LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs

Huaicheng Li, Mingzhe Hao
University of Chicago

Stanko Novakovic
Microsoft Research

Vaibhav Gogte
University of Michigan

Sriram Govindan
Microsoft

Dan R. K. Ports, Irene Zhang, Ricardo Bianchini
Microsoft Research

Haryadi S. Gunawi
University of Chicago

Anirudh Badam
Microsoft Research

Abstract

Today's cloud storage stack is extremely resource hungry, burning 10-20% of datacenter x86 cores, a major "storage tax" that cloud providers must pay. Yet, the complex cloud storage stack is not completely offload-ready to today's IO accelerators. We present LeapIO, a new cloud storage stack that leverages ARM-based co-processors to offload complex storage services. LeapIO addresses many deployment challenges, such as hardware fungibility, software portability, virtualizability, composability, and efficiency. It uses a set of OS/software techniques and new hardware properties that provide a uniform address space across the x86 and ARM cores and expose virtual NVMe storage to unmodified guest VMs, at a performance that is competitive with bare-metal servers.

CCS Concepts. • **Computer systems organization → Cloud computing**; **Client-server architectures**; **System on a chip**; *Real-time system architecture.*

Keywords. Data Center Storage; ARM SoC; NVMe; SSD; Virtualization; Performance; Hardware Fungibility

ACM Reference Format:
Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. 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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378531

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378531

1 Introduction

Cloud storage has improved drastically in size and speed in the last decade, with a market size expected to grow to $88 billion by 2022 [11]. With this growth, making cloud storage efficient is paramount. On the technical side, cloud storage is facing two trends, the growing complexity of cloud drives and the rise of IO accelerators, both unfortunately have not blended to the fullest extent.

First, to satisfy customer needs, today's cloud providers must implement a wide variety of storage (drive-level) functions as listed in Table 1. Providers must support both local and remote isolated virtual drives with IOPS guarantees. Users also demand drive-level atomicity/versioning, and not to mention other performance, reliability, and space-related features (checksums, deduplication, elastic volumes, encryption, prioritization, polling for ultra-low latencies, striping, replication, etc.) that all must be composable. Last but not least, future cloud drives must support fancier interfaces [19, 24, 27, 63, 70].

As a result of these requirements, the cloud storage stack is extremely resource hungry. Our experiments suggest that the cloud provider may pay a *heavy tax* for storage: 10–20% of x86 cores may have to be reserved for running storage functions. Ideally, host CPU cores are better spent for providing more compute power to customer VMs.

The second trend is the increasing prevalence of IO acceleration technologies such as SmartSSDs [7, 16], SmartNICs [4, 8] and custom IO accelerators with attached computation that can offload some functionality from the host CPU and reduce the heavy tax burden. However, IO accelerators do not provide an end-to-end solution for offloading real-deployment storage stacks. Today, the storage functions in Table 1 cannot be fully accelerated in hardware for three reasons: (1) the functionalities are too complex for low-cost hardware acceleration, (2) acceleration hardware is typically designed for common-case operations but not end-to-end scenarios, or (3) the underlying accelerated functions are not composable.

Table 1 summarizes why the functionalities are not fully offload ready. We use one simple case as an example. For "local virtual SSDs," a cloud storage provider can employ Single Root IO Virtualization (SR-IOV) SSDs [20] where IOPS/bandwidth virtualization management is offloaded to the SSD hardware, freeing the host from such a burden. However, the cloud provider might want to combine virtualization with aggressive caching in spare host DRAM, but in-SSD accelerators cannot leverage the large host DRAM (*i.e.*, not composable with other host resources) and do not provide the same flexibility as software.

Local/remote *virtual* SSDs/services and caching. SR-IOV SSDs (hardware-assisted IO virtualization) do not have access to host DRAM. Thus local SSD caching for remote storage protocols (*e.g.* iSCSI [15], NVMeoF [2]) cannot be offloaded easily from x86.
Atomic write drive. Smart transactional storage devices [58, 65] do not provide atomicity across replicated drives/servers.
Versioned drive. A multi-versioned drive that allows writers to advance versions via atomic writes while the readers can stick to older versions, not supported in today's smart drives.
Priority virtual drive. Requires IO scheduling on every IO step (*e.g.*, through SSDs/NICs) with *flexible* policies, hard to achieve in hardware-based policies (*e.g.*, SSD-level prioritization).
Spillover drive. Uses few GBs of a local virtual drive and spills the remaining over to remote drives or services (elastic volumes), a feature that must combine local and remote virtual drives/services.
Replication & distribution. Accelerated cards can offload consistent and replicated writes, but they typically depend on a particular technology (*e.g.* non-volatile memory).
Other functionalities. Compression, deduplication, encryption, etc. must be composable with the above drives, not achievable in custom accelerators.

Table 1. Real storage functions, not offload ready. *The table summarizes why real cloud drive services are either not completely offload ready or not easily composable with each other.*

Custom IO accelerators have another downside. As acceleration hardware evolves, the entire fleet of servers may never be uniform; each generation of servers will have better, but slightly different hardware from previous ones. Without a *unifying software platform*, we run the risk of fragmenting the fleet into silos defined by their hardware capabilities and specific software optimizations.

We observe another trend in cloud platforms: ARM co-processors are being deployed for server workloads. This is a more suitable alternative compared to custom accelerators; ARM cores are more general (retain x86 generality) and powerful enough to run complex storage functions without major performance loss.

Offloading the storage stack to ARM co-processors can bring substantial cost savings. The bill-of-material cost of an ARM System-on-Chip (SoC) is low and the power consumed is 10W, making an annual total Cost of Ownership (TCO) of less than ~$100 (<~3% of a typical server's annual TCO). In turn, this SoC frees up several x86 cores thereby directly increasing the revenue from the services running on the server proportional to the additional cores – annually ~$2,000 or more (20×) when the cores are used for running customer VMs and significantly higher for more lucrative services. Even when accounting for typical replacement rates, ARM SoC TCO would still be less than one year rent of the smallest recommended VM in the cloud.

But there is a major challenge, just dropping an ARM SoC on a PCIe slot would not be enough. We had to rethink the entire storage stack design to meet real deployment challenges: hardware fungibility, portability, virtualizability, composability, efficiency, and extensibility (all laid out in §2.1), which led us to designing LeapIO.

1.1 LeapIO

We present LeapIO, our next-generation cloud storage stack that leverages ARM SoC as co-processors. To address deployment goals (§2.1) in a holistic way, LeapIO employs a set of OS/software techniques on top of new hardware capabilities, allowing storage services to portably leverage ARM co-processors. LeapIO helps cloud providers cut the storage tax and improve utilization without sacrificing performance.

At the abstraction level we use NVMe, "the new language of storage" [5, 17]. All involved software layers from guest OSes, LeapIO runtime, to new storage services/functions all see the same device abstraction: *virtual NVMe drive*. They all communicate via the mature NVMe *queue-pair* mechanism accessible via basic memory instructions pervasive across x86 and ARM, QPI or PCIe.

On the software side, we build a runtime that hides the NVMe mapping complexities from storage services. Our runtime provides a *uniform address space across the x86 and ARM cores*, which brings two benefits.

First, our runtime *maps NVMe queue pairs across hardware/software boundaries* – between guest VMs running on x86 and service code offloaded to the ARM cores, between client- and server-side services, and between all the software layers and backend NVMe devices (*e.g.*, SSDs). Storage services can now be written in *user space* and be *agnostic* about whether they are offloaded or not.

Second, our runtime provides an *efficient data path* that alleviates unnecessary copying across the software components via *transparent address translation* across multiple address spaces: guest VM, host, co-processor user and kernel address spaces. The need for this is that while ARM SoC retains the computational generality of x86, it does not retain the peripheral generality that would allow different layers access the same data from their address spaces.

The runtime features above cannot be achieved without *new hardware support*. We require four new hardware properties in our SoC design: host DRAM access (for NVMe queue mapping), IOMMU access (for address translation), SoC's DRAM mapping to host address space (for efficient data path), and NIC sharing between x86 and ARM SoC (for RDMA purposes). All these features are addressable from the SoC side; no host-side hardware changes are needed.

We build LeapIO in 14,388 LOC across the runtime, host OS/hypervisor and QEMU changes, and design the SoC using Broadcom StingRay V1 SoC (a 2-year hardware development).

