NeSC: A self-virtualizing, nested storage controller

Yonatan Gottesman

NeSC: A self-virtualizing, nested storage controller

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering

Yonatan Gottesman

Submitted to the Senate
of the Technion — Israel Institute of Technology
Kislev 5777 Haifa December 2016

This research was carried out under the supervision of Prof. Yoav Etsion, in the Faculty of Electrical Engineering.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's master research period, the most up-to-date versions of which being:

Y. Gottesman and Y. Etsion , In Intl. Symp. on Microarchitecture (MICRO-49), Oct 2016

Acknowledgements

I would like to thank my advisor, my parents, my friends, and my beloved wife.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

The emergence of high-speed, multi GB/s storage devices has shifted the performance bottleneck of storage virtualization to the software layers of the hypervisor. The hypervisor overheads can be avoided by allowing the virtual machine (VM) to directly access the storage device (a method known as direct device assignment), but this method voids all protection guarantees provided by filesystem permissions, since the device has no notion of client isolation. Recently, following the introduction of 10Gbs and higher networking interfaces, the PCIe specification was extended to include the Single Root IO Virtualization (SR-IOV) specification for self-virtualizing devices, which allows a single physical device to present multiple virtual interfaces on the PCIe interconnect. Using SR-IOV, a hypervisor can directly assign a virtual PCIe device interface to each of its VMs. However, as networking interfaces simply multiplex packets sent from/to different clients, the specification does not dictate the semantics of a virtual storage device and how to maintain data isolation in a self virtualizing device.

In this research we present the self-virtualizing, nested storage controller (NeSC) architecture, which includes a filesystem-agnostic protection mechanism that enables the physical device to export files as virtual PCIe storage devices. The protection mechanism maps file offsets to physical blocks and thereby offloads the hypervisor's storage layer functionality to hardware. Using NeSC, a hypervisor can securely expose its files as virtual PCIe devices and directly assign them to VMs. We have prototyped a 1GB/s NeSC controller using a Virtex-7 FPGA development board connected to the PCIe interconnect. Our evaluation of NeSC on a real system shows that NeSC virtual devices enable VMs to access to their data with near-native performance (in terms of both throughput and latency).

Chapter 1

Introduction

The prevalence of machine consolidation through the use of virtual machines (VMs) necessitates improvements in VM performance. On the architecture side, major processor vendors have introduced virtualization extensions to their instruction set architectures [PG74, Int15a, ARM16]. The emergence of high-speed (10Gbe and faster) networking interfaces also requires that VMs be granted direct access to physical devices, thereby eliminating the costly, software-based hypervisor device multiplexing.

Enabling untrusted VMs to directly access physical devices, however, compromises system security. To overcome the fundamental security issue, the PCIe standard was extended to support self-virtualizing devices through the Single-Root I/O Virtualization (SR-IOV) interface [PS09]). This method enables a physical device (*physical function* in SR-IOV parlance) to dynamically create virtual instances (*virtual functions*). Each virtual instance receives a separate address on the PCIe interconnect and can, therefore, be exclusively assigned to, and accessed by, a specific VM. This method thereby distinguishes between the physical device, managed by the hypervisor, and its virtual instances used by the VMs. Importantly, it is up to the physical device to interleave and execute requests issued to the virtual devices.

Self-virtualizing devices thus delegate the task of multiplexing VM requests from the software hypervisor to the device itself. The multiplexing policy, however, depends on the inherent semantics of the underlying device and must, naturally, isolate request streams sent by individual virtual devices (that represent client VMs). For some devices, such as networking interfaces, the physical device can simply interleave the packets sent by its virtual instances (while protecting the shared link state [SBYT15]). However, enforcing isolation is nontrivial when dealing with storage controllers/devices, which typically store a filesystem structure maintained by the hypervisor. The physical storage controller must therefore enforce the access permissions imposed by the filesystem it hosts. The introduction of next-generation, commercial PCIe SSDs that deliver multi-GB/s bandwidth [Int15b, Sea16]) emphasizes the need for self-virtualizing storage technology.

In this research we present NeSC, a self-virtualizing nested storage controller that

enables hypervisors to expose files and persistent objects[1](or sets thereof) as virtual block devices that can be directly assigned to VMs. NeSC enforces the permissions set by the hypervisor and the hosted filesystem and prevents virtual devices (and VMs) from accessing stored data for which they have no access permissions.

