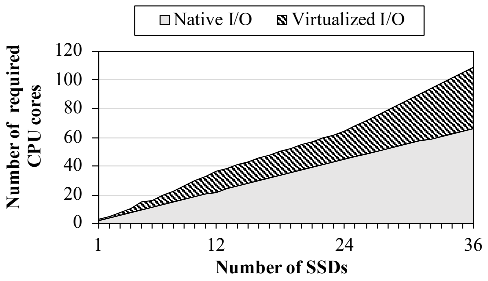


Figure 1: CPU usage of native block I/O in Linux and virtualized block I/O with SPDK vhost-nvme [21]

storage virtualization can map multiple virtual storage devices onto a smaller set of physical storage devices and make them shared by many virtual machines (VMs) [66]. At the same time, it facilitates VM management by providing a variety of functionalities in a virtualized context (e.g., live migration [41, 58], replication [52, 61], consolidation [62, 65], aggregation, metering, server-side caching [35, 37]).

The importance of storage virtualization is growing for modern datacenters running I/O-intensive big-data workloads on their fast but expensive solid-state drives (SSDs). In particular, it is critical to reduce virtualization overhead and provide near-native storage performance to the VM workloads. A conventional way to overcome the virtualization overhead is to utilize hardware-assisted virtualization mechanisms (e.g., passthrough [30], SR-IOV [19]). However, the existing hardware virtualization mechanisms have become much less appealing to modern datacenters due to their extremely limited VM management support.

To provide highly flexible VM management at minimal virtualization overhead, a new software-based storage virtualization mechanism is now considered as a highly promising solution. A storage performance development kit (SPDK) vhost-target implementation not only enables flexible VM management but also significantly improves performance by exclusively dedicating computing cores (i.e., sidecores) to its user-level virtualization layer [21, 69].

However, such sidecore approaches require a significant

*Corresponding author.

amount of computing resources to execute their polling-based virtual device emulation [53, 59]. Furthermore, the required computing bandwidth quickly increases as a single server is equipped with an increasing number of storage devices, and each device gets faster. As shown in Figure 1, our projection result shows that virtualized I/O with SPDK vhost-nvme [21] necessitates 42% – 65% more CPU cores to saturate multiple Intel Optane SSDs [7] than native block I/O in Linux.

Due to the severe computing resource requirement, the software-based storage virtualization cannot provide high performance or high scalability. First, without enough computing cores dedicated to the storage virtualization layer, the storage system comes to suffer from low performance. Second, without the capability of adding virtualization-dedicated cores as needed, the system comes to suffer from low scalability. Therefore, to achieve high performance, scalability, and flexibility all together, the ideal storage virtualization should decouple itself from host CPU cores, scale with a target storage system, and exploit the most cost-effective computing solution for the programmable VM management.

In this paper, we design and implement FVM, a new hardware-assisted storage virtualization mechanism, to achieve high performance and scalability while maintaining the flexibility to support a variety of VM management features. The key idea of FVM is to implement (1) a storage virtualization layer on an FPGA card (*FVM-engine*) which is decoupled from the host resources, and (2) a hardware-based device-control mechanism to make the card directly manage the physical storage devices. FVM also leverages (3) high-level synthesis (HLS) techniques to provide easy programmability for VM management. Our solution can also be implemented on ASICs for higher performance, but in that case, the ASIC implementations lose future flexibility for new VM management features.

FVM achieves the design goals as follows. First, FVM achieves high performance by utilizing a hardware-assisted virtualization mechanism and leveraging massive parallelism in the modern storage virtualization stack. FVM-engine can cost-effectively exploit the virtualization's parallelism by implementing many wimpy FVM cores and distributing virtual/physical I/O queues and queuing routines to them for fine-grained parallel executions.

Second, FVM achieves high scalability by executing virtual device emulation on FVM-engine, which is decoupled from host CPU cores and device resources. In addition, its direct device-control mechanism further improves the scalability by enabling FVM-engine to directly manage the physical devices. Therefore, without relying on expensive host CPU cores, FVM can achieve highly scalable virtualization performance by implementing FVM-engine on a more powerful FPGA card or adding more FPGA cards on a system board.

Third, FVM achieves highly flexible storage virtualization by implementing existing VM management features on a reconfigurable FPGA card. For user programmability, we leverage an HLS-based design flow and separate the virtualization layer from the I/O logic to interact with the host machine and the physical storage devices.

	Sidecore		PassTh	SR-IOV	FVM
	CPU	On-dev SoC	(1VM)		
	[59, 69]	[53]	[30]	[19]	
Performance†	✓		✓	✓	✓+
Host efficiency		✓	✓	✓	✓
Scalability			✓	✓	✓
Device sharing	✓	✓		✓	✓
Flexibility‡	✓	✓			✓
Programmability	✓	✓			✓

†: I/O throughput, latency, ‡: Providing seamless storage-related services.

Table 1: Comparison of the existing and proposed storage virtualization mechanisms

Table 1 summarizes FVM's key advantages over existing software- and hardware-based storage virtualization mechanisms, in terms of performance, host efficiency, scalability, device sharing, flexibility, and programmability. FVM solves the performance and scalability issues of the recent sidecore approaches, while achieving device sharing and flexibility that the existing hardware-assisted techniques cannot provide. A detailed explanation can be found in Section 3.

For evaluation, we implemented our FVM-engine prototype on a Xilinx FPGA board [23] and Intel Optane SSDs [7]. We implemented Linux device drivers for the software support and augmented an SPDK vhost-target implementation [21] to apply FVM to an existing KVM-based virtualization system [11].

Our experimental results show that the FVM prototype obtains 1.36× higher I/O throughput than the software-based virtualization method when allocating the same amount of host CPU cores. FVM also scales well with the increasing number of VMs and virtual/physical storage devices by achieving 9.5 GB/s aggregate I/O throughput with four SSDs. Also, our HLS-based design flow requires only 10s – 100s of code lines to implement example VM management functionalities.

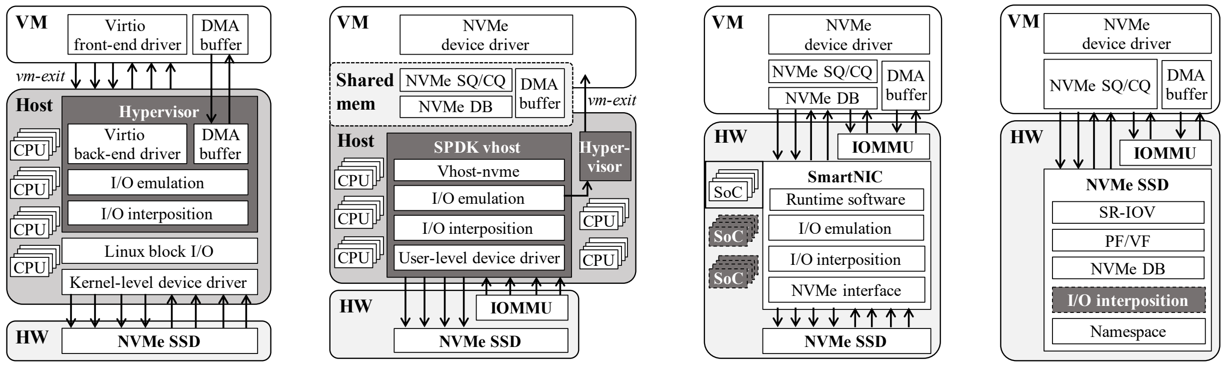
In summary, we make the following contributions:

- **Novel storage virtualization mechanism**: We propose a novel FPGA-assisted virtual device emulation mechanism for fast, scalable, and flexible storage virtualization.

- **High performance**: FVM achieves high performance by utilizing hardware-assisted virtualization and parallelizing virtual/physical device operations on FVM-engine.

- **High scalability with host efficiency**: FVM can easily increase its computing power to match the target virtualization scalability without depending on host resources.

- **Flexibility & programmability**: Our HLS-based FVM design flow supports easy VM management and feature programmability.



(a) Paravirtualization (b) Host sidecore (c) On-device sidecore (d) Direct assignment

Figure 2: System architectures for conventional storage virtualization mechanisms

2 Background

In this section, we introduce modern non-volatile memory (NVM) technologies and the latest advances in storage virtualization mechanisms.

2.1 NVM and NVMe Protocol

Modern NVM technologies such as 3D XPoint [8] and Z-NAND [18] have significantly improved the storage performance [39, 50, 67, 71]. At the same time, virtualization for such fast storage devices becomes one of the most critical components in cloud environments [53, 55, 59, 69]. For example, Amazon Web Services (AWS) accelerates I/O virtualization through dedicated hardware components [1, 55]. Other major cloud providers, including Microsoft Azure [12] and Google Cloud Platform (GCP) [6], are allowing advanced NVM devices to be used as primary storage for VMs.