Storage services on LeapIO are "offload ready;" they can portably run in ARM SoC or on host x86 in a trusted VM. The software overhead only exhibits 2-5% throughput reduction compared to bare-metal performance (still delivering 0.65 million IOPS on a datacenter SSD). Our current SoC prototype also delivers an acceptable performance, 5% further reduction on the server side (and up to 30% on the client) but with more than 20× cost savings.

Finally, we implement and compose different storage functions such as a simple RAID-like aggregation and replication of local/remote virtual drives via NVMe-over-RDMA/TCP/REST, an IO priority mechanism, a multi-block atomic-write drive, a snapshot-consistent readable drive, the first virtualized OpenChannel SSDs exposed to guest VMs, block cache, and many more, all written in 70 to 4400 LOC in user space, demonstrating the ease of composability and extensibility that LeapIO delivers.

Overall, we make the following contributions:

- We define and design a set of hardware properties to make ARM-to-peripheral communications as efficient as x86-to-peripherals.
- We introduce a uniform address space across x86, ARM SoC and other PCIe devices (SSDs, NICs) to enable line-rate address translations and data movement.
- We develop a portable runtime which abstracts away hardware capabilities and exploits the uniform address space to make offloading seamless and flexible.
- We build several novel services composed of local/remote SSDs/services and perform detailed performance benchmarks as well as analysis.

2 Extended Motivation

2.1 Goals

Figure 1 paints the deployment goals required in our next-generation storage stack. As shown, the fundamental device abstraction is *NVMe virtual drive*, illustrated with a "●", behind which are the NVMe submission and completion queue pairs for IO management. The deployment/use model of LeapIO can be seen in Figure 1a. Here a user mounts a virtual block drive ● to her VM (guest VM) just like a regular NVMe drive. LeapIO then manages all the complex storage functions behind this "simple" ●, as illustrated in the figure. We now elaborate the goals.

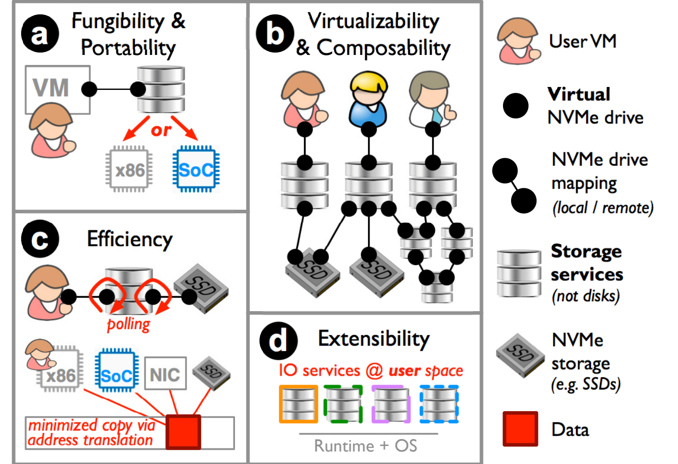


Figure 1. Goals. *As described in Section 2.1.*

ⓐ **Fungibility and portability:** We need to keep servers fungible regardless of their acceleration/offloading capabilities. That is, we treat accelerators as *opportunities* rather than necessities. In LeapIO, the storage software stack is portable – able to run on x86 *or* in the SoC whenever available (*i.e.*, "offload ready") as Figure 1a illustrates. The user/guest VMs are agnostic to what is implementing the virtual drive.

Fungibility and portability prevent "fleet fragmentation." Different server generations have different capabilities (*e.g.*, with/without ARM SoC, RDMA NICs or SR-IOV support), but newer server generations must be able to provide services to VMs running on older servers (and vice versa). Fungibility also helps provisioning; if the SoC cannot deliver enough bandwidth in peak moments, some services can borrow the host x86 cores to augment a crowded SoC.

ⓑ **Virtualizability and composability:** We need to support virtualizing and composing of, not just local/remote SSDs, but also local/remote IO *services* via NVMe-over-PCIe /RDMA/TCP/REST. With LeapIO runtime, as depicted in Figure 1b, a user can obtain a local virtual drive that is mapped to a portion of a local SSD that at the same time is also shared by another remote service that glues multiple virtual drives into a single drive (*e.g.*, RAID). A storage service can be composed on top of other remote services.

ⓒ **Efficiency:** It is important to deliver performance close to bare metal. LeapIO runtime must perform continuous *polling* on the virtual NVMe command queues as the proxy agent between local/remote SSD and services. Furthermore, ideally services must *minimize data movement* between different hardware/software components of a machine (on-x86 VMs, in-SoC services, NICs, and SSDs), which is achieved by LeapIO's uniform address space (Figure 1c).

ⓓ **Service extensibility:** Finally, unlike traditional block-level services that reside in the kernel space for performance, or FPGA-based offloading which is difficult to program, LeapIO enables storage services to be implemented at the *user space* (Figure 1d), hence allowing cloud providers to easily manage, rapidly build, and communicate with a variety of (trusted) complex storage services.

2.2 Related Work

		Acc	Uni	Port	vNVMe	Usr	sVirt
A	ActiveFlash [76]	√					
	Biscuit [40]	√			√	√	
	IDISKS [46]	√					
	INSIDER [69]	√					
	LightStore [35]	√				√	
	SmartSSD [44]	√					
	SR-IOV [20]	√	√		√		
	Summarizer [50]	√			√	√	
B	Decibel [60]				√	√	
	IOFlow [75]					√	
	LightNVM [32]	√			√	√	
	NVMeoF [2]			√	√	√	
	SDF [63]	√				√	
	SPDK [78]				√	√	
C	Helios [61]	√	√				
	Reflex [49]				√	√	
	Solros [59]	√	√				
	VRIO [51]					√	
D	AccelNet [39]	√					
	Bluefield [4]	√			√	√	
	FlexNIC [45]	√				√	
	Floem [64]	√				√	
	KV-Direct [54]	√					
	NetCache [42]	√				√	
	NICA [38]	√					
	UNO [52]	√		√			
E	GPUfs [74]		√			√	
	HeteroISA [30]			√		√	
	HEXO [62]	√		√			
	OmniX [73]	√	√			√	
	Popcorn [31]	√		√			
	LeapIO	√	√	√	√	√	√

Table 2. Related Work (§2.2). *The columns (dimensions of support) are as follow.* **Acc***: Hardware acceleration;* **Uni***: Unified address space;* **Port***: Portability/fungibility;* **vNVMe***: Virtual NVMe abstraction;* **Usr***: User-space/software-defined storage functions;* **sVirt***: Simultaneous local+remote NVMe virtualization. The row (related work) categories are:* **A***. Storage accelartion/offloading;* **B***. Software-defined storage;* **C***. Disaggregated/split systems;* **D***. Programmable NICs; and* **E***. Heterogeneous system designs. We reviewed in detail a total of 85 related papers (not all shown here), other works include [25, 26, 28, 29, 33, 34, 37, 43, 47, 48, 53, 56, 57, 66–68, 70–72, 77].*

To achieve all the goals above, just dropping in ARM SoCs on the PCIe slots in the server would not be enough. The features above require new hardware capabilities and OS/hypervisor-level support. We reviewed the growing literature in IO accelerator and virtual storage and did not find a single technology that meets our needs. Below we summarize our findings as laid out in Table 2, specifically in the context of the six dimensions of support (represented by the last six columns).

First, many works highlight the need for IO accelerators (the "**Acc**" column), *e.g.*, with ASIC, FPGA, GPU, Smart SSDs, and custom NICs. In our case, the accelerator is a custom ARM-based SoC (§3.1) for reducing the storage tax.

When using accelerators, it is desirable to support a unified address space (**Uni**) to reduce data movement. While most work in this space focus on unifying two address spaces (*e.g.*, host-GPU, host-coprocessor, or host-SSD spaces), we have to unify three address spaces (guest-VM/host/SoC) with 2-level address translations (§3.4).

One uniqueness of our work is addressing portability/ fungibility (**Port**) where LeapIO runtime and arbitrary storage functions can run on either the host x86 or ARM SoC, hence our SoC deployment can be treated as an acceleration opportunity rather than a necessity. In most of other works, only specifically provided functions (*e.g.*, caching, replication, or consensus protocol) are offloadable.

We chose virtual NVMe (**vNVMe**) as the core abstraction such that we can establish an end-to-end storage communication from guest VMs to the remote backend SSDs through many IO layers/functions that speak the same NVMe language (§3.3). With this for example, LeapIO is the first platform that enables virtual SSD channels (backed by Open-Channel SSDs) to be composable for guest VMs (§5.3).