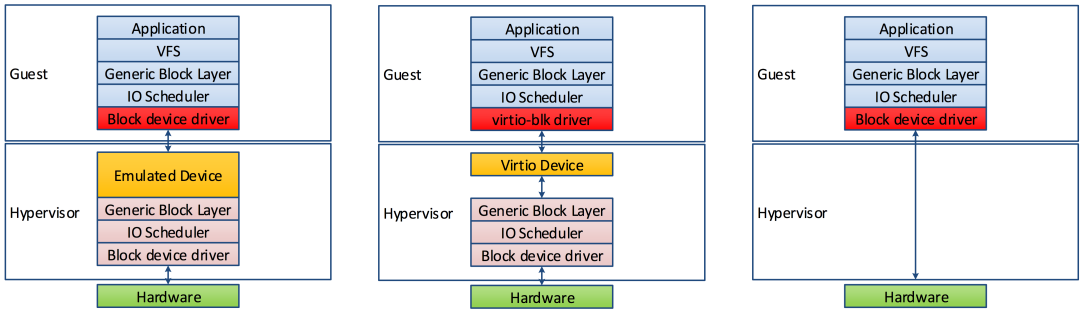
NeSC enforces isolation by associating each virtual NeSC device with a table that maps offsets in the virtual device to blocks on the physical device. This process follows the way filesystems map file offsets to disk blocks. VMs view virtual NeSC instances as regular PCIe storage controllers (block devices). Whenever the hypervisor wishes to grant a VM direct access to a file, it queries the filesystem for the file's logical-to-physical mapping and instantiates a virtual NeSC instance associated with the resulting mapping table. Each access by a VM to its virtual NeSC instance is then transparently mapped by NeSC to a physical disk block using the mapping table associated with the virtual device (e.g., the first block on the virtual device maps to offset zero in the mapped file).

We evaluate the performance benefits of NeSC using a real working prototype implemented on a Xilinx VC707 FPGA development board. Our evaluation shows that our NeSC prototype, which provides 800MB/s read bandwidth and almost 1GB/s write bandwidth, delivers 2.5× and 3× better read and write bandwidth, respectively, compared to a paravirtualized *virtio* [Rus08] storage controller (the de facto standard for virtualizing storage in Linux hypervisors), and 4× and 6× better read and write bandwidth, respectively, compared to an emulated storage controller. We further show that these performance benefits are limited only by the bandwidth provided by our academic protoype. We expect that NeSC will greatly benefit commercial PCIe SSDs capable of delivering multi-GB/s of bandwidth.

[1]We use the terms files and objects interchangeably.

Chapter 2

IO Virtualization



(a) Emulation (b) virtio (c) Direct-IO

Figure 2.1: IO Virtualization techniques.

2.1 Virtualization Background

Virtualization is the act of logically dividing the system resources between different entities running on it. Virtualization hides the physical characteristics of a computing platform from the users, presenting instead another abstract computing platform. The software that controls the virtualization of the system is called *hypervisor* or *virtual machine monitor* (VMM).

Virtualization is performed on a given hardware platform by host software (hypervisor), which creates a simulated computer environment, a virtual machine (VM), for its guest software. The guest software is not limited to user applications; many hosts allow the execution of complete operating systems. The guest software executes as if it were running directly on the physical hardware, with several notable caveats. Access to physical system resources (such as the network access, display, keyboard, and disk storage) is generally managed at a more restrictive level than the host processor and system-memory. Guests are often restricted from accessing specific peripheral devices, or may be limited to a subset of the device's native capabilities, depending on the

hardware access policy implemented by the virtualization host. Virtualization often exacts performance penalties, both in resources required to run the hypervisor, and as well as in reduced performance on the virtual machine compared to running native on the physical machine.

Virtualization Aspects

There are three main resources that the hypervisor needs to virtualize: the CPU, Memory and I/O devices.

CPU A common implementation of CPU virtualization is *Trap-and-Emulate*. When using this technique, the guest code (OS or user application) may run directly on the CPU and only when an unprivileged command is executed, the CPU will issue a trap to the hypervisor that will emulate the guest command. To configure the instructions that cause a VM to trap, modern CPUs expose a special interface (Intel's VT-x and AMD AMD-V) for the hypervisor to program.

Memory Just like any other process running on the system, a guest OS has no control over the hosts physical memory. The hypervisor will have to provide each of the guest OSs with the illusion of a more or less contiguous block of physical memory starting at address 0, because that's what all OSs expect. The simplest way to do that is to introduce one more level of virtualization, one more level of mapping between what the guest OS considers a physical address (it's called *guest physical address*) and the actual physical address (which is called *host physical address*). The hypervisor isolates the running VM with page tables that translate *guest physical address* to *host physical address*. When a guest application is running, each memory access (to *guest virtual address*) must pass two translation layers, the first translates to *guest physical address*, and the second translates to *host physical address*.