NVM Express (NVMe) [14] is a standard storage architecture used to enable fast NVM storage through PCIe and optimized I/O paths. First, it brings multiple deep I/O queues to take full advantage of NVM technologies. The current specification supports up to 65,535 I/O queues, each with 1 – 65,535 outstanding commands. As a result, it can enable highly parallel processing on multiple cores by assigning independent I/O queues and queuing routines to each core or thread. Second, its protocol provides fast I/O submission and completion paths by reducing the number of memory-mapped I/O (MMIO) operations. For example, it does not require MMIO register reads in the common I/O paths, while including a maximum of one MMIO register write for the command submission path.

An NVMe I/O queue consists of a *submission queue* (SQ)/*completion queue* (CQ) pair. For I/O submission, host software places NVMe commands in the SQ and writes the SQ tail pointer to the target *SQ doorbell register* exposed through PCIe base address registers (BARs). The target NVMe storage device then fetches the newly added commands and processes them. Once the NVMe commands are completed, the NVMe device writes completion messages to the associated CQ and then generates an interrupt. Lastly, the host software handles the completion messages and updates the target *CQ doorbell register* to clear the interrupt and release the CQ entries.

2.2 Storage Virtualization

2.2.1 Paravirtualization

In a paravirtualization scheme, a guest operating system (OS) is installed with a VM abstraction to make it efficient to emulate virtual devices. For example, *Virtio* [60] is an abstraction for virtual devices in a hypervisor. This abstraction allows the hypervisor to export a common set of virtual devices and makes them available to guests through an efficient device interface. Figure 2a shows the system architecture for virtio-based device emulation. The guest implements front-end virtio drivers, with particular virtual device emulation behind a set of back-end drivers in the hypervisor [60]. This paravirtualization mechanism can reduce the number of *VM exits* by reducing the number of MMIO operations for the virtual device of the guest, which addresses the huge performance overhead incurred by CPU mode switches and cache pollution [51]. However, the guest OS should be aware that it is being virtualized, which requires modifications to collaborate with the hypervisor efficiently.

Virtio SCSI (*virtio-scsi*) [47] or block (*virtio-blk*) [45] can be used to emulate an NVMe device with this paravirtualization mechanism. They handle VM requests directed at the *virtual* NVMe device as follows: (1) A guest OS makes a request to a virtual device through virtual I/O queues (e.g., *vring* [60]) in virtio front-end drivers. (2) The guest then calls a VM exit and traps into a host machine. (3) The hypervisor emulates the virtual device through virtio back-end drivers, interacting with kernel-level device drivers. (4) Once the I/O request is completed, the virtio back-end drivers read completion messages from the physical devices, confirm their completion status, and inject an interrupt to the guest OS through the hypervisor.

2.2.2 Host Sidecore Approach

CPU-dedicated (or *sidecore*) approaches can further accelerate storage virtualization by avoiding expensive traps to the hypervisor and reducing cache pollution [33, 48]. The recently proposed SPDK *vhost-scsi* and *vhost-blk* implementations [21] can accelerate virtualization of NVMe storage. As shown in Figure 2b, a hypervisor pre-allocates *shared memory regions* for guests and allows them to exchange storage commands with SPDK vhost-target directly for virtual device emulation. The SPDK vhost-target implementations emulate VM requests as follows: (1) A user-space thread running on a dedicated sidecore continues to poll virtual I/O queues (e.g., NVMe SQ/CQ pairs) via a shared memory region. (2) It reads newly received VM SCSI or block requests and converts them to NVMe commands (i.e., *protocol conversion*). (3) It conducts the I/O operations through an SPDK *user-level* NVMe device driver. (4) Once the requests are completed, another dedicated thread in SPDK vhost-target deals with completion messages and injects an interrupt to the guest through the hypervisor.

The recent sidecore approaches can offer near-native performance of modern NVMe devices to VMs. A dedicated sidecore polls guest I/O operations through shared memory regions, so there is no need to call VM exits to submit NVMe commands. Moreover, SPDK's user-level NVMe device driver enables sidecores to conduct I/O operations without user-kernel mode switches. SPDK vhost-target also reduces the number of data copies by allocating guest DMA buffers in a *pinned* shared memory region. For this, the software-based virtualization layer translates *guest physical addresses* (gPAs) to pinned *host physical addresses* (hPAs). Due to the address translation, the NVMe device can transfer data directly to the guest's memory space without being aware that it receives requests from VMs.

Vhost-nvme. The SPDK *vhost-nvme* implementation [69] further optimizes the sidecore approaches by directly exposing NVMe devices to guest OSes. This transparent view of the NVMe devices can eliminate the performance loss caused by the protocol conversion between SCSI/block and NVMe. It also allows the guest OSes to exploit advanced NVMe features (e.g., shadow doorbell buffer [42]) to get higher performance. A recent study [69] demonstrated that the vhost-nvme implementation gets $1.11\times - 1.26\times$ higher random-read throughput than the other SPDK vhost-target implementations.

2.2.3 On-device Sidecore Approach

To save the host resources required for storage virtualization, a recent study [53] offloaded the virtualization layer to system-on-chip (SoC) cores in other peripheral devices (e.g., SmartNIC [2, 13]). Figure 2c shows its system architecture. The *on-device* sidecore approach exposes virtual NVMe interfaces to guest OSes by providing a uniform address space across host CPUs and SoC cores. The SoC allows the runtime software running on the on-device cores to reach virtual NVMe queue pairs mapped in the host memory through DMA. In addition, host software allocates NVMe queue pairs in the SoC's memory space and provides their locations to the physical NVMe device to make it interact with the SoC directly. Since it utilizes on-device sidecores to emulate virtual storage devices, it can save the host CPU resources and offer more compute power to VMs or other VM management features.

Moreover, on-device sidecore mechanisms provide flexible and programmable implementations leveraging ARM-based SoC cores. In particular, it facilitates implementing essential functionalities in storage virtualization, which are not fully offloaded or not easily composable via other hardware-based virtualization mechanisms (e.g., SR-IOV). For example, a recent study [53] implemented storage versioning, prioritization, isolation, replication, and aggregation functionalities in runtime software installed on the SoC.

2.2.4 Direct Device Assignment

To overcome the virtualization overhead, VMs can make use of support for DMA and interrupt remapping (e.g., Intel VT-d [27], AMD-Vi [26]), which allows guest software to access a target storage device directly. For the remapping support, major processor manufacturers introduced *I/O memory management units* (IOMMUs). A DMA remapping engine in an IOMMU allows DMAs from a guest to be accomplished with gPAs. The IOMMU translates them into hPAs according to page tables that are configured by host software. Likewise, an interrupt remapping engine translates interrupt vectors issued by devices based on an interrupt translation table.

The direct device assignment (or *passthrough*) eliminates the virtualization overhead in software layers since the hypervisor is no longer in a guest's I/O paths. However, this approach requires the physical devices to be exclusively assigned to a single VM and does not support device sharing across multiple VMs. Therefore, the passthrough mechanism has limitations in improving the utilization of storage devices and reducing operating costs in modern datacenters.

SR-IOV. To address the challenges of the direct passthrough scheme, the PCIe specification currently supports *SR-IOV* [19], a standardized hardware virtualization protocol. An SR-IOV capable PCIe device supports a *physical function* (PF) and multiple *virtual functions* (VFs). The PF provides resource management for the device and is managed by the host software, and each VF can be assigned to a single VM exclusively for direct access. SR-IOV is now supported by high-performance I/O devices such as network and storage devices as well as accelerators. Recently, Xilinx released an SR-IOV capable PCIe IP block [28], supporting up to 252 VFs.

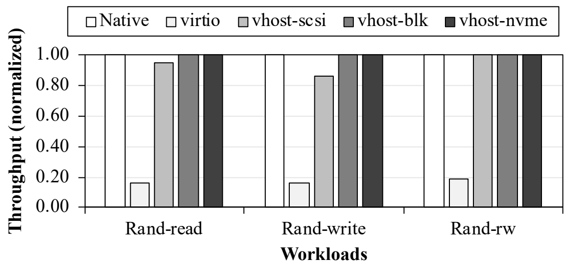


Figure 3: Random I/O throughput with a single SSD

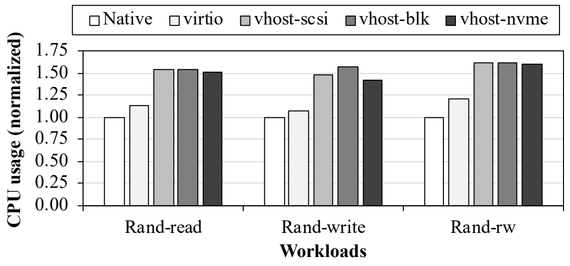


Figure 4: CPU usage of random I/O with a single SSD

Since an SR-IOV capable device implements how to multiplex itself at the hardware level, it does not rely on host software to multiplex the virtual device instances, as shown in Figure 2d. In addition, with SR-IOV and an IOMMU in a host machine, VFs can carry out DMA transactions with gPAs, while avoiding the software-side address translation. Similarly, interrupt remapping for each VF addresses the performance overhead generated by triggering interrupts to notify guests regarding the completion of their I/O requests.