All of the above allow us to support user-space/software-defined storage functions (**Usr**) just like many other works. In LeapIO, user-level storage functions can be agnostic to the underlying hardware (x86/SoC, local/remote storage). With this for example, LeapIO is the first to support user-space NVMeoF with stable performance (§5.1).

Finally, to support "spillover drive" (Table 1), LeapIO is the first system that supports *simultaneous* local and remote NVMe virtualization (the **sVirt** column). Related work like Bluefield with hardware NVMe emulation [4, 14] cannot achieve this because it can only support running in either local or remote virtualization mode (*e.g.*, initiator or target), but not both simultaneously in composable ways.

Overall, our unique contribution is combining these six dimensions of support to address our deployment goals. We also would like to emphasize that our work is orthogonal to other works. For example, in the context of GPU/FPGA offloading, application computations can be offloaded there, but when IOs are made, the storage functions are offloaded to our ARM SoC. In terms of virtual NIC, its network QoS

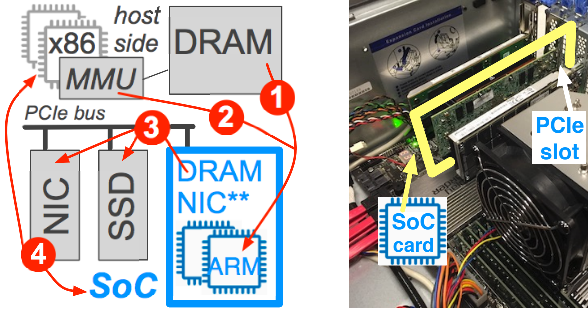


Figure 2. Hardware requirements. *The figure is explained in Section 3.1. "NIC**" means an optional feature of the SoC. x→y means x should be exposed or accessible to y.*

capability can benefit remote storage QoS. In terms of framework/language level support for in-NIC/Storage offloading, it can be co-located with LeapIO to accelerate server applications. In terms of accelerator level support for OS services or OS support for accelerators, LeapIO can benefit from such designs for more general purpose application offloading.

3 Design

We now present the design of LeapIO from different angles: hardware (§3.1), software (§3.2), control flow (§3.3), data path (§3.4), and x86/ARM portability (§3.5).

We first clarify several terms: *"ARM"* denotes cheaper, low-power processors suitable enough for storage functions (although their architecture need not be ARM's); *"SoC"* means ARM-based co-processors with ample memory bundled as a PCIe SoC; *"x86"* implies the powerful and expensive host CPUs (although can be non x86); *"rNIC"* stands for RDMA-capable NIC; *"SSD"* means NVMe storage; *"functions"* and *"services"* are used interchangeably.

3.1 The Hardware View

We begin with the hardware view.

In the left side of Figure 2, the top area is the host side with x86 cores, host DRAM, and IOMMU. In the middle is the PCIe bus connecting peripheral devices. In the bottom right is our SoC card (bold blue edge) containing ARM cores and on-SoC DRAM. Our SoC and rNIC are co-located on a single PCIe card as explained later. The right side in Figure 2 shows a real example of our SoC deployment. In terms of hardware installation, the SoC is simply attached to a PCIe slot. However, easy offloading of services to the SoC while maintaining performance requires four hardware capabilities (labels ① to ④ in Figure 2), which all can be addressed from the SoC vendor side.

① **HW_1: Host DRAM access by SoC**. The SoC must have a DMA engine to the host DRAM (just like rNIC). However, it must allow the user-space LeapIO runtime (running in the ARM SoC) to access the DMA engine to reach the location of all the NVMe queue pairs mapped between the on-x86 user VMs, rNIC, SSD, and in-SoC services (§3.3).

② **HW_2: IOMMU access by SoC**. The trusted in-SoC LeapIO runtime must have access to an IOMMU coherent with the host in order to perform page table walk of the VM's address space that submitted the IO. When an on-x86 user VM accesses a piece of data, the data resides in the host DRAM, but the VM only submits the data's guest address. Thus, the SoC must facilitate the LeapIO runtime to translate guest to host physical addresses via the IOMMU (§3.4).

③ **HW_3: SoC's DRAM mapped to host**. The on-SoC DRAM must be visible by the rNIC and SSD for zero-copy DMA. For this, the SoC must expose its DRAM space as a PCIe BAR (base address register) to the host x86. The BAR will then be mapped as part of the host physical address space by the host OS. With this capability, main hardware components such as rNIC, SSD, host x86, and the SoC can read/write data via the host address space without routing data back and forth (§3.4).

④ **HW_4: NIC sharing**. The NIC must be "shareable" between the host x86 and ARM SoC because on-x86 VMs, other host agents, and in-SoC services are all using the NIC. NIC can be used by the host to serve VM traffic as well as by the SoC for offloaded remote storage functions. One possibility is to co-locate the ARM cores and the NIC on the same PCIe card ("NIC**" in Figure 2), hence not dependent on the external NIC capabilities (§4).

3.2 The Software View

Now we move to the software view. To achieve all the goals in §2.1, LeapIO software is relatively complex, thus we decide to explain it by first showing the high-level stages of the IO flows, as depicted in stages ⓐ to ⓕ in Figure 3.

ⓐ **User VM**. On the client side, a user runs her own application and guest OS of her choice on a VM where *no* modification is required. For storage, the guest VM runs on the typical NVMe device interface (*e.g.*, `/dev/nvme0n1`) exposed by LeapIO as a queue pair (represented by ●) containing submission and completion queues (SQ and CQ) [1]. This NVMe drive which can be a local (ephemeral) drive or something more complex will be explained later.

ⓑ **Host OS**. We add a capability into the host OS for building queue-pair mappings (more in 3.3) such that the LeapIO runtime ⓒ sees the same NVMe command queue exposed to the VM. The host OS is *not* part of the datapath.

ⓒ **Ephemeral storage**. If the user VM utilizes local SSDs (*e.g.*, for throughput), the requests will be put into the NVMe queue mapped between the LeapIO runtime and the SSD device (the downward ●—●). Because the SSD is not in the SoC (not inside the bold edge), they need to share the NVMe queue stored in the host DRAM (more in §3.3).

ⓓ **Client-side LeapIO runtime and services**. The client-side runtime (shaded area) represents the LeapIO runtime running on the ARM SoC (bold blue edge). This runtime "glues" all the NVMe queue pairs (●—●) and end-to-end storage paths over a network connection (◆—◆).

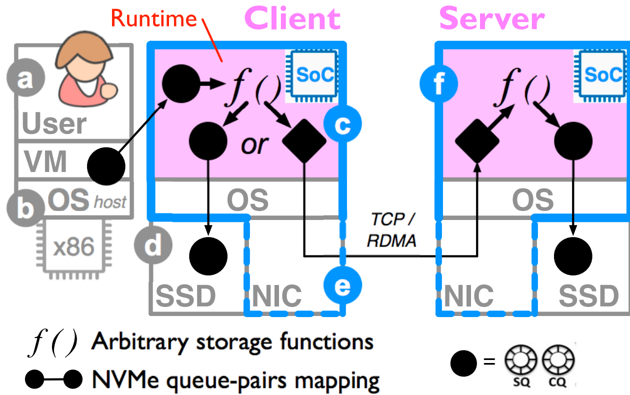


Figure 3. Software view. *The figure shows the software design of LeapIO (Section 3.2). For simplicity, we use two nodes, client and server, running in a datacenter. The arrows in the figure only represent the logical control path, while the data path is covered in §3.4. Our runtime and storage services (the shaded/pink area) can transparently run in the SoC (as shown above) or on the host x86 via our "SoC_{VM}" support (in §3.5).*

To quickly process new IOs, the LeapIO runtime polls the VM-side NVMe submission queue that has been mapped to the runtime address space (●–●). This runtime enables services to run arbitrary storage functions in user space ("$f()$", see Table 1) that simply operate using NVMe interface. The functions can then either forward the IO to a local NVMe drive (●) and/or a remote server with its own SoC via RDMA or TCP (◆). Later, §3.4 will provide details of the data path.