Modern processors use *Extended Page Tables* (EPT) technology. This technology enables the *MMU* to do this double translation: First it translates the *guest virtaul address* to *guest physical address* using the page tables set by the guest OS. Then it translates the *guest physical address* to *host physical address* using the page tables set by the hypervisor.

Another technique used on systems without EPT support is called *shadow page tables*. The idea is to create a second copy of VM's page tables called shadow page tables that map *guest virtual addresses* directly to host physical addresses (actual DRAM), and let the processor use them for address translation (so the CR3 register points to those, rather than to guest, or primary page tables).

I/O devices Its the hypervisor's responsibility to share an I/O device between virtual machines. When sharing a device, isolation must be enforced so that one VM cannot access other VMs' resources on the same I/O device. Figure 2.1 illustrates the three

most common methods by which hypervisors virtualize storage devices (We discuss storage devices, but the same technologies work for any I/O device):

1. *Full device emulation* [SVL01] (Figure 2.1a) In this method, the host emulates a well known hardware device. The hypervisor attaches the emulated device to the VM's physical address space (*guest physical address*), and the gust OS will discover the device while scanning the system for physical devices. When the device is discovered the OS will load the appropriate driver. Whenever the guest will access the address space representing the device (a PCIe function for example), a trap will be issued by the processor, and the hypervisor will emulate the behavior of the device for that access.

   When the device is a block device, the emulated device is represented as a file on the host filesystem. Whenever the guest tries to access a *logical block address* (LBA) on the device, the host converts the LBA to an offset in the file, and depending on the command, reads/writes data from/to that offset.

   The advantage of this method is that the guest doesn't require any changes and can run unmodified device drivers. The disadvantage is the high latency of each request, that has to pass through all the block I/O software layers in the guest, and then again in the host. The overheads of device emulation is discussed later.

2. *Paravirtualization* [BDF+03, Rus08] (Figure 2.1b) This method eliminates the need for the hypervisor to emulate a complete physical device and enables the guest VM to directly request a virtual LBA from the hypervisor, thereby improving performance. The hypervisor still emulates a device, but the guest knows the device is emulated and not real and can optimize the interface between it and the hypervisor. This is the most common storage virtualization method used in modern hypervisors.

3. *Direct device assignment* [RS07] (Figure 2.1c) This method allows the guest VM to directly interact with the physical device without hypervisor mediation. Consequently, it delivers the best storage performance to the VM. However, since legacy storage devices do not enforce isolation among clients, it does not allow multiple VMs to share a physical device. The new PCIe protocol's SR-IOV feature allows multiple clients to access the same physical device while enforcing isolation at the hardware level. We discuss SR-IOV in section 2.3

2.2 The I/O Stack

The I/O stack is the set of abstractions an I/O request passes starting at the application issuing the read/write request, and ending at the data written/read to the physical storage device.

I/O Stack Components

Filesystem A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. Every filesystem implements a mapping mechanism in which file offsets are mapped to physical storage blocks. Traditionally, UNIX-derived filesystems used per-file direct and indirect block mapping tables to map offsets in a file to their corresponding data block. But tracking individual blocks incurs large spatial and latency overheads when dealing with large files. Modern UNIX filesystems (e.g., ext4 [MCB+07],btrfs [RBM13], xfs [SDH+96]) therefore group contiguous physical blocks into *extents* and construct extent trees, which consist of variants of B-trees [Com79], to spatially map offsets in the device to extents. Each file is associated with an extent tree (pointed to by the file's *inode*) that maps file offsets to physical blocks.

Block Layer The block layer is a kernel component that handles the requests for all block devices in the system. A request can reach this layer either from the filesystem layer, or directly from an application reading/writing straight to the raw device.

Device Driver The role of device drivers is to communicate with I/O devices attached to the system. Block device drivers get block read/write requests from the block layer in the form of a *bio struct*. The bio object contains everything that a block driver needs to carry out the request to the block device, including the LBA of the request, the size and the virtual address from where the data is copied from/to.

Block Device A block device is a computer data storage device that supports reading and writing data in fixed-size blocks. These blocks are generally 512 bytes or a multiple thereof in size. The term is often used in contrast with a "word-addressed device" which supports reading and writing data a word at a time, where a "word" is a much smaller block, usually 1 to 8 bytes in size.