3 Motivation

In this section, we discuss the challenges of the existing virtualization mechanisms for modern datacenters. We identify the critical performance and scalability issues of the existing host and on-device sidecore approaches and the limited VM management support of the hardware-assisted virtualization technologies.

3.1 CPU-inefficient Storage Virtualization

Modern software-based storage virtualization mechanisms dedicate CPU sidecores to emulate virtual NVMe devices. For example, recent NVMe virtualization studies [59,69] allocate multiple CPU cores to poll virtual I/O queues via shared memory regions, instead of making a trap into a host machine. Figure 3 shows the random I/O throughput of the various software-based virtualization implementations on a single CPU sidecore and a single Intel Optane SSD [7], normalized to the native performance. Virtio denotes virtio-based paravirtualization through KVM, and vhost-scsi, -blk, and -nvme mean three different virtual device interfaces through SPDK vhost-target. We ran FIO [5] random I/O benchmarks with four threads and 32 queue depth and measured the random I/O throughput on VMs. On the other hand, Figure 4 shows the relative CPU usage normalized to that of the native I/O operations on an Intel Xeon server [22] with the same experiment environments. Our experimental results show that virtio fails to offer the full native performance due to frequent VM exits, and all the SPDK vhost implementations can achieve close to the maximum native performance (i.e., 550K IOPS). However, at the same time, to get such near-native performance, they demand 1.42× – 1.61× more CPU resources than native random I/O operations.

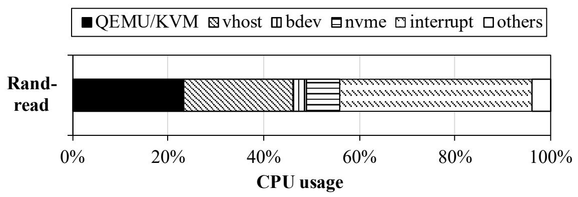


Figure 5: CPU usage breakdown of random-read I/O with SPDK vhost-nvme

There are two primary sources of such high CPU resource usage. First, they utilize multiple active polling cores to reduce the number of VM exits [59, 69]. Because NVMe is a highly parallel storage architecture, the conventional trap-and-emulate approach will generate an unacceptable number of VM exits for a VM to take full advantage of multiple I/O queues [59]. For this reason, the sidecore approaches allocate CPU resources exclusively and poll guest I/O operations through a shared memory region to handle such frequent NVMe requests quickly. Second, the SPDK vhost-target implementations trigger guest interrupts through *eventfd*, which requires system calls and VM exits [69]. Figure 5 shows the CPU usage breakdown of the host machine running SPDK vhost-nvme. Our experimental result demonstrates that around 22% of active CPU cycles are used to poll and emulate virtual devices (vhost) and 39% to trigger guest interrupts (interrupt). The other portions are consumed by the necessary VM management (QEMU [15]/KVM [11]) and the SPDK storage stack (bdev and nvme).

This resource-inefficiency issue poses a significant challenge to scalable storage virtualization and efficient VM management in modern cloud and datacenter environments. To support many NVMe devices and guarantee quality-of-service (QoS) at the same time, the current host sidecore approaches will continue to demand a considerable portion of host CPUs for storage virtualization [44]. Eventually, the number of VMs that can be supported within a single server will decrease, and the total datacenter costs will increase. Otherwise, the VMs will have serious performance problems due to the lack of computing resources. Also, with the limited capability of adding virtualization-dedicated CPU cores per server, the system will come to suffer from the low scalability.

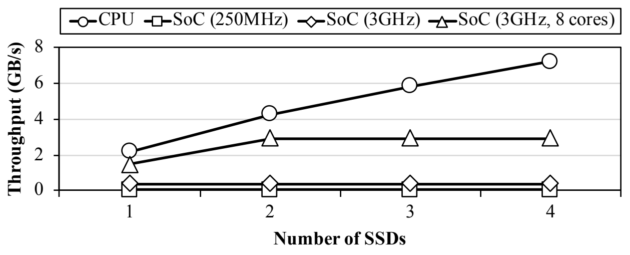


Figure 6: Performance comparison of different virtualization-dedicated core implementations

3.2 Weak Computing Power of SoC Cores

Modern on-device sidecore approaches offload the virtual device emulation to SoC cores embedded in peripheral devices instead of CPU cores to reduce the burden of host CPU [53]. However, their I/O performance can be severely bounded by the SoC cores' weak computing power. To measure the performance bottleneck due to the small cores, we implemented Microblaze softcores (250 MHz) [24] on an FPGA [23] and ran storage virtualization runtime software. We linearly scaled this performance result to evaluate more aggressive SoC designs that use higher clock speed and more cores (3 GHz, 8 cores). Figure 6 shows the random I/O performance of CPU (vhost-nvme) and SoC (runtime software) sidecore approaches with many NVMe devices. Our experimental results show that, even with the number of SoC cores increased, their weak computing capabilities become the significant performance bottleneck.

In particular, SoC sidecore designs significantly suffer from inefficient DMA mechanisms incurred by SW abstraction layers. For example, SmartNICs [2, 13], which utilize SoC cores in NICs, expose RDMA APIs instead of native DMA primitives, and it nearly doubles the DMA read/write latency [56]. Our Microblaze softcore implementation emulated the overhead by adding 5 μs per DMA transaction, and as a result, it achieved only 68% of the maximum performance of a single device. In addition, SoC sidecore designs cannot support a large number of virtual/physical devices and advanced storage management features due to their limited computing capabilities. As shown in Figure 6, an eight-core SoC cannot fully utilize two or more NVMe devices. These scalability issues will become more severe as storage devices get faster.

3.3 Absence of Interposition Layer

To save the host and on-device sidecores, hardware-assisted virtualization techniques can bypass the host software entirely. We experimentally confirmed that the two popular HW-assisted virtualization technologies in modern NVMe SSDs (i.e., passthrough and SR-IOV) provide the near-native performance in VMs. For this purpose, we installed a Samsung PM1733 SSD [17] which offers both passthrough and SR-IOV capabilities, and measured its FIO random I/O performance in VMs. We created a single VF through SR-IOV and assigned a 128-GB namespace, eight virtual queues, and eight virtual interrupt resources. When the device is connected through PCIe Gen3, we obtained around 800k IOPS for random reads and 250k IOPS for random writes in both passthrough and SR-IOV environments.

However, they suffer from the limited VM management and storage features in cloud environments. For example, SR-IOV does not support critical features to enable easy storage management such as live migration [41, 58] and seamless switching between different I/O channels. Also, it does not allow hypervisors to add critical features that are not natively provided by physical devices: replication [52, 61], snapshot [40, 70], record-replay, deduplication [68, 72], compression, encryption [63], metering, accounting, billing, and throttling [36, 46, 54] of guest I/O activities.

In addition, such hardware techniques enabling only the specific in-storage features significantly limit their portability and fungibility in modern datacenters. Furthermore, their fixed and vendor-specific storage functionalities do not provide enough flexibility to support advanced VM management. It is still challenging to provide flexibility and high performance at the same time with the current hardware-assisted virtualization schemes.

4 FVM Design and Implementation

This section introduces the design goals for fast, scalable, and flexible storage virtualization, and proposes our FVM solution to satisfy the goals. We describe our solution by presenting (1) a front-end implementation that emulates virtual devices and (2) a back-end implementation that directly manages physical devices.

4.1 Design Goals

We set the following design goals to resolve the challenges in modern storage virtualization: (1) A next-generation virtualization mechanism should ensure the near-native performance of NVMe storage devices. (2) It should minimize the amount of host resources used for virtualization so that a host machine can provide more computing power to VMs. (3) At the same time, it should nicely scale with the number of storage devices. (4) A physical storage device should be shared by multiple VMs. (5) It should not rely on hard-wired units to enable flexible and essential management functionalities, as summarized in Table 2. For example, software-based virtualization can implement flexible VM management features, while SR-IOV makes it hard for system administrators to guarantee accurate feature behaviors as in-SSD resource allocation and scheduling are done in a vendor-specific way.

Category	Features	SW	SR-IOV	FVM
Storage configuration	Consolidation	✔	✔	✔
	Aggregation	✔		✔
	Caching	✔		✔
Resource management	Isolation	✔	△	✔
	Throttling	✔	△	✔
Fault tolerance	Replication	✔	△	✔
	Snapshot	✔	△	✔
Data manipulation	Compression	✔	✔	✔
	Deduplication	✔	△	✔
	Encryption	✔	✔	✔
Administration	Migration	✔		✔
	Metering	✔	△	✔
	Billing	✔	△	✔

△: Limited to single-device use cases.

Table 2: Example VM management features in storage virtualization layers

4.2 FPGA-assisted Storage Virtualization

To meet the design goals, we propose FVM, a new hardware-assisted storage virtualization mechanism. The key idea of FVM is to implement *FVM-engine*, an FPGA-based virtualization acceleration card. We implement a storage virtualization layer and a device-control mechanism on FVM-engine.