At this stage, we recap the aforementioned benefits of LeapIO. First, the cloud providers can develop and deploy the services in user space (*extensibility*). The LeapIO runtime also does not reside in the OS, hence all data transfers bypass the OS level (both host OS and SoC-side OS are skipped). The SoC-side OS can be any standard OS. Second, with mapped queue pairs, the runtime employs polling to maintain fast latency and high throughput (*efficiency*). Third, VMs can obtain a rich set of block-oriented services via virtual NVMe drives (*virtualizability/composability*). Most importantly, although in the figure, the LeapIO runtime and services are running in the SoC, they are also designed to transparently run in a VM on x86 to support older servers (*portability*), which we name the "SoC_{VM}" feature (more in §3.5).

ⓔ **Remote access (NIC).** If the user stores data in a remote SSD or service, the client runtime simply forwards the IO requests to the server runtime via TCP or RDMA through the NIC (◆–◆). Note that the NIC is housed in the same PCIe slot (dotted bold edge) as the ARM SoC in order to fulfill property $\mathbf{HW_4}$ (shareable rNIC).

ⓕ **Server-side LeapIO runtime and services.** The server-side LeapIO runtime prepares the incoming command and data by polling the queues connected to the client side (◆). It then invokes the server-side storage functions $f()$ that also run in the user level within the SoC. The server-side service then can forward/transform the IOs to one or more NVMe drives (●) or remote services. The figure shows the access path to its local SSD (the right-most ●–●).

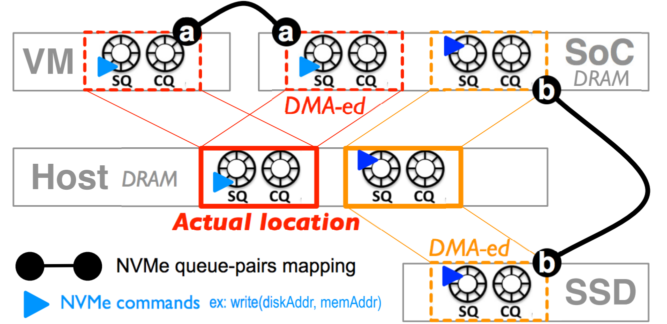


Figure 4. Control setup. *The figure shows the mappings of NVMe queue pairs to pass IO commands across the hardware and software components in LeapIO, as described in §3.3.*

3.3 The Control Setup

We now elaborate on how LeapIO provides the *NVMe queue-pair mapping support* to allow different components in LeapIO to use the same NVMe abstraction to communicate with each other. To illustrate this, we use the two logical queue-pair mappings (two ●–●) in the client side of Figure 3 and show the physical mappings in Figure 4a-b.

ⓐ**VM-runtime queue mapping**. This is the mapping between the user VM and the in-SoC client runtime (red lines). The actual queue-pair location is in the host DRAM (middle row). The user VM (upper left) can access this queue pair via a standard mapping of guest to host address (via hypervisor-managed page table of the VM). For the in-SoC runtime to see the queue pair, the location must be mapped to the SoC's DRAM (upper right), which is achieved via DMA. More specifically, our modified host hypervisor establishes an NVMe *admin* control channel with LeapIO runtime. There is a single admin NVMe queue pair that resides in the host DRAM but it is DMA-ed by the host to the runtime address space, thus requiring property $\mathbf{HW_1}$ (§3.1).

ⓑ**Runtime-SSD queue mapping**. This is the mapping between the in-SoC runtime with the SSD (orange lines). Similar to the prior mapping, the hypervisor provides to the SSD the address ranges within the memory mapped region. The SSD does not have to be aware of the SoC's presence. Overall, the memory footprint of a queue pair is small (around 80 KB). Thus, LeapIO can easily support hundreds of virtual NVMe drives for hundreds of VMs in a single machine without any memory space issue for the queues.

With this view (and for clarity), we repeat again the control flow for local SSD write operations, using Figure 4. First, a user VM submits an NVMe command such as `write()` to the submission queue (red `SQ` in VM space, ⓐ). Our in-SoC runtime continuously polls this `SQ` in its address space (red

SQ in SoC's DRAM, ⓐ) and does so by *only* burning an ARM core. The runtime converts the previous NVMe command, submits a new one to the submission queue in the runtime's address space (orange SQ in SoC's DRAM, ⓑ), and rings the SSD's "doorbell" [1]. The SSD controller reads the NVMe write command that has been DMA-ed to the device address space (orange SQ in the SSD, ⓑ). Note that, all of these *bypass* both the host and the SoC OSes.

3.4 The Data Path (Address Translation Support)

Now we describe the most challenging goal: *efficient data path*. The problem is that in existing SmartNIC or SmartSSD SoC designs, ARM cores are hidden behind either the NIC controller or storage interface, thus ARM-x86 communication must be routed through NIC/storage control block and not efficient. Furthermore, many software/hardware components are involved in the data path, hence we must *minimize data copying*, which we achieve by building an *address mapping/translation support* using the aforementioned HW properties (3.1). Figure 5 walks through this most complicated LeapIO functionality (write path only) in the context of a VM accessing a remote SSD over RDMA. Before jumping into the details, we provide high-level descriptions of the figure and the legend.

Components: The figure shows different hardware components and software layers in data and command transfers such as user application, guest VM, host DRAM ("hRAM"), SoC-level device DRAM buffer ("sRAM[1]"), client/server runtime, rNICs, and the back-end SSD.

Command flow: Through these layers, NVMe commands (represented as blue ▶) flow through the NVMe queue-pair abstraction as described before. The end-to-end command flow is shown in non-bold blue line. An example of an NVMe IO command is `write(blkAddr, memAddr)` where `blkAddr` is a block address within the virtual drive exposed to the user and `memAddr` is the address of the data content in the guest VM.

Data location and path: We attempt to minimize data copying (reduced ■ count) and allow various software and hardware layers access the data via memory mapping (□). The data is transferred (bold red arrow) between the client and the server, in this context via RDMA-capable NICs.

Address spaces: While there is only one copy of the original data (■), different hardware components and software layers need to access the data in their own address spaces, hence the *need for an address translation support*. Specifically, there are four address spaces involved (see the figure legend): **(1)** `guestAddr` *gA* representing the guest VM address, **(2)** `hostAddr` *hA* denoting the host DRAM physical address, **(3)** `logicalAddr` *lA* implying the logical address (SoC user space) used by the LeapIO runtime and services, **(4)** `socAddr` *sA* representing the SoC's DRAM physical address.

[1] **sRAM** denotes on-SoC DRAM, not static RAM.

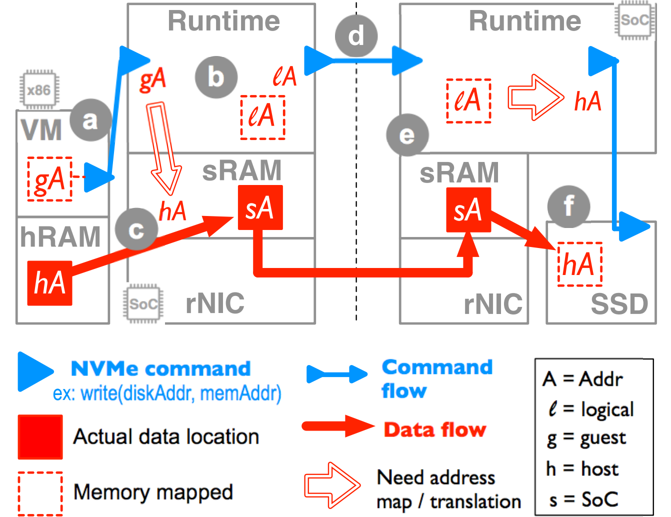


Figure 5. Datapath and address translation. *The figure shows how we achieve an efficient data path (minimized copy) with our address translation support, as elaborated in §3.4. The figure only shows write path via RDMA (read path is similar).*

In our SoC deployment, the SoC and rNIC are co-located ($\mathbf{HW}_4$ in §3.1), hence `logicalAddr` mode is the most convenient one for using RDMA between the client/server SoCs.

3.4.1 Client-Side Translation.

For the client side, we will refer to Figure 5ⓐ–ⓓ.

In step ⓐ, on-x86 guest VM allocates a data block, gets `guestAddr` *gA* and puts a `write` NVMe command ▶ with the `memAddr` pointing to the `guestAddr` *gA*, *i.e.*, `write(blkAddr,` *gA*`)`. The data is physically located in the host DRAM (■ at `hostAddr` *hA*).