2.3 Single Root I/O Virtualization

The single root I/O virtualization (SR-IOV) interface is an extension to the PCI Express (PCIe) specification. SR-IOV allows a device to separate access to its resources among various PCIe hardware functions. These functions consist of the following types:

- *Physical Function (PF)* The PF is a full-featured PCIe function. The PF includes the SR-IOV capability in the PCIe Configuration space. The capability is used to configure and manage the SR-IOV functionality of the device, such as enabling virtualization and exposing VFs.
- *Virtual Function (VF)* A VF is a lightweight PCIe function that supports the SR-IOV interface. The VF represents a virtualized instance of the physical device.

Each VF has its own PCI Configuration space. Each VF also shares one or more physical resources on the physical device with the PF and other VFs.

SR-IOV extends the direct device assignment method discussed earlier to allow multiple virtual machines to directly access the same physical device. This is done by creating multiple VFs' and attaching each of them to a different VM. Isolation of the resources is done by the device and the hypervisor doesn't intercept every access done to the device.

NVM Express

NVM Express (NVMe) is a logical device interface specification for accessing non-volatile storage media attached via PCI Express (PCIe) bus. The NVMe specification defines how an SSD device divides its resources among several VFs'. The hypervisor divides the storage blocks into *namespaces* and defines a mapping between a VF to a *namespace*. When a VM accesses a VF, the requests are translated to the correct *namespace*.

Chapter 3

Motivation

As described in chapter 2, hypervisors virtualize local storage resources by mapping guest storage devices onto files in their local filesystem, in a method commonly referred to as a nested filesystem [LHW12]. As a result, they replicate the guest operating system's (OS) software layers that abstract and secure the storage devices. Notably, these software layers have been shown to present a performance bottleneck even when not replicated [YSS+14], due to the rapid increase in storage device bandwidth [Int15b, Sea16]. Moreover, further performance degradation is caused by the method by which hypervisors virtualize storage devices and the resulting communication overheads between the guest OS and the underlying hypervisor. Consequently, the storage system is becoming a major bottleneck in modern virtualized environments. In this section we examine the sources of these overheads and outline how they can be mediated using a self-virtualizing storage device.

Prevalent storage devices present the software stack with a raw array of logical block addresses (LBA), and it is up to the OS to provide a flexible method to partition the storage resources into logical objects, or files. In addition, the OS must enforce security policies to prevent applications from operating on data they are not allowed to operate on. The main software layer that provides these functionalities is the filesystem, which combines both allocation and mapping strategies to construct logical objects and map them to physical blocks (for brevity, we focus the discussion on these two functionalities and ignore the plethora of other goals set by different filesystems). In addition, the filesystem layer also maintains protection and access permissions. Besides the filesystem, another common layer is the block layer, which caches disk blocks and abstracts the subtleties of different storage devices from the filesystem layer.

When an application accesses a file, the OS uses the filesystem layer to check the access permissions and map the file offset to an LBA on the storage device. It then accesses its block layer, which retrieves the block either from its caches or from the physical device. In a VM, this process is replicated since the storage device viewed by the guest OS is actually a virtual device that is mapped by the hypervisor to a file on the host's storage system. Consequently, the hypervisor invokes its own filesystem and

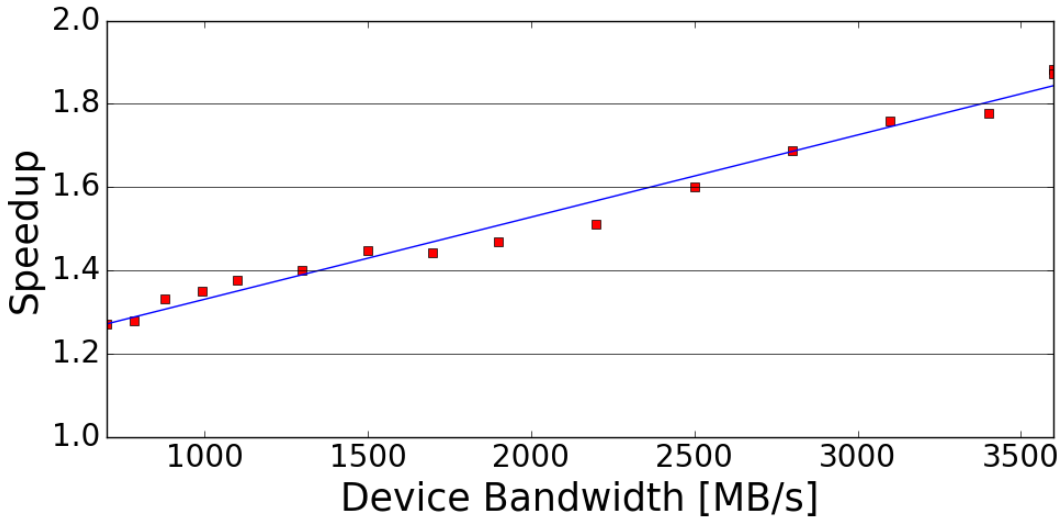


Figure 3.1: The performance benefit of direct device assignment over virtio for high-speed storage devices. Fast devices were emulated using an in-memory disk (ramdisk) whose bandwidth peaks at 3.6GB/s.

block layers to retrieve the data to the guest OS.