In contrast to on-device SoC cores, an FPGA can be configured only with essential elements for storage virtualization and can take advantage of highly parallel NVMe protocols. Our FPGA-based solution can implement many cost-effective cores, and distribute virtual and physical NVMe queues and management routines to the cores. In this way, our solution achieves fine-grained parallel executions and scalable performance. In addition, our solution uses an FPGA's on-chip memory for SQ/CQ pairs and doorbell registers, which can be fast and directly accessed by VMs and NVMe devices through PCIe.

Another advantage of our FPGA-based solution is its programmability to implement new VM management features. Our FPGA-based solution has the potential of the hardware-based virtualization to solve the performance and efficiency challenges, while allowing to implement various VM management functionalities with its reconfigurability. In this work, we propose an FPGA-based virtualization layer, but it is also possible to implement the mechanism on ASICs. In such a case, an ASIC implementation can achieve higher performance by leveraging its optimized circuits for virtualization functions, but its flexibility for new storage management features will be limited.

Figure 7 shows the FVM architecture and its components. First, FVM bypasses host software stacks entirely and minimizes the use of host resources. Through the SR-IOV implementation on FVM-engine, VMs can enter a virtualization layer without any arbitration support from the host software. Moreover, its hardware-level NVMe interface makes the card directly manage the physical NVMe devices through PCIe.

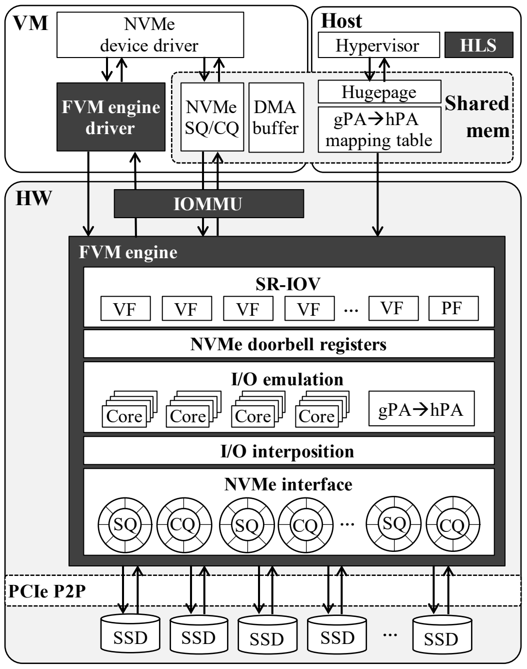


Figure 7: FVM architecture

Second, FVM is able to scale with many NVM devices by employing a parallel architecture for the device emulation. Instead of relying on on-device SoC cores, FVM-engine incorporates many specialized hardware units to poll and emulate guest I/O operations. Third, its HLS-based design flow enables flexible and programmable implementations for FVM-engine and other storage management services.

4.3 Front-end: VM-to-FVM-Engine

Direct FVM-engine assignment. FVM assigns virtual instances of FVM-engine to each VM through its SR-IOV interface. The current FVM-engine implementation integrates a PCIe IP block [28] to enable its own SR-IOV interface and supports up to four physical functions (PFs) and 252 virtual functions (VFs). The PFs are managed by host software for resource management, and each VF is assigned to a single VM exclusively for direct access to FVM-engine. Since all VFs have an identical PCIe configuration (e.g., PCIe BAR), VMs can install the same guest FVM-engine driver. FVM-engine also successfully isolates MMIO from different VMs by applying *non-overlapping* address translation to its internal address space (e.g., PCIe-to-AXI address translation [28]). At the same time, with the IOMMU support, FVM-engine can perform DMA transactions to guest memory space and inject an interrupt without a host software arbitration. To enable such *exitless* DMA transactions and interrupts, we install Linux *virtual function I/O* (VFIO) drivers in the host machine.

There are three major benefits of providing SR-IOV in FVM-engine. First, this design enables CPU-efficient virtual device emulation. All VMs can directly enter this hardware

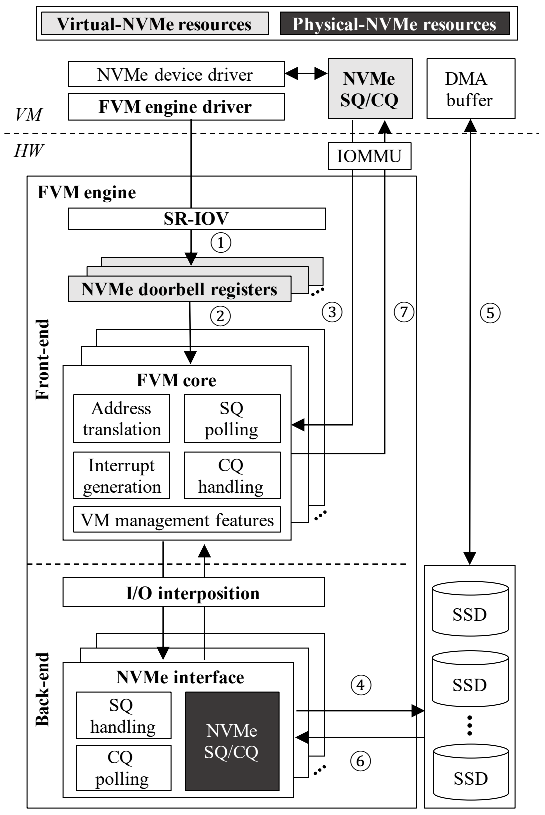


Figure 8: Hardware-level device emulation mechanism

interposition layer and handle interrupts without host software intervention. Second, it allows multiple VMs to share FVM-engine through 252 VFs. Using this interposition layer, FVM-engine can map virtual devices onto a much smaller set of physical NVMe devices. Third, it does not rely on fixed or vendor-specific storage capabilities. By simply deploying FVM-engine, any host machine can benefit from this virtualization mechanism.

Doorbell register remapping. FVM-engine reserves a memory space for NVMe doorbell registers and exposes it through PCIe BARs. When guest NVMe device drivers call `nvme_write_sq_db()` to submit I/O requests, the guest FVM-engine driver intercepts them and obtains their *(virtual) device id*, *SQ id*, and *SQ tail* information. The guest FVM-engine driver then calculates the address of the target doorbell register and writes the received SQ tail pointer to FVM-engine (Figure 8–①). In this way, the guest OS can indicate new NVMe commands to be executed. Similarly, to notice that the command completions are normally handled, FVM-engine driver intercepts `nvme_process_cq()` function and acquires *(virtual) device id*, *CQ id*, and *CQ head* information. It then writes the received CQ head pointer to the target address in the FVM-engine doorbell regions.

Virtual I/O queue emulation. To process a guest I/O request at the hardware layer, FVM-engine polls doorbell registers using multiple FVM cores (Figure 8–②). Algorithm 1 demonstrates its polling routine to emulate virtual NVMe devices.

Algorithm 1: Polling function in the front-end for I/O submission

```
1  fvm_nvme_submit ($devices_{virtual}$);
2  while $true$ do
       /* Iterate over the assigned virtual
          devices and their SQs                    */
3      foreach $vdev \in devices_{virtual}$ do
4          foreach $sq \in available\_sqs(vdev)$ do
               /* Poll the doorbell registers mapped
                  in FVM-engine                    */
5              $tail = get\_tail(sq)$
               /* Find newly added NVMe commands
                  from the guest OS                */
6              $head = get\_head(sq)$
7              while $tail \neq head$ do
8                  $cmd = get\_cmd(sq, head)$
9                  $cmd = manipulate\_cmd(cmd)$
10                 $submit\_cmd(cmd)$
11                 $head = (head + 1)\%SQ\_SIZE$
12             end
13             $set\_head(sq, head)$
14         end
15     end
16 end
```

First, the FVM core gets the newly updated `SQ tail` (line 5) and compares it with the `SQ head` that stores the previous `tail` value (line 6). The difference between these two values indicates the number of commands that are submitted by the guest OS. Since FVM-engine reserves its doorbell memory regions using on-chip memory (e.g., BRAM [29]), it can quickly poll those regions. This design can easily scale up the number of VMs as modern FPGAs currently support tens of MBs on-chip memory [23].

To enable FVM-engine to access a submission queue in the guest memory space, we utilize an internal DMA engine [28] and an IOMMU. When VMs install guest NVMe drivers, they deliver SQ/CQ gPAs to FVM-engine. FVM cores then use these addresses to directly read the submitted commands through the DMA engine (Figure 8–③). Since FVM guarantees exitless DMA transactions with an IOMMU and VFIO drivers, each virtual instance of FVM-engine can safely access target SQ/CQ pairs allocated in the guest memory space without software intervention.

Similarly, to deliver NVMe completion entries to a VM, FVM-engine directly writes the completion messages to the guest CQ memory region (Figure 8–⑦). In addition, it triggers an interrupt to the guest directly through the interrupt remapping engine. The FVM-engine driver then forwards the interrupt with an associated IRQ vector to the NVMe driver and allows it to handle the received completions.