In ⓑ, the LeapIO user-space runtime sees the newly submitted command ▶ and prepares a data block via user-space `malloc()`, hence later it can touch the data □ via `logicalAddr` *lA* in the runtime's address space. Because the runtime runs in the SoC, this *lA* is physically mapped to the SoC's DRAM (■ at `socAddr` *sA*). Remember that at this point the data at `socAddr` *sA* is still empty.

In step ⓒ, we need to make a host-to-SoC PCIe data transfer (see notes below on efficiency) and here ***the first address translation is needed*** (the first double-edged arrow). That is, to copy the data from the host to SoC's DRAM, we need to translate `guestAddr` *gA* to `hostAddr` *hA* because the runtime only sees "*gA*" in the NVMe command. This `guestAddr-hostAddr`translation is *only available* in the host/hypervisor-managed page tables of the VM that submitted the request. Thus, our trusted runtime must be given access to the host IOMMU (property $\mathbf{HW}_1$ in §3.1).

Next, after obtaining the `hostAddr` *hA*, our runtime must read the data and copy it to `socAddr` *sA* (the first bold red arrow). Thus, the runtime must also have access to the SoC's DMA engine that will DMA the data from the host to SoC's DRAM (hence property $\mathbf{HW}_2$ in §3.1).

In (d), at this point, the data is ready to be transferred to the server via RDMA. The client runtime creates a new NVMe command ▶ and supplies the client side's `logicalAddr` *lA* as the new `memAddr`, *i.e.*, `write(blkAddr,`*lA*`)`. The runtime must also register its `logicalAddr` via the `ibverbs` calls so that the SoC OS (not shown) can tell the rNIC to fetch the data from `socAddr` *sA* (the SoC OS has the *lA*-*sA* mapping). This is a standard protocol to RDMA data.

We make several notes before proceeding. First, the host-to-SoC data transfer should *not* be considered as an overhead, but rather a *necessary* copy as the data must traverse the PCIe boundary at least once. This transfer is not done in software, it is performed by enqueueing a single operation to the PCIe controller that does a hardware DMA operation between the two memory regions. Second, LeapIO must be fully trusted to be given host-side page table access, which is acceptable as LeapIO is managed by the cloud provider. A malicious VM's attack surface is restricted to the NVMe queue pairs. Whenever LeapIO detects illegal NVMe commands, it fails the IOs directly. Overall, LeapIO doesn't expose extra attack surface compared to existing on-x86 hypervisor IO interface.

3.4.2 Server-Side Translation. For the server side, we refer to Figure 5(e)–(g). LeapIO server keeps monitoring data coming from the network and migrates data to SSD efficiently with direct NVMe access and DMA data transfer between ARM and SSD.

In (e), LeapIO server runtime sees the new command ▶ and prepares a data buffer □ at its `logicalAddr` *lA* (a similar process as in step (b)). The runtime then makes an RDMA command to fetch the data from the client runtime's `logicalAddr` *lA* provided by the incoming NVMe command. The server rNIC then puts the data directly in the SoC's DRAM (■ at `socAddr` *sA*). Now LeapIO services can read the data via the `logicalAddr` *lA* and run any storage functions $f()$ desired. When it is time to persist the data, the runtime submits a new NVMe command ▶ to the SSD.

In (f), being outside the SoC, the backend SSD can only DMA data using `hostAddr`(server side) , hence does not recognize `socAddr` *sA*. Thus, the server runtime must submit a new NVMe command that carries "`hostAddr` *hA*" as the `memAddr` of the next write command, *i.e.* `write(blkAddr,`*hA*`)`. This is ***the need for another address translation*** *lA*→*sA*→*hA* (the second double-edged arrow).

For *sA*→*hA*, we need to map the SoC's DRAM space to the aggregate host address space, which can be done with p2p-mem technology (property **HW**$_3$ in §3.1). With this, the aggregate host address space is the sum of the host and SoC DRAM. As a simplified example, the "`hostAddr` *hA*" that represents the `socAddr` *sA* can be translated from `hA=hostDramSize+sA` (details can vary).

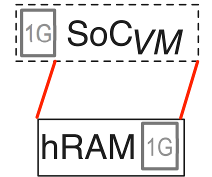
#lines	Core	SoC$_{VM}$	Emu
Runtime	8865	+850	+680
QEMU	1388	+385	
Host OS	2340	+560	+360

Table 3. LeapIO complexity (LOC). *(As described in §4)*

For *lA*→*sA*, the runtime can obtain the `logicalAddr` to `socAddr` translation from the standard `/proc` page map interface in the SoC OS. We use huge page tables and pin the runtime's buffer area so the translation can be set up in the beginning and not slow down the data path.

3.5 SoC$_{VM}$

For fungibility, we design LeapIO to portably run on SoC or x86, such that LeapIO runtime and services are *one code base* that does not fragment the fleet. To support LeapIO to run on x86, we design "SoC$_{VM}$" (a SoC-like VM) such that our overall design remains the same. Specifically, in Figure 4, the "SoC's DRAM" simply becomes the SoC$_{VM}$'s guest address space. In Figure 5, the `socAddr` essentially becomes the SoC$_{VM}$'s `guestAddr`.



To enable SoC$_{VM}$'s capability to access the host DRAM, our host hypervisor trusts the SoC$_{VM}$ and performs the memory mapping shown on the right figure. Imagine for simplicity that the SoC$_{VM}$ boots asking for 1GB. The hypervisor allocates a 1G space in the host DRAM, but before finishing, our modified hypervisor extends the SoC$_{VM}$'s address space by *virtually* adding the entire DRAM size. Thus, any `hostAddr` *hA* can be accessed via SoC$_{VM}$'s address `1GB+`*hA* (details can vary). To perform the user `guestAddr` *gA* to *hA* translation, we write a host kernel driver that supplies this to the SoC$_{VM}$ via an NVMe-like interface to avoid context switches. Finally, to share the guest VM's NVMe queue pairs with SoC$_{VM}$, we map them into SoC$_{VM}$ as a Base Address Register (BAR) of a virtual PCIe device.

SoC$_{VM}$ also supports legacy storage devices with no NVMe interface. Older generation servers and cheaper server SKUs that rely on SATA based SSDs or HDDs can also be leveraged in LeapIO via the SoC$_{VM}$ implementation (via `libaio`), furthering our fungibility goal. Moreover, SoC$_{VM}$ can *coexist* with the actual SoC such that LeapIO can schedule services on spare x86 cores when the SoC is full.

4 Implementation

Table 3 breaks down LeapIO 14,388 LOC implementation. The rows represent the software layers we add/modify, including LeapIO runtime, QEMU (v2.9.0), and the host OS/hypervisor (Linux 4.15). In the columns, "Core" represents the required code to run LeapIO on SoC, "SoC$_{VM}$" represents the support to portably run on x86, and "Emu" means the small emulated part of an ideal SoC (more below).

We develop LeapIO on a custom-designed development board based on the Broadcom StingRay V1 SoC that colocates an 100Gb Ethernet NIC with 8 Cortex-A72 ARM cores at 3 GHz. Our development board appears to x86 as a smart RDMA Ethernet controller with one physical function dedicated to the on-board SoC (and another for host/VM data), hence the ARM cores can communicate with x86 via RDMA over PCIe (*e.g.*, for setting up the queue pairs).

Of the four HW requirements (§3.1), our current SoC, after a 2-year joint hardware development process with Broadcom, can fulfill $\mathbf{HW}_1$, $\mathbf{HW}_3$ (SSD direct DMA from/to SoC DRAM) and $\mathbf{HW}_4$ (in-SoC NIC shareable to x86) fully and $\mathbf{HW}_2$ with a small caveat. For $\mathbf{HW}_2$ (IOMMU access), we currently satisfy this via huge page translations (fewer address mappings to cache in SoC) facilitated by the hypervisor, which bodes well with the use of huge pages in our cloud configuration. Our software is also conducive to using hardware based virutal NVMe emulators [10, 14] that can directly interact with the IOMMU.

For data-oriented services (*e.g.*, caching and transaction) in LeapIO local virtualization and remote server mode, peer-to-peer DMA (p2p-mem) [6] is used for direct SSD-SoC data transfer to efficiently stage data in SoC DRAM (no x86 involvement). Computation intensive tasks such as compression, encryption can be further offloaded to in-SoC hardware accelerators. Otherwise, we bypass SoC's DRAM (default SSD-host DMA mode) if data path services are not needed.