Figure 3.1 quantifies the potential bandwidth speedup of direct device assignment over the common virtio interface for high-speed storage devices. We have emulated such devices by throttling the bandwidth of an in-memory storage device (ramdisk). Notably, due to OS overhead incurred by its software layers, the ramdisk bandwidth peaks at 3.6GB/s.

The figure shows the raw write speedups obtained using direct device assignment over virtio for different device speeds, as observed by a guest VM application. Notably, we see that compared to the state-of-the-art virtio method, direct device assignment roughly doubles the storage bandwidth provided to virtual machines for modern, multi GB/s storage devices. The reason for these speedups is that as device bandwidth increases, the software overheads associated with virtualizing a storage device become a performance bottleneck. Using the direct device assignment method eliminates both the virtualization overheads as well as the overheads incurred by the replication of the software layers in the hypervisor and the guest OS.

The potential performance benefits of direct device assignment motivate the incorporation of protection and isolation facilities into the storage device. These facilities will enable multiple guest VMs to share a directly accessed physical device without compromising data protection.

This research presents the nested storage controller (NeSC), which enables multiple VMs to concurrently access files on the host's filesystem without compromising storage security (when NeSC manages a single disk, it can be viewed simply as a PCIe SSD). Figure 3.2 illustrates how NeSC provides VMs with secure access to a shared physical

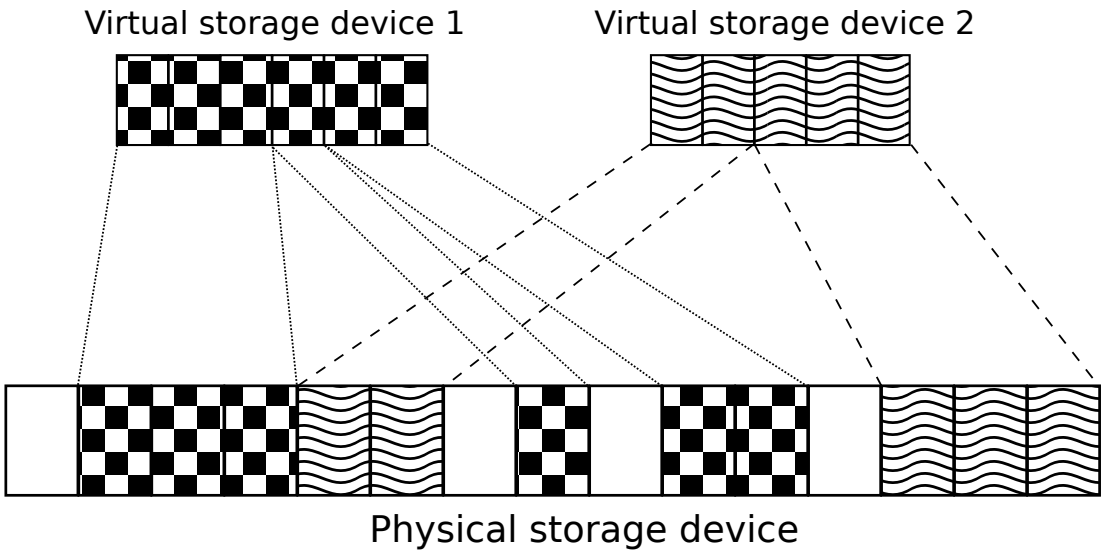


Figure 3.2: Exporting files as virtual devices using NeSC.

device. NeSC leverages the SR-IOV features of the PCIe gen3 [PS09] to export host files as multiple virtual devices on the PCIe address space. Each virtual device is associated with a collection of non-contiguous blocks on the physical device, which represent a file, and is prevented from accessing other blocks. VMs can therefore directly access the virtual device and bypass the virtualization protocol and hypervisor software (notably, NeSC is compatible with the modern NVMe standard [NVM15]).