PRP and LBA translation. FVM-engine processes the received NVMe commands from VMs before submitting them to physical NVMe devices. Specifically, FVM-engine manipulates *physical region page* (PRP) entries (pointing guest DMA buffers) from gPAs to hPAs. To enable such gPA-to-hPA translation at the hardware level, FVM leverages *hugepages* to allocate pinned memory [4, 20]. Since the current operating system does not change their physical locations, FVM-engine can statically translate PRP entries by incorporating the gPA-to-hPA mapping table. The translation does not incur any performance overhead in this design as FVM-engine manages the mapping table using its on-chip memory. Also, due to the hugepages (2MB), the required table size is small enough to keep them in the on-chip memory (i.e., 4KB table to cover 1GB guest memory space).

In addition, FVM-engine needs to manipulate a *start logical block address* (SLBA) to allocate separate block regions of physical devices to VMs. Since the current implementation of FVM assumes a *static* partition, the SLBA in guest NVMe command can be simply modified by applying a different offset value, which is managed by host software.

Virtual admin queue emulation. FVM manages a virtual NVMe *admin* SQ/CQ pair through QEMU and SPDK vhost-target implementations. Since QEMU and KVM can track VM exits caused by MMIO on administration doorbell registers, they are still able to interact with SPDK vhost-target via a UNIX domain socket. QEMU delivers critical administration commands (e.g., I/O queue creation, deletion, shutdown) to the SPDK vhost-target implementation following the conventional vhost-target protocol.

4.4 Back-end: FVM-Engine-to-SSD

Physical SQ/CQ remapping. To allow FVM-engine to interact with physical NVMe devices directly, host software remaps their NVMe I/O queues onto FVM-engine's PCIe BAR regions. At the installation time, the host FVM-engine driver provides the memory-mapped region's address to the physical NVMe devices. The NVM devices are unaware of FVM-engine, but a PCIe switch delivers DMA transactions to FVM-engine seamlessly. Also, our experimental result demonstrates that FVM-engine can fully utilize a single Intel Optane SSD with eight SQ/CQ pairs (4KB each queue). Thus, FVM-engine can nicely scale with a large number of physical devices and VMs without any on-chip memory space issue for these remapped queues.

Direct NVMe device-control mechanism. FVM-engine incorporates standard NVMe interfaces to implement a direct device-control mechanism. (1) FVM-engine moves the NVMe commands to the submission queue in the FVM-engine on-chip memory. (2) FVM-engine then rings doorbell registers located in the NVMe device to notify the number of newly submitted commands. (3) The NVMe controller fetches the NVMe commands through PCIe P2P communications (Figure 8–④). (4) After the NVMe device processes the commands (Figure 8–⑤), it writes the command completions to the FVM-engine address space (Figure 8–⑥). (5) FVM-engine processes them and (6) rings doorbell registers located in the NVMe device.

Polling CQs. To immediately handle completions from physical devices, an NVMe interface polls its CQ memory space. Algorithm 2 shows its polling function. First, the NVMe interface handles a CQ entry pointed by its `head` pointer (line 7) and compares its `phase` bit with the current round (line 9). This enables the NVMe interface to determine whether a new entry was posted as a part of the previous or current round of completion notifications. After that, it processes the completion entries (line 10) and forwards them to the front-end (line 11). The FVM core then writes completion messages to the guest CQ memory space. Since FVM-engine manages all SQ/CQ pairs using the on-chip memory, its polling routine does not incur any performance overhead.

Algorithm 2: Polling function in the back-end for I/O completion

```
1  fvm_nvme_complete ($devices_{physical}$);
2  while $true$ do
       /* Iterate over the assigned physical
          devices and their CQs                    */
3      foreach $pdev \in devices_{physical}$ do
4          foreach $cq \in available\_cqs(pdev)$ do
5              $head = get\_head(cq)$
6              $cq\_phase = get\_cq\_phase(cq)$
               /* Poll the completion entries mapped
                  in FVM-engine                    */
7              $cpl = get\_cpl(cq, head)$
8              $cpl\_phase = get\_cpl\_phase(cpl)$
               /* Find newly added NVMe completions
                  from the physical device         */
9              while $cpl\_phase == cq\_phase$ do
10                 $cpl = manipulate\_cpl(cpl)$
11                 $forward\_cpl(cpl)$
12                 $head = (head + 1)\%CQ\_SIZE$
13                 if $head == 0$ then
14                     $cq\_phase = invert\_phase(cq\_phase)$
15                 end
16                 $cpl = get\_cpl(cq, head)$
17                 $cpl\_phase = get\_cpl\_phase(cpl)$
18             end
19             $set\_head(cq, head)$
20             $set\_cq\_phase(cq, cq\_phase)$
21         end
22     end
23 end
```

4.5 FVM Core Design

FVM maximizes the opportunities of its hardware-level virtualization mechanism by instantiating multiple FVM cores to poll and emulate guest I/O. This design choice can offer more

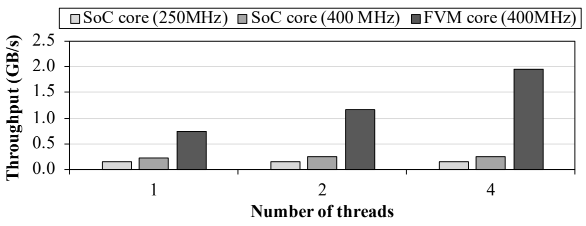


Figure 9: Performance comparison of different virtualization processing core implementations

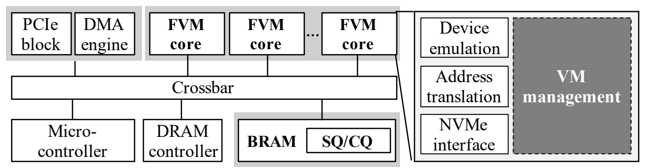
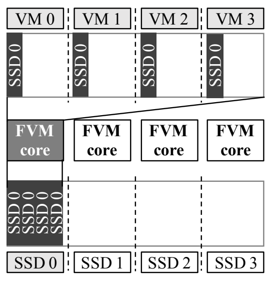


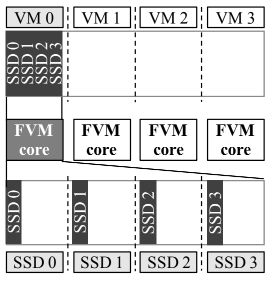
Figure 10: Multiple FVM cores with crossbar interconnect

scalable performance than an on-device SoC core as it replaces the general-purpose sidecores with the customized hardware units. By doing so, FVM can address the performance bottleneck in the processing cores. Figure 9 shows the performance difference between *single-core* SoC and *single-FVM core* implementations. To measure the performance bottleneck on the SoC cores, we implemented runtime software on Microblaze softcores [24] and ran FIO random read benchmarks with the increasing number of threads. We projected this performance result for the higher clock speed (400 MHz) to fairly compare it with our FVM core implementation running at the same clock frequency. Our experimental result demonstrates that the current FVM core implementation achieves 8× higher throughput than the softcore implementations.

Figure 10 shows an example system architecture using multiple FVM cores for storage virtualization. First, to reduce the burden on users in building and integrating system functions required for interacting between VMs and NVMe devices, we separate the virtualization logic (storage service) from the common I/O (BRAM, crossbar) and the board-specific logic (DRAM, PCIe). Second, by interconnecting them with crossbars, FVM-engine can be extended to support a multi-core



(a) Multi-VM support



(b) Multi-SSD support

Figure 11: Multi-VM and multi-SSD support with FVM cores

Designs	LUTs	Registers	BRAMs	Clock speed
1 core	4682 (0.36%)	7528 (0.29%)	5.5 (0.29%)	400 MHz
1 VM - 6 cores (5 SSDs)	106809 (8.19%)	130064 (4.99%)	362.5 (17.98%)	400 MHz
4 VMs - 6 cores (4 SSDs)	103539 (7.94%)	131394 (5.04%)	359.5 (17.83%)	400 MHz
4 VMs - 2 cores (1 SSD)	80259 (6.16%)	93997 (3.61%)	321.5 (15.95%)	400 MHz

Table 3: FPGA resource utilization for different FVM configurations

design. Our crossbar interconnection can have 16 input/output ports and can be connected with other switches for higher scalability. As each crossbar switch takes tens of nanoseconds, the overall switching latency is negligible compared to modern SSD's microsecond-scale access latency.

In addition, FVM-engine can be configured to support various FVM core mapping strategies. For example, Figure 11 shows two different mapping strategies. Figure 11a demonstrates that a single FVM core is shared by multiple VMs, while it is dedicated to a single NVMe device. This design can easily cover an increasing number of VMs. On the other hand, Figure 11b shows that an FVM core is dedicated to a single VM, while it covers multiple physical NVMe devices. With this mapping strategy, we can allocate more virtualization resources to more performance-critical VMs.

As Table 3 shows, FVM can cost-effectively scale with multiple FVM cores without sacrificing its clock speed. We implemented three different FVM configurations depending on the number of VMs and SSDs that can be supported. The 1-VM and 6-FVM-core implementation supports five NVMe SSDs and utilizes 8.19% LUTs, 4.99% registers, and 17.98% BRAMs in the FPGA chip. Also, its light resource usage (< 0.5% for a single FVM core) provides opportunities to utilize the remaining resources to implement more FVM cores, and/or deploy much cheaper FPGA boards to minimize the FPGA costs. In addition, FPGAs and FVM cores can be more easily added/upgraded on servers, which provides higher scalability using expandable slots than CPU cores requiring extra sockets.