Despite lack of full $\mathbf{HW}_2$ support, we note that the LeapIO software design and implementation are complete and ready to leverage newer hardware acceleration features such as hardware NVMe emulation features when they are available. Therefore the system performance will improve as hardware evolves while a full software-only (SoC_{VM}) as well as SoC-only implementation allow us to reduce resource/code fragmentation and hardware dependency. To the best of our knowledge, LeapIO is the first comprehensive storage function virtualization stack that uses acceleration opportunistically. It enables cloud providers to expose identical storage services to VMs regardless of server configurations.

5 Evaluation

We thoroughly evaluate LeapIO with the following questions: §5.1: How much overhead does LeapIO runtime impose compared to other IO pass-through/virtualization technologies? §5.2: Does LeapIO running on our current ARM SoC deliver a similar performance compared to running on x86? §5.3: Can developers easily write and compose storage services/functions in LeapIO?

To mimic a datacenter setup, we use a high-end machine with an 18-core (36 hyperthreads) Intel i9-7980XE CPU running at 2.6GHz with 128G DDR4 DRAM. The SSD is a 2TB data-center Intel P4600 SSD. The user/guest VM is given 8 cores and 8 GB of memory and LeapIO runtime uses 1 core with two loops, one each for polling incoming submission queues, and SSD completion queues.

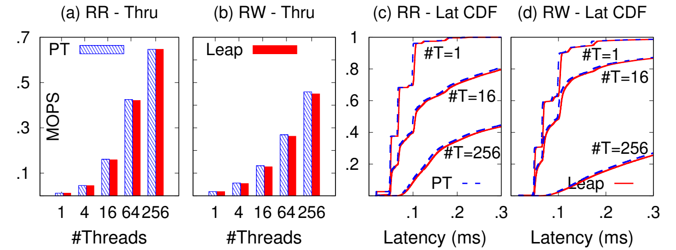


Figure 6. LeapIO vs. Pass-through (PT) with FIO. *The figure compares LeapIO and PT performance as described in §5.1(1). From left to right, the figures show read-only (RR) and read-write (RW) throughputs followed with latency CDFs.*

As we mentioned earlier, modern storage stack is deep and complex. To guide readers in understanding the IO stack setup, we will use the following format: A/B/C/... where *A*/*B* implies *A* using/running on top of *B*. For clarity, we compare one layer at a time, *e.g.*, A/$\mathbf{B}_1$**-or-**$\mathbf{B}_2$/... when comparing two approaches at layer B. Finally, to easily find our main observations, we label them with obs.

5.1 Software Overhead

This section dissects the software overhead of LeapIO runtime. To not mix performance effects from our SoC hardware, we first run LeapIO inside SoC_{VM} (§3.5) on x86.

(1) LeapIO vs. PT on Local SSD with FIO/SPDK. We first compare LeapIO with "pass-through" technology (PT) which arguably provides the *most bare-metal* performance a guest VM can reap. With pass-through, guest VM ("gVM") owns the entire local SSD and directly polls the NVMe queue pairs without host OS interference (but PT does not virtualize the SSD like we do). We name this lower stack "gVM/**PT**/SSD" and compare it with our "gVM/**LeapIO**/SSD" stack. Now, we vary what we run on the guest VM.

First, we run the FIO benchmark [12] on top of SPDK in the guest VM to not wake up the guest OS (**FIO/SPDK**). This setup gives the highest bare-metal performance as *neither* the guest/host OS is in the data path. We run FIO in two modes (read-only or 50%/50% read-write mix) of 4KB blocks with 1 to 256 threads. To sum up, we are comparing these two stacks: FIO/SPDK/gVM/**PT-or-LeapIO**/SSD.

obs Figure 6 shows that we are not far from the bare-metal performance. More specifically, Figure 6a-b shows that LeapIO runtime throughput drops only by 2% and 5% for the read-only and read-write throughputs respectively. The write overhead is higher because our datacenter SSD employs a large battery-backed RAM that can buffer write operations in <5µs. In Figure 6c-d, below p99 (the 99th percentile), LeapIO runtime shows only a small overhead (3% on average). At p99.9, our overhead ranges between 6 to 12%.

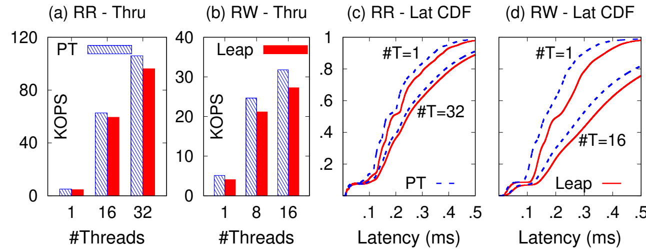


Figure 7. Leap vs. PT (YCSB/RocksDB). *The figure compares LeapIO and pass-through (PT) as described in §5.1(2).*

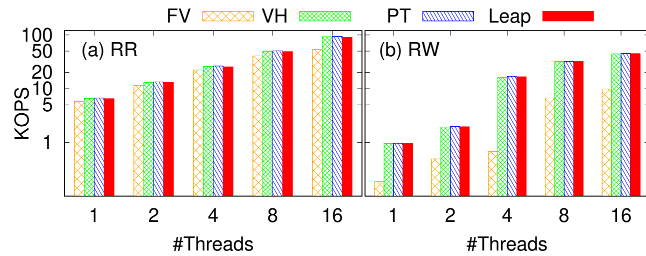


Figure 8. Leap vs. other virt. technologies. *The figure compares LeapIO with full virtualization (FV) and virtual host (VH) as described in §5.1(3).*

LeapIO runtime is fast because of the direct NVMe queue-pair mapping across different layers. For each 64-byte submission entry, LeapIO runtime only needs to translate 2-3 fields with simple calculations and memory fetches (for translations).

In another experiment (not shown), we convert the 256 threads from 1 guest VM in Figure 6a into 8 guest VMs each with 32 threads and obtain the same results. This demonstrates that LeapIO scales well with the number of guest NVMe queue pairs managed.

(2) LeapIO vs. PT on Local SSD with YCSB/RocksDB. Next, we run a real application: RocksDB (v6.0) [18] on ext4 serving YCSB workloads [36] (YCSB/RocksDB/gOS). YCSB is set to make uniform request distributions (to measure the worst-case performance) across 100 million key-value entries. We perform read-only or 50-50 read/write workloads. Figure 7 confirms the low software overhead of LeapIO by comparing these two stacks: YCSB/RocksDB/gOS/gVM/**PT-or-LeapIO**/SSD. Compared to Figure 6c-d, LeapIO latencies are worse than PT mainly due to the software virtual interrupt overhead (VM-exits).

(3) LeapIO vs. Other Technologies on Local SSD. Now we repeat the above experiments but cover other virtualization technologies. To make a faster RocksDB setup that bypasses the guest OS, we run RocksDB on SPDK and run `db_bench` benchmark (db_bench/RocksDB/SPDK/gVM). We switch to `db_bench` as YCSB workloads require the full POSIX API that is currently not supported by SPDK.

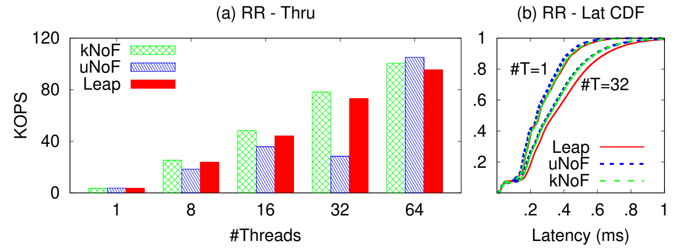


Figure 9. LeapIO vs. kernel/user NVMeoF. *We compare LeapIO remote NVMe feature with kernel and user NVMeoF (κNoF and υNoF). υNoF is unstable; with 32 threads, at p99.9 υNoF reaches 14ms while LeapIO can deliver 1.4ms, and at p99.99 υNoF reaches almost 2000ms while LeapIO is still around 7ms.*

We now vary the technologies under the guest VM (gVM/**FV-or-VH-or-PT-or-LeapIO**/SSD). Full virtualization ("**FV**") [13] provides SSD virtualization but is the slowest among all as it must wake up the host OS (via interrupts) to reroute all the virtualized IOs. Virtual host ("**VH**") [9] is a popular approach [3] that combines virtualization and polling but requires guest OS changes (*e.g.*, using the `virtio` interface and SPDK-like polling to get rid of interrupts.)

obs Figure 8 shows the results. While LeapIO loses by 3% to PT, when compared to popular IO virtualization technologies such as virtual-host, LeapIO throughput degradation is only 1.6%. At p99.99 latency LeapIO is only slower by 26µs (1%). This is an acceptable overhead considering that now we can easily move IO services to ARM co-processors.