Chapter 4

Related Work

NeSC examines the design of the hardware/software storage stack for modern, high-speed, self-virtualizing storage devices. In this section we examine related research efforts.

The Moneta [CDC+10] and Moneta-D [CME+12] projects both try to optimize the I/O stack by introducing new hardware/software interfaces. The Moneta project introduces a storage array architecture for high-speed non-volatile memories. Moneta-D extends the design to allow applications to directly access storage by introducing per-block capabilities [Lev84]. When an application directly accesses the storage, permission checks are done by the hardware to preserve protection policies dictated by the OS. The design, however, requires applications to use a user space library to map a file offsets to the physical blocks on the device. In contrast, NeSC enforces file protection by delegating file mapping to the device level rather than to the application level.

Willow [SGB+14] examines offloading computation to small storage processing units (SPU) that reside on the SSD. The SPUs run a small OS (SPU-OS) that enforces data protection. Willow implements a custom software protocol that maintains file mapping coherence between the main OS and the SPU-OS instances.

NVMe [NVM15] is a new protocol for accessing high-speed storage devices. Typically implemented over PCIe, NVMe defines an abstract concept of *address spaces* through which applications and VMs can access subsets of the target storage device. The protocol, however, does not specify how address spaces are defined, how they are maintained, and what they represent. NeSC therefore complements the abstract NVMe address spaces and enables the protocol to support protected, self-virtualizing storage devices.

DFS [**?**] focuses on reducing the overheads imposed by traditional file systems and block device drivers, when using high performance storage devices. The DirectFS relies on the flash storage layer for functionality traditionally implemented in the OS, such as block allocation and is implemented as a kernel-level file system. DFS focuses on optimizing the kernel IO path to better support high speed storage, while NeSC investigates how to access the storage without going through the kernel at all.

Finally, FlashMap [HBQS15] provides a unified translation layer for main memory

and SSDs, which enables applications to map files to their memory address space. Specifically, FlashMap unifies three orthogonal translation layers used in such scenarios: the page tables, the OS file mapping, and the flash translation layer. FlashMap, however, only supports memory-mapped SSD content, and it does not address how the unified translation would support virtualized environments.

In summary, NeSC's self-virtualizing design is unique in that it delegates all protection checks and filesystem mappings to the hardware. Furthermore, its filesystem-agnostic design preserves the hypervisor's flexibility to select and manage the host filesystem.

Chapter 5

NeSC Design

In this chapter we overview the NeSC design principles and basic functionality.

5.1 The NeSC interface

NeSC implements the SR-IOV specification [PS09], which allows an I/O device to be shared by multiple VMs. NeSC exposes two types of functions on the PCIe bus: (1) A *physical function* (2) *virtual functions*, Each according to the SR-IOV standard. A single *physical function* (PF) represents the main device with all its features. Additionally, the device may dynamically expose multiple *virtual functions* (VFs), which typically present to clients (e.g., VMs, accelerators) a subset of features supported by the main device. Importantly, each VF has a unique PCIe address and can receive direct I/O requests from its associated client without hypervisor or OS intervention. Since both the PF and the VFs effectively represent different facets of a single physical device, which multiplexes and processes the communication of all the clients with the different facets, it is thus up to the physical device to determine the behavior of the device exported as the PF and that of a virtual device exposed as a VF.

The physical function

The NeSC PF exports a full-featured PCIe function. It is mapped only to the hypervisor address space and allows it to fully manage the physical device. The PF has two main functionalities:

1. Export a full-featured storage device Through the PF, the hypervisor can access all NeSC's storage blocks without any layers of translation. specifically, This allows the hypervisor to build a filesystem that covers all the storage blocks. With this filesystem (discussed in chapter 5.2) the hypervisor manages NeSC's storage block allocations and access permissions.

2. Export the virtual devices managing interface Through the PF, the hypervisor controls the creation, deletion and management of a VF. When a new VF is

created, the hypervisor defines the subsets of storage it is allowed to access.

Virtual functions

Each NeSC VF is viewed by the system as a PCIe device that provides a complete block device interface. A VF can thus be mapped to a VM's address space and viewed by it as a fully fledged storage device, which can be programmed by the VM's driver to issue read/write commands to the storage device. Technically, the only difference between the PF and VF interface is that a VF is not allowed to create nested VFs (although, in principle, such a mechanism can be implemented to support nested virtualization).

The PF and VF differ semantically in that VFs can only access the subset of the storage they are associated with. When a VM accesses a VF, all of its read/write block requests are translated to the subset of blocks assigned to that VM as described below.