4.6 HLS

FVM enables flexible and easily programmable implementations through its high-level synthesis (HLS)-based design flow. Modern HLS supports high-level languages and has become a standard hardware design flow for FPGAs. Our HLS-based FVM implementation allows users to extend their designs easily.

In this work, we implemented five different storage functions on FVM-engine First, we implemented device sharing, which allows multiple VMs to share a single NVMe device. Second, we designed a token-based throttling mechanism to effectively manage guest I/O operations. Third, we implemented replication to achieve fault tolerance. Fourth, we

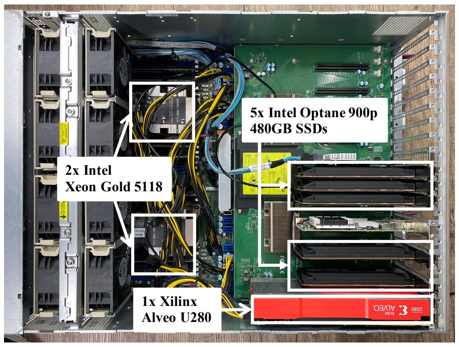


Figure 12: FVM hardware prototype

implemented server-side caching to accelerate storage accesses from VMs. Fifth, we designed direct copying in which a guest OS moves data between two different storage devices using FVM-engine. For this purpose, the VM utilizes FVM-engine's internal memory space as an intermediate buffer, while bypassing the entire software stacks.

5 Evaluation

In this section, we evaluate our FVM implementation and compare its random I/O and RocksDB performance with other storage virtualization schemes. We also present five example VM management features implemented through our HLS-based design flow.

5.1 Experimental Setup

To evaluate FVM, we ran FIO [5] random I/O benchmarks and RocksDB [16] workloads on VMs. We evaluated our FVM implementation against its native execution and existing virtualization mechanisms including SPDK vhost-nvme v20.01 [21] (configured with the option `-with-internal-vhost-lib`) and *passthrough*. Since the passthrough technique avoids most of the virtualization software stack and directly assigns the device to the VM, it can provide the near-native execution performance. As FVM's use cases, we also implemented five different storage services (device sharing, throttling, replication, caching, and direct copy) based on our HLS-based design flow, and validated them with respect to the software reference implementations.

Figure 12 shows our hardware FVM prototype. We built this prototype on a host machine (Super Micro SuperServer 4029GP-TRT2) with two 12-core Intel Xeon Gold 5118 CPUs running at 2.3GHz, 256GB DDR4 DRAM, and five 480GB Intel Optane 900P NVMe SSDs. The Optane SSD (based on the 3D XPoint NVM technology) can support up to 550k IOPS in random-read and 500k IOPS in random-write with 10 μs latency [7]. We implemented FVM-engine on a Xilinx Alveo U280 Data Center Accelerator Card using Vivado and Vivado HLS v2019.2 EDA tools. We configured the PCIe IP block to meet the PCIe Gen3 x4 specification and connected it with other NVMe SSDs through PCIe Gen3 x16 lanes. For accurate performance measurements, we disabled hyperthreading and dynamic voltage and frequency scaling (DVFS).

On the software side, we installed 64-bit Ubuntu 18.04 with the Linux kernel version 5.3.0 and QEMU emulator version 3.0.0 on the host machine. We installed the same OS and Linux kernel versions on VMs, and implemented an FVM-engine Linux device driver. We modified an SPDK vhost-target implementation and applied FVM to an existing QEMU/KVM virtualization system.

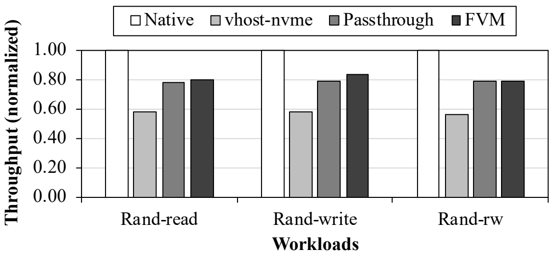


Figure 13: Random I/O throughput with two NVMe SSDs

5.2 Performance

5.2.1 Random I/O Benchmark

To evaluate the random I/O performance, we ran FIO with two SSDs and measured (1) the maximum achievable throughput, (2) latency, and (3) CPU utilization. For passthrough and FVM, we allocated four CPU cores and 1GB system memory per VM. To show the performance impact due to the lack of host resources in vhost-nvme, we allocated one CPU core for the vhost-nvme virtualization layer and three cores for the VM.

Figure 13 shows the relative throughput of 4KB random read, write and read/write (50% of read and write each) for native, SPDK vhost-nvme, passthrough, and FVM. For all three random I/O benchmarks, passthrough and FVM can achieve about 79% (2.65GB/s on average) of native performance (3.36GB/s on average). However, SPDK vhost-nvme achieves about 58% (1.95GB/s on average) due to the CPU resource competition between VMs and the vhost-nvme virtualization layer.

In this experiment, we observed that other virtualization overheads still prevent even the passthrough and FVM from achieving the full native performance. First, passthrough and FVM include VM exits caused by MSR_WRITE and HLT instructions to manage timer interrupts and to yield CPU resources to a host machine. Second, they involve IOMMU's address translation to transfer data to and from NVMe storage directly (passthrough) or to manage guest OSes' SQ/CQ pairs from FVM-engine (FVM). There have been efforts to mini-

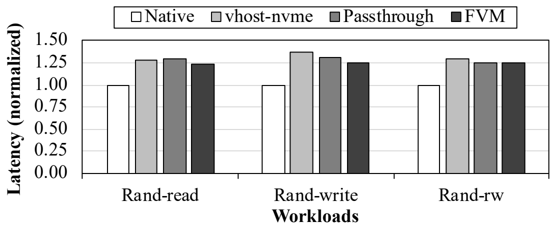


Figure 14: Random I/O latency with two NVMe SSDs

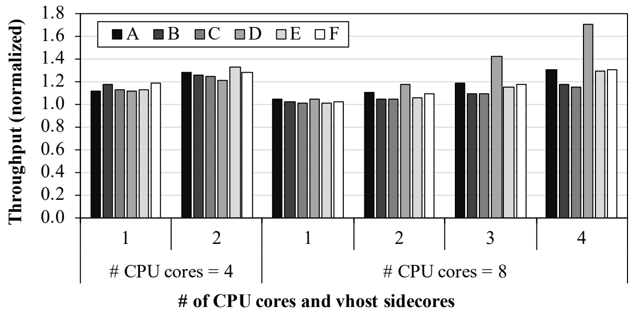


Figure 15: RocksDB throughput with FVM (normalized to SPDK vhost-nvme)

mize the overheads [34, 57]. In this work, we do not address the overhead as they are orthogonal to our work.

In addition, we can see that FVM performs better than passthrough in some cases. Our current FVM-engine implementation aggregates completions from those two SSDs and delivers the smaller number of interrupts to the VM, which provides more CPU resources to random I/O operations.

Figure 14 shows the average latency normalized to that of the native execution for the FIO experiments. For all three workloads, FVM outperforms vhost-nvme and passthrough thanks to the fast FVM core design and the direct device-control mechanism through PCIe P2P communications.

5.2.2 RocksDB

To evaluate a server workload on FVM, we ran RocksDB [16] on the EXT4 file system and YCSB [38] to generate workloads. We configured YCSB to generate the workloads as follows: (A) 50% of read and write each, (B) 95% of read and 5% of write, (C) read-only, (D) read-latest (most reads access the last write), (E) short-ranges (most reads access recent writes), and (F) read-modify-writes. We scaled up RocksDB's `recordcount` and `operationcount` parameters to highlight its I/O activities.

For SPDK vhost-nvme, we considered various CPU allocation scenarios and increased the portion of dedicated sidecores up to 50% to emulate future high-performance NVMe devices. For this purpose, we assigned 1 – 4 CPU cores (out of 4 or 8) for SPDK vhost-target and the remaining CPU cores for the VM. For FVM, we assigned four or eight CPU cores for the VM.

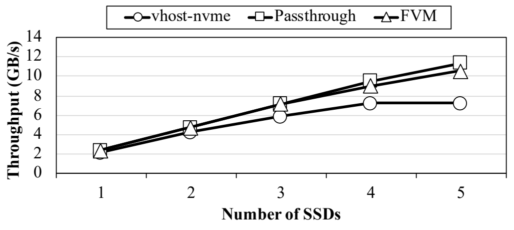


Figure 16: I/O throughput with multiple SSDs

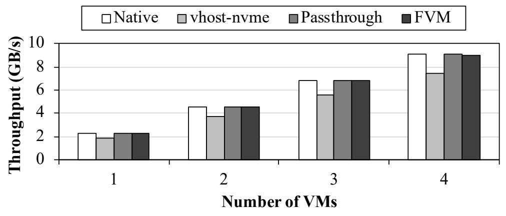


Figure 17: Multi-VM throughput with multiple SSDs

Figure 15 shows the operation throughput of FVM, normalized to that of SPDK vhost-nvme. Since FVM saves host CPU resources to provide more computing power to VMs, it obtains $1.20\times$ (average) higher and $1.33\times$ (maximum) higher throughput than vhost-nvme with four CPU cores. On the other hand, with eight total CPU cores, FVM achieves $1.15\times$ (average) higher and $1.71\times$ (maximum) higher throughput than vhost-nvme. With these trends, FVM will become more promising as the storage devices get faster in future.