(4) LeapIO vs. NVMeoF for Remote SSD access. We compare LeapIO server-side runtime with a popular remote IO virtualization technology, NVMeoF, which is a standard way for disaggregating NVMe storage access over RDMA/TCP [2]. Once connecting the NVMeoF client, the server continuously minotors and routes incoming NVMe commands to the backend SSDs. There are two server-side NVMeoF options we evaluate: *kernel-based* one that works in an interrupt-driven mode ("**κNoF**") and *user-space* one that utilizes SPDK for polling ("**υNoF**"). We use the YCSB/RocksDB client setup as before, but now with remote SSD. Thus, we compare YCSB/RocksDB/gOS/gVM/*client*/–RDMA–/**κNoF-or-υNoF-or-LeapIOServer**/SSD, where "*client*" implies the client-side runtime of either κNoF, υNoF, or LeapIO (TCP setup omitted due to space limit).

obs Based on Figure 9, we make two observations here. First kernel-based NVMeoF (κNoF) is most stable and performant, but is not easily extensible as services must be built in the kernel. However, our more extensible LeapIO only imposes a small overhead (6% throughout loss and 8% latency overhead). Second, interestingly we found that user-space NVMeoF (υNoF) is *unstable*. In majority of the cases, it is worse than LeapIO but in one case (64 threads) υNoF is better (after repeated experiments). υNoF combined with RDMA is a relatively new and some performance and reliability issues have been recently reported [21–23]. We also

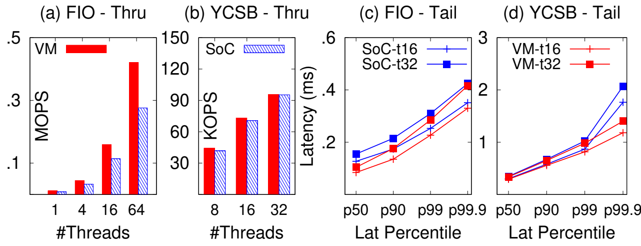


Figure 10. SoC vs. SoC$_{VM}$ Benchmarks. *The figure compares the performance of LeapIO SSDs (local and remote) running on a SoC vs. in a SoC$_{VM}$ as described in §5.2.*

tried running UNoF over TCP to no avail (not shown for space). With this, we can claim that LeapIO is *the first* user-space NVMeoF platform that delivers stable performance for the VMs.

5.2 SoC Performance

We now dissect separately the performance of LeapIO client- and server-side runtimes on an ARM SoC vs. on x86.

(1) Local SSD (realSoC vs. SoC$_{VM}$). We reuse the FIO-on-local-SSD stack in §5.1.1 for this experiment (specifically FIO/SPDK/gVM/**SoC$_{VM}$-or-realSoC**/SSD). Figures 10a&c show realSoC runtime is up to 30% slower than SoC$_{VM}$. This is because in our current implementation, we access the guest VMs' and SoC-side queue pairs via SoC-to-host one-sided RDMA (§4), which adds an expensive 5µs per operation. We are working with Broadcom to revamp the interface between SoC and the host memory to get closer to native PCIe latencies. Another reason is that the ARM cores run at a 25% lower frequency compared to the x86 cores.

(2) Remote SSD (realSoC vs. SoC$_{VM}$). Next, to measure remote SSD performance, we repeat the setup in §5.1.4 (YCSB/RocksDB/gOS/gVM/**SoC$_{VM}$**/–RDMA–/**SoC$_{VM}$-or-realSoC**/SSD). Figures 10b&d show that realSoC on remote side (and SoC$_{VM}$ on client side) has a *minimal overhead* compared to the previous setting (only 5% throughput reduction and 10% latency overhead at p99) because the remote runtime does not need to communicate with the remote host, hence does not suffer from any overheads. However, we note that the overheads would be similar to the previous experiment when both sides use realSoC.

obs Overall, although current realSoC-LocalSSD is up to 30% slower (will be improved in our future SoC), our cost benefit analysis shows that using even 4× more cores in realSoC compared to the number of cores in SoC$_{VM}$ to achieve performance parity still pays off. From the second experiment, we show that x86 is an overkill for polling and ARM co-processors can easily take over the burden when serving SSDs over the network.

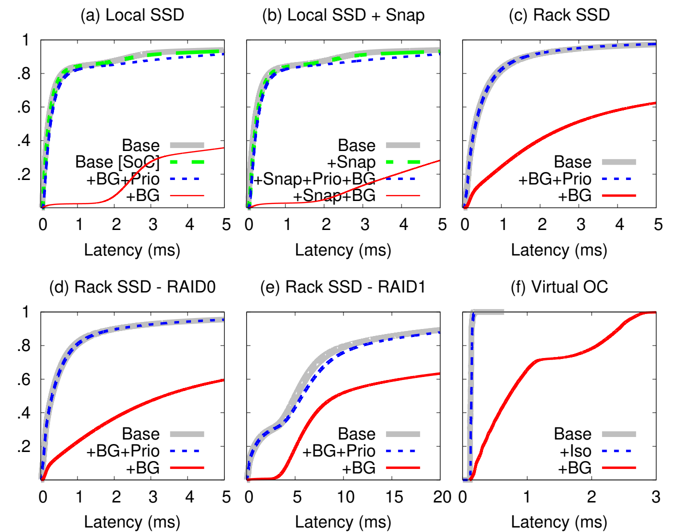


Figure 11. Service features. *The figures are described in §5.3(a)-(f). The experiments are done on SoC$_{VM}$ for faster evaluation. In (a), we run one experiment on SoC "[SoC]" to show a similar pattern.*

5.3 Composability

LeapIO runtime enables composing services in easy ways. Like filtering operations in networking, LeapIO services get a command, process it, and then either send a completion back to the upstream queue or forward sub-commands to many downstream queues (*e.g.*, striping). Composability is achieved by chaining and striping filters. For instance, one simple example we demonstrate later is combining priority and snapshot services.

We build various storage services that compose local/remote devices as well as remote services in just 70 to 4400 LOC each, all in *user space* on LeapIO runtime. In all the experiments below, we use a "search-engine" workload trace containing read-only, user-facing index lookups. We take 1 million IOs, containing various IO sizes from 4K to 7M bytes with average and median size of 36K and 32K bytes respectively. We also co-locate the search workload with a background ("**BG**") workload that performs intensive read/write IOs such as rebuilding the index. The purpose of using a real search-engine trace is as a case study of migrating latency-sensitive services from dedicated servers to a shared cloud, thereby making latency-sensitive services more elastic with resources in proportion to the load.

(a) Prioritization service. A crucial enabler for high cloud utilization is the ability to prioritize time-sensitive, user-facing queries over non-interactive background workloads such as index rebuilding. For this, we build a new service that prioritizes interactive workloads while keeping the batch processing workload make meaningful forward progress when hosts are under-utilized.

The "`Base`" line in Figure 11a shows the latency CDF of the search-engine VM without contention. However, when

co-located with batch workloads (BG), the search VM suffers long latencies ("+BG" line). With our prioritization service, the search VM observes the same performance as if there were no contention ("+BG+Prio" line). At the same time, the batch workload obtains 10% of the average resources to make meaningful progress (not shown).

(b) Snapshot/version service. Another important requirement of search engines is to keep the index fresh. The index-refresher job must help foreground jobs serve queries with the freshest index. It is undesirable to refresh the index in an offline manner where the old index is entirely rejected and a new one is loaded (causing invalidated caches and long tail latencies). A more favorable way is to update the index gradually.

For this, we build two new services. The first service implements a snapshot feature (the index-refresher job). Here, all writes are first committed to a log using our NVMe multi-block atomic write command while a background thread gradually checkpoints them (while the foreground thread serves consistent versions of the index). The second service is a search VM that looks up the log and obtains blocks of the version they need if they are present in the log and reads the remaining data from the SSD.

Figure 11b shows that this snapshot-consistent read feature adds a slightly longer latency to the base non-versioned reads ("+Snap" vs "Base" lines). When combined with the background writer, the snapshot-consistent reads exhibit long latencies ("+Snap+BG" line). Here we can easily compose our snapshot-consistent and prioritization features in LeapIO (the "+Snap+Prio+BG" line).