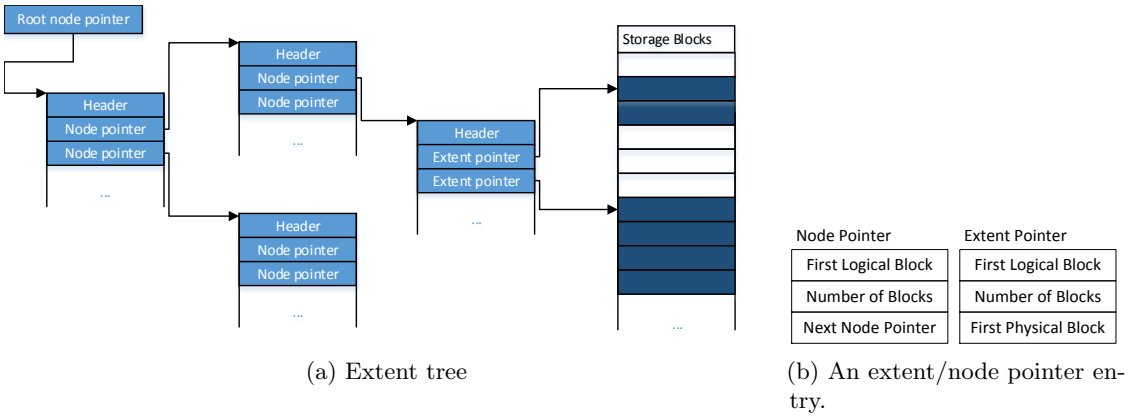
5.2 Virtual-to-physical block mapping

Decoupling the PF from the VFs enables the hypervisor to logically manage the physical device with its own filesystem, and expose files (or collections thereof) to client VMs as raw virtual devices through VFs (as illustrated in Figure 3.2). Each VF is thus associated with a mapping table that translates client requests to physical blocks. Specifically, since client VMs view VFs as regular block devices, they send requests pertaining to LBAs on the virtual device. NeSC refers to client LBAs as virtual LBAs (vLBA), to host LBAs as physical LBAs (pBLA), and the translation process is referred to as a vLBA-to-pLBA translation.

The vLBA-to-pLBA mapping is performed using per-VF mapping tables. The mapping tables are designed as *extent trees*, a method inspired by modern UNIX filesystems.

The key benefit of extent trees is that their depth is not fixed but rather depends on the mapping itself. In ext4, for example, a 100MB file can be allocated using a single extent, thereby obviating the need to create the indirect mappings for each individual block. Extents thus improve performance and also reduce management overheads.

Figure 5.1a illustrates the NeSC VF extent tree (which resembles the ext4 extent tree format). The extent tree is implemented as a B+ tree, each node contains only keys and the leaves contain the data itself. A key is a tuple of (1) First Logical Block, which represents the first vLBA that the sub-tree beneath that key represents (2) Number of Blocks, which represents the amount of blocks the sub-tree represents. The data is an extent, which holds the translation from vLBA to pLBA. Each node in the tree comprises either a group of node pointers (the keys), which point to the next level of nodes, or a group of extent pointers, which are the tree leaves that point to the physical locations on the device. The header of each node indicates whether it contains node indices or extent pointers.



(a) Extent tree

(b) An extent/node pointer entry.

Figure 5.1: The extent tree structure used for address translations.

Figure 5.1b illustrates the content of the entries in each type of node in the extent tree. Each *extent pointer* entry consists of the first logical block it represents, a pointer to the first physical block of its extent, and the size of the extent. Each *node pointer* entry comprises the first logical block it represents, the number of (non contiguous) logical blocks it covers, and a pointer to its array of child nodes. If memory becomes tight, the hypervisor can prune parts of the extent tree and mark the pruned sections by storing NULL in their respective *Next Node Pointer*. When NeSC needs to access a pruned subtree, it interrupts the host to regenerate the mappings.

Each NeSC's VF is associated with an extent tree, which is stored in host memory, and the NeSC architecture (described in Chapter 6) stores the pointer to the root of each VF's extent tree. Whenever a VF is accessed, its extent tree is traversed using DMA accesses from the device to host memory. To mitigate the DMA latency, extents are cached on the NeSC device (more on this in Chapter 6).

Importantly, this use of software-defined, hardware-traversed per-VF extent trees eliminates the need to enforce protection and isolation in the hypervisor software layers, as discussed in Chapter 3. Instead, this task is offloaded to hardware and thereby mitigates one of the key performance bottlenecks of virtualized storage [LHW12].