5.3 Scalability

For the scalability test, we installed one VM with 18 CPU cores to fully utilize five NVMe SSDs and measured the aggregate throughput. For vhost-nvme, we assigned four cores to SPDK vhost-target and 14 cores to the VM. Figure 16 shows the total throughput that a single VM can achieve with the given number of SSDs. As the number of SSDs increases, both passthrough and FVM scale nicely, while vhost-nvme does not scale well due to its excessive CPU usage to emulate the guest I/O operations in the hypervisor.

When the VM utilizes more than four SSDs, FVM's total throughput is 7% lower compared to that of passthrough. Because the current PCIe IP core [25] supports only eight interrupt vectors per VF, the FVM-engine device driver installed in the guest OS should make the interrupt vectors shared by many SQ/CQ pairs and look up multiple CQs to identify completion messages.

Next, we assigned four CPU cores and one SSD to each VM and ran four VMs concurrently. For vhost-nvme, we assigned one CPU core to SPDK vhost-target and three cores to the VM. Figure 17 shows the total achievable throughput of each virtualization implementation as the number of VMs increases. The results show that FVM scales well as the number of VM increases. With four VMs, FVM achieves up to 9.5

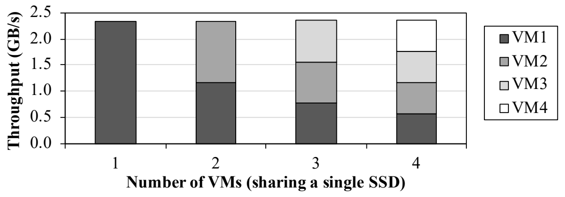


Figure 18: Device sharing with balanced allocation

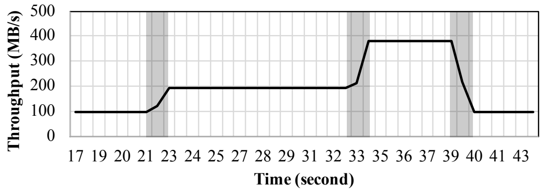


Figure 19: SSD throughput trace with token-based throttling

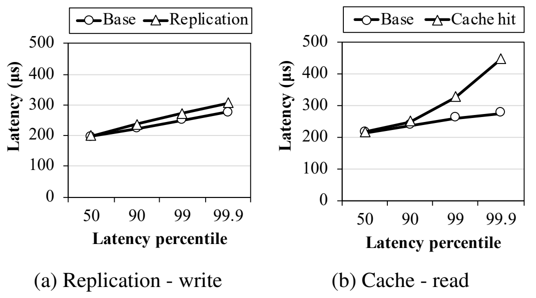


Figure 20: Tail latency of replication and caching

GB/s by processing on FVM-engine. However, vhost-nvme fails to achieve the full native throughput, due to its software overhead.

5.4 Programming Example Functions

To evaluate FVM's flexibility, we implemented five example storage functions in FVM's hardware-level virtualization layer: (1) device sharing, (2) throttling, (3) replication, (4) caching, and (5) direct copy. To implement these functions, we added and modified only 10s-100s of C++ code lines (i.e., device sharing: 40 LOC, throttling: 70 LOC, replication: 15 LOC, caching: 220 LOC, direct copy: 570 LOC).

Device sharing. To implement the device sharing functionality, we mapped multiple virtual I/O queue pairs from different VMs to a single physical NVMe queue pair (similar to SPDK vhost-nvme implementations). To correctly arbitrate completion messages from the physical device, we made an additional data structure to keep track of the virtual device identifications of the submitted NVMe commands which requires 64 bytes for each physical I/O queue. We also deployed a round-robin method between virtual I/O queues as the vhost-nvme implementation does. Figure 18 shows the FIO throughput results of FVM when running multiple VMs on a single NVMe SSD. FVM achieves the perfectly balanced throughput allocation among VMs without any performance loss.

Throttling. We implemented a token-based throttling algorithm on FVM, which can limit the bandwidth with periodically refilled tokens and a bucket which can save a certain amount of tokens. The FVM-engine driver configures the period of `refill_token` signal, the amount of tokens to be refilled in a period, and the size of the bucket. An FVM core periodically polls the `refill_token` signal and filters every command by checking the size of the request and the amount of remaining tokens. If the command is issued, a proper amount of tokens are removed from the bucket. Figure 19 shows the aggregate bandwidth within a time interval of the FIO benchmark while throttling the I/O operations on FVM-engine. We configured the FIO benchmark to achieve the maximum bandwidth of an SSD (2GB/s), and we throttled the I/O from 100MB/s to 400MB/s. The figure shows that the bandwidth is stable and limited as configured.

Replication. By seamlessly replicating I/O operations from VMs, a server with FVM-engine can achieve fault tolerance. To enable this feature, we assigned multiple physical NVMe SSDs to a single virtual storage device. When an FVM core receives a write command from a VM, it replicates the commands and broadcasts them to the physical devices. The FVM core then waits for completion messages from all physical storage devices assigned to the virtual device before sending corresponding completions to the target VM. Figure 20a shows its tail latency results of 4KB random writes through FVM. As an FVM core should replicate NVMe commands and wait completions from all the NVMe SSDs, the replication feature adds the extra latency compared to the baseline FVM implementation.

Caching. To enable caching, we implemented a hash table that has 256k entries in the FPGA's 4MB on-chip memory space. Each table entry contains a start logical block address (SLBA) and a corresponding cached block address (CBA). If an FVM core finds a valid entry in the hash table, it replaces the received SLBAs with CBAs and submits them to the NVMe devices. We measured the tail latency of 4KB random reads with FVM and its caching mechanism, while emulating a perfect cache hit ratio. Figure 20b shows that the caching mechanism increases the tail latency due to the increased number of contentions on FVM-engine's interconnection resources for the hash table accesses.

Direct copy. FVM can enable a direct device-to-device (D2D) data copy feature, while bypassing the host CPU and memory. Using the feature, a server with FVM-engine can perform intra-VM or inter-VM data transfers efficiently. We implemented a direct-copy feature on FVM leveraging its hardware-based device-control mechanism. When FVM-engine receives a request, it splits the bulk data transfer into multiple smaller-sized requests. It then generates NVMe commands for each

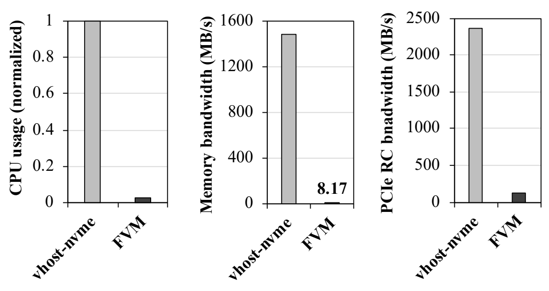


Figure 21: Resource usage of direct copying

split request and submits them using FVM-engine's NVMe interface. To bypass the host memory, the NVMe commands utilize the FPGA's on-chip memory as an intermediate data buffer. Figure 21 shows the CPU usage, host memory bandwidth, and PCIe root complex (RC) bandwidth usage while performing a 32GB inter-SSD bulk data transfer. This figure shows that the data transfer controlled by FVM provides high bandwidth without consuming the host resources.

6 Discussion

Cost analysis. FVM can be used for the cost saving, as it minimizes the number of the required CPUs and servers for the target storage virtualization. For example, for a server with 24 NVMe devices, FVM can reduce the CPU core usage by up to 30% (i.e., saving 20 cores in a 64-core machine). Based on the current prices on major online stores (e.g., Amazon) and vendor websites, FVM can save \$2000–\$6400 (\$100–\$320 per core [9,10]) for 64-core machines. If we consider the trend of increasing the number and performance of storage devices per server, the cost saving will be even more significant. In addition, as our wimpy FVM core uses very small FPGA resources (< 0.5%), we can implement FVM on the cost-effective FPGA boards (e.g., \$3000 for Alveo U50 [3], \$1500–\$3500 for evaluation boards).

FVM-engine scalability. The scalability bottleneck can occur if an FPGA is short of resources and/or communication takes too long. On our FPGA, we can implement around 140 wimpy FVM cores (0.5% per FVM core) including multi-level crossbar networks (2.5% per crossbar module) and hundreds of SQ/CQ pairs (5KB per pair). Our crossbar switch can have 16 input/output ports and can be connected with other switches. As each crossbar switch takes 24 ns (6 cycles, 250MHz), the overall switching latency is negligible compared to modern SSD's tens of microsecond access latency.