(c) Remote rack-local SSD. Decoupling compute and storage is a long standing feature of many storage services. We want to decouple the search service also from its storage. Figure 11c shows the results the same workload used in Figure 11b (prioritization) but now the storage is a remote SSD (in the local rack shared by multiple search VMs). The experiment shows that the prioritization mechanism works end-to-end even with the network now in the data path.

(d) Agile rack-local RAID. Our servers that power search engines require disproportionately larger and more powerful SSDs compared to traditional VM workloads. Currently, this means we must overprovision SSD space and bandwidth to keep the fleet uniform and fungible. With LeapIO, we propose *not* to overprovision the dedicated SSDs but rather build larger composable virtual rack-local SSDs.

More specifically, in every rack, each server publishes the free SSD IOPS, space and available network bandwidth that it can spare to a central known billboard every few minutes. Any server that needs to create a virtual drive beyond the capacity of its free space consults the billboard. It then sequentially contacts each server directly to find out if it still has the necessary free capacity until one of them responds affirmatively. In such a case, they execute a peer-to-peer transaction with a producer/consumer relationship and establish a direct data path between the two. The consumer uses the additional space to augment its local SSDs to support the search VMs in the rack which are interested in this partition of the index.

Figure 11d shows the same experiments in Figure 11c but now the backend drive is a RAID-0 of two virtual SSDs (more is possible) in two machines, delivering a higher performance and capacity for the workload.

(e) Rack-local RAID 1. To protect against storage failures, one can easily extend RAID 0 to RAID 1 (or other RAID protection levels). Figure 11e shows the results with RAID-1 of two remote SSDs as the backend. Note that we lose the performance of RAID-0 but now get reliability.

(f) Virtualized Open Channel service for isolation. Another related approach to prioritization is isolation – different VMs use isolated virtual drives within the same SSD. For this, we compose a *unique stack* enabled by LeapIO: gVM/LeapIOClient/OC (where "**OC**" denotes OpenChannel SSD [32, 55]). OC can be configured to isolate flash channels for different tenants [41]. Unfortunately, OC *cannot* be virtualized across multiple VMs, because LightNVM must run in the host OS and directly talks to OC.

With LeapIO, we can *virtualize OC*. Guest VM/OSes can run LightNVM not knowing that underneath it LeapIO remaps the channels. As an example, a guest VM can ask for 4 channels and our new OC service can map the requested channels to the local (or even remote) OC drives. Hence, our new OC service is capable of *exposing virtual channels* and allow guest VMs to reap OC performance isolation benefits. In Figure 11f, when a VM (running basic FIO) competes with another write-heavy VM on a shared SSD, the FIO latencies are heavily affected ("Base" vs. "+BG" lines). However, after we dedicate different channels for these two VMs, the search engine performance is now isolated ("+Iso" line).

6 Conclusion

LeapIO is our next-generation cloud storage stack that leverages ARM SoC to alleviate taxing x86 CPUs. Our experience and experiments with LeapIO show that the engineering and performance overhead of moving from x86 to ARM is minimal. In the shorter term, we will continue to move existing host storage services to LeapIO, while our longer term goal is to develop new capabilities that allow even the guest software stack to be offloaded to ARM.

7 Acknowledgments

We thank the anonymous reviewers for their tremendous feedback and comments. The university authors were supported by funding from NSF (grant Nos. CNS-1350499, CNS-1526304, CNS-1405959, CNS-1563956).

Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Introduction to NVMe. https://www.osr.com/nt-insider/2014-issue4/introduction-nvme-technology, 2014.

[2] NVM Express over Fabrics Revision 1.0. https://nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605-1.pdf, 2016.

[3] Alibaba: Using SPDK in Production. https://ci.spdk.io/download/events/2018-summit/day1_08_ShengMingAlibaba.pdf, 2018.

[4] BlueField SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic, 2018.

[5] NVMe Is The New Language Of Storage. https://www.forbes.com/sites/tomcoughlin/2018/05/03/nvme-is-the-new-language-of-storage, 2018.

[6] p2pmem: Enabling PCIe Peer-2-Peer in Linux. https://www.snia.org/sites/default/files/SDC/2017/presentations/Solid_State_Stor_NVM_PM_NVDIMM/Bates_Stephen_p2pmem_Enabling_PCIe_Peer-2-Peer_in_Linux.pdf, 2018.

[7] *Smart SSD: Faster Time To Insight*, 2018. https://samsungatfirst.com/smartssd/.

[8] Stingray 100GbE Adapter for Storage Disaggregation over RDMA and TCP. https://www.broadcom.com/products/storage/ethernet-storage-adapters-ics/ps1100r, 2018.

[9] Accelerating NVMe I/Os in Virtual Machines via SPDK vhost. https://www.lfasiallc.com/wp-content/uploads/2017/11/Accelerating-NVMe-I_Os-in-Virtual-Machine-via-SPDK-vhost_Ziye-Yang-_-Changpeng-Liu.pdf, 2019.

[10] Broadcom Announces Availability of Industry's First Universal NVMe Storage Adapter for Bare Metal and Virtualized Servers. https://www.globenewswire.com/news-release/2019/08/06/1897739/0/en/Broadcom-Announces-Availability-of-Industry-s-First-Universal-NVMe-Storage-Adapter-for-Bare-Metal-and-Virtualized-Servers.html, 2019.

[11] Cloud Storage Market worth 88.91 Billion USD by 2022. https://www.marketsandmarkets.com/PressReleases/cloud-storage.asp, 2019.

[12] Flexible I/O Tester. https://github.com/axboe/fio.git, 2019.

[13] Full Virtualization. https://en.wikipedia.org/wiki/Full_virtualization, 2019.

[14] In-Hardware Storage Virtualization - NVMe SNAP Revolutionizes Data Center Storage. http://www.mellanox.com/related-docs/solutions/SB_Mellanox_NVMe_SNAP.pdf, 2019.

[15] iSCSI: Internet Small Computer Systems Interface. https://en.wikipedia.org/wiki/ISCSI, 2019.

[16] NGD Newport Computational Storage Platform. https://www.ngdsystems.com, 2019.

[17] NVM Express Revision 1.4. https://nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.4.pdf, 2019.

[18] RocksDB - A Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org, 2019.

[19] Samsung Key Value SSD Enables High Performance Scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf, 2019.

[20] Single-Root Input/Output Virtualization. http://www.pcisig.com/specifications/iov/single_root, 2019.

[21] SPDK Fails to Come Up After Long FIO Run. https://github.com/spdk/spdk/issues/691, 2019.

[22] SPDK NVMf Target Crashed While Running File System. https://github.com/spdk/spdk/issues/763, 2019.

[23] SPDK Performance Very Slow. https://github.com/spdk/spdk/issues/731, 2019.

[24] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen mei Hwu. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[25] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[26] Nils Asmussen, Marcus Volp, Benedikt Nothen, Hermann Hartig, and Gerhard Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[27] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[28] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[29] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.

[30] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[31] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[32] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[33] Adrian M. Caulfield and Steven Swanson. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[34] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, 2013.

[35] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[36] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[37] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[38] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[39] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek

Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[40] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[41] 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 *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[42] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[43] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[44] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling Cost-effective Data Processing with Smart SSD. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[45] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[46] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISKs). In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1998.

[47] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. HyperLoop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[48] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[49] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[50] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, 2017.

[51] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual Remote I/O. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[52] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T.V. Lakshman. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[53] Sungjin Lee, Ming Liu, Sang Woo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[54] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[55] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[56] Ming Liu, Tianyi Cui, Henrik Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. iPipe: A Framework for Building Distributed Applications on Multicore SoC SmartNICs. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[57] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[58] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, 2016.

[59] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[60] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[61] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[62] Pierre Olivier, A K M Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, and Binoy Ravindran. HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost, Low-Power Embedded Systems. In *Proceedings of the 28th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2019.

[63] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[64] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[65] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[66] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, 1998.

[67] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[68] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[69] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[70] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[71] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[72] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct Universal Access: Making Data Center Resources Available to FPGA. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[73] Mark Silberstein. OmniX: an accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.

[74] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[75] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[76] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[77] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. BlueCache: A Scalable Distributed Flash-based Key-value Store. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB)*, 2016.

[78] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.