Supporting other storage protocols. One of the biggest benefits of using programmable FPGAs is to provide various storage protocols as needed. The FPGA-based virtualization layer can easily implement a new interface to activate advanced features in modern storage devices. For example, it can easily support standardized key-value store (KV) acceleration extensions [43] by reprogramming the FPGA according to their interface specifications.

7 Related Work

NVMe virtualization. NVMe virtualization requires a special mechanism to make full use of its parallel and high-performance storage protocol. SPDK [20] is a user-space library for high-performance and scalable storage applications. It integrates all the necessary drivers into the user space to avoid system calls and enable zero-copy access from the applications. In addition, it adopts polling to monitor I/O completions instead of relying on interrupts. Specifically, SPDK vhost-nvme [69] extends the SPDK library to provide virtual NVMe controllers to QEMU-based VMs. Similarly, MDev-NVMe [59] provides a mediated passthrough mechanism in kernel space with an active polling mode.

Direct device-control mechanism. A direct device-control mechanism at the hardware level provides fast and resource-efficient I/O paths. For example, device-centric server (DCS) and its direct device-control method [32, 49] implement a device orchestration scheme on an FPGA to enable fast device-to-device direct data communications. In this way, DCS can enable hardware-offloaded direct data transfers between NVMe SSDs and network adapters through PCIe P2P. As another example, GPUDirect Async [31] enables a direct data transfer between GPUs and NICs to free CPUs from the control path, while moving data between GPUs and NICs. Lynx [64] offloads the server data and control planes to a SmartNIC, and enables direct networking from accelerators via a lightweight hardware-friendly I/O mechanism. It enables to develop hardware-accelerated network servers which do not require much CPU involvement.

8 Conclusion

In this work, we present FVM, a new hardware-assisted storage virtualization mechanism. The key idea is to implement (1) a storage virtualization layer on an FPGA card (FVM-engine) decoupled from the host resources and (2) a device-control method to have the card directly manage the physical storage devices. In this way, a server equipped with FVM-engine achieves high performance, scalability, and flexibility by saving the invaluable host-side resources and by adding the decoupled VM management-efficient FPGA cards as needed.

Acknowledgments

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-12. We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

References

[1] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[2] Broadcom Stingray SmartNIC Adapters. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[3] DigiKey A-U50DD-P00G-ES3-G. https://www.digikey.com/en/products/detail/xilinx-inc/A-U50DD-P00G-ES3-G/10642492.

[4] DPDK. https://www.dpdk.org/.

[5] Flexible I/O Tester. https://github.com/axboe/fio.

[6] Google Cloud Computing Services. https://cloud.google.com/.

[7] Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series.html.

[8] Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[9] Intel® Xeon® Gold 5118 Processor. https://ark.intel.com/content/www/us/en/ark/products/120473/intel-xeon-gold\-5118-processor-16-5m-cache-2-30-ghz.html.

[10] Intel® Xeon® Processor E5-4669 v4. https://ark.intel.com/content/www/us/en/ark/products/93805/intel-xeon-processor\-e5-4669-v4-55m-cache-2-20-ghz.html.

[11] Linux KVM. https://www.linux-kvm.org/.

[12] Microsoft Azure Cloud Computing Services. https://azure.microsoft.com/en-us/.

[13] NVIDIA Mellanox BlueField-2 DPU. https://www.mellanox.com/products/bluefield2-overview.

[14] NVM Express. https://nvmexpress.org/.

[15] QEMU. https://www.qemu.org/.

[16] RocksDB - A Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org/.

[17] Samsung PM1733 NVMe SSD. https://www.samsung.com/semiconductor/ssd/enterprise-ssd/MZWLJ3T8HBLS-00007/.

[18] Samsung Z-SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[19] Single-Root Input/Output Virtualization. http://www.pcisig.com/specifications/.

[20] SPDK. https://spdk.io/.

[21] SPDK I/O Virtualization with Vhost-user. https://spdk.io/doc/vhost_processing.html.

[22] Super Micro Computer, SuperServer, 4029GP-TRT2. https://www.supermicro.com/en/products/system/4U/4029/SYS-4029GP-TRT2.cfm.

[23] Xilinx Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[24] Xilinx MicroBlaze Soft Processor Core. https://www.xilinx.com/products/design-tools/microblaze.html.

[25] Xilinx QDMA Subsystem for PCI Express. https://www.xilinx.com/products/intellectual-property/pcie-qdma.html.

[26] AMD I/O Virtualization Technology (IOMMU) Specification, Rev 1.26. 2009.

[27] Intel® Virtualization Technology for Directed I/O, Rev 1.3. 2011.

[28] Xilinx QDMA Subsystem for PCI Express v3.0. 2019.

[29] Xilinx UltraScale Architecture Memory Resources v1.11. 2020.

[30] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3), 2006.

[31] Elena Agostini, Davide Rossetti, and Sreeram Potluri. Offloading communication control logic in gpu accelerated applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 248–257. IEEE, 2017.

[32] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. Dcs: a fast and scalable device-centric server architecture. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 559–571. IEEE, 2015.

[33] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[34] Andrea Arcangeli. Micro-optimizing kvm vm-exits. In *KVM Forum*, 2019.

[35] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 355–369, 2016.

[36] Peter A Balinski, Sasikanth Eda, Ashwin M Joshi, John T Olson, and Sandeep R Patil. Dynamic i/o throttling in a storlet environment, March 10 2020. US Patent 10,585,596.

[37] Deepavali Bhagwat, Mahesh Patil, Michal Ostrowski, Murali Vilayannur, Woon Jung, and Chethan Kumar. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 287–300, 2015.

[38] Erwin Tam Raghu Ramakrishnan Brian F. Cooper, Adam Silberstein and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154. IEEE, 2010.

[39] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 329–344, 2017.

[40] Hoi Chan and Trieu Chieu. An approach to high availability for cloud servers with snapshot mechanism. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, pages 1–6, 2012.

[41] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[42] NVM Express. NVM Express revision 1.3 specification. page 220, 2017.

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

[44] Jim Harris. Accelerating NVMe-oF* for VMs with the Storage Performance Development Kit. In *Flash Memory Summit*, 2017.

[45] Asias He. Virtio-blk Performance Improvement. In *KVM Forum*, 2012.

[46] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, 2017.

[47] Masaki Kimura. Better Utilization of Storage Features from KVM Guest via virtio-scsi. In *LinuxCon and CloudOpen North America*, 2013.

[48] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual remote i/o. *ACM SIGARCH Computer Architecture News*, 44(2):49–65, 2016.

[49] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. Dcs-ctrl: a fast and flexible device-control mechanism for device-centric server architecture. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 491–504. IEEE, 2018.

[50] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[51] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. Splitx: Split guest/hypervisor execution on multi-core. In *WIOV*, 2011.

[52] Emmanuel S Levijarvi and Ognian S Mitzev. Private cloud replication and recovery, January 6 2015. US Patent 8,930,747.

[53] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[54] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. Pslo: Enforcing the xth percentile latency and throughput slos for consolidated vm storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–14, 2016.

[55] Anthony Liguori. The Nitro Project – Next Generation AWS Infrastructure. In *Hot Chips: A Symposium on High Performance Chips*, 2018.

[56] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[57] David Matlack. Kvm message passing performance. In *KVM Forum*, 2015.

[58] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX Annual technical conference, general track*, pages 391–394, 2005.

[59] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. Mdev-nvme: a nvme storage virtualization solution with mediated pass-through. In *2018 USENIX Annual Technical Conference (ATC 18)*, pages 665–676, 2018.

[60] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[61] Yossi Saad, Assaf Natanzon, and Yedidya Dotan. Securing data replication, backup and mobility in cloud storage, October 6 2015. US Patent 9,152,578.

[62] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE, 2008.

[63] Uma Somani, Kanika Lakhani, and Manish Mundra. Implementing digital signature with rsa encryption algorithm to enhance the data security of cloud in cloud computing. In *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010)*, pages 211–216. IEEE, 2010.

[64] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–131, 2020.

[65] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. Srcmap: Energy proportional storage using dynamic consolidation. In *FAST*, volume 10, pages 267–280, 2010.

[66] Carl Waldspurger and Mendel Rosenblum. I/O Virtualization. *Communications of the ACM*, 55(1):66–73, 2012.

[67] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[68] Jiwei Xu, Wenbo Zhang, Shiyang Ye, Jun Wei, and Tao Huang. A lightweight virtual machine image deduplication backup approach in cloud environment. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 503–508. IEEE, 2014.

[69] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76. IEEE, 2018.

[70] Lei Yu, Chuliang Weng, Minglu Li, and Yuan Luo. Snpdisk: an efficient para-virtualization snapshot mechanism for virtual disks in private clouds. *IEEE Network*, 25(4):20–26, 2011.

[71] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.

[72] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *2010 IEEE international conference on cluster computing*, pages 88–96. IEEE, 2010.