The per-VF extent tree model also enables the hypervisor to decouple the virtual device's size from its physical layout. This allows the hypervisor to initialize virtual devices whose logical size is larger than their allocated physical space, and to allocate further physical space when needed. This enables the hypervisor to maintain a compact physical representation of the stored data.

Finally, the NeSC design also enables multiple VFs to share an extent tree and thereby files. Nevertheless, NeSC only guarantees the consistency of the extent tree for shared files; it is up to the client VMs to address data synchronization and consistency issues.

Root Filesystem

Through the PF, the hypervisor builds the root filesystem covering all the storage blocks of the device. With this filesystem the hypervisor manages the the allocation of blocks to VMs by using extent based files. Every file on the filesystem represents a virtual disk, and the same mappings used by the filesystem to map file offsets to disk blocks, is used by NeSC to map vLBA accesses to pLBA on the device. In modern filesystems, the extent tree is located on the storage device. Whenever the filesystem traverses the tree, it brings the node from the disk to memory and reads the LBA of the next block to bring. With NeSC, the whole tree must always be in memory because the the device reads from it using DMA operations. Therefore the root filesystem keeps the tree pinned to physical memory and never on the device.

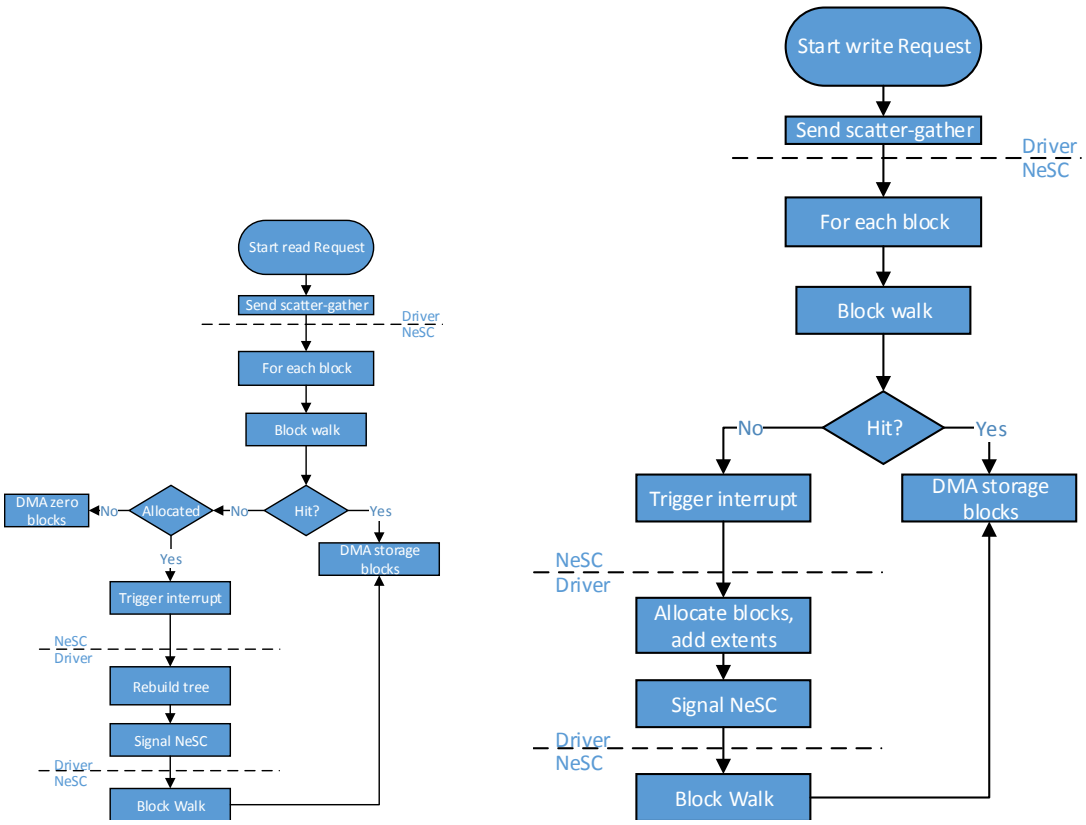
5.3 Operational flow

Creating a new virtual disk

When creating a new virtual device, the hypervisor first creates the extent tree that will map the virtual device to physical blocks. Since the root filesystems uses extent trees to map files to the physical layout, this stage typically consists of translating the root filesystem's own per-file extent tree to the NeSC tree format. The hypervisor will create a file on the root filesystem and the extent tree created will be used by NeSC for translating vLBA accesses for that virtual disk. The hypervisor does not fully preallocate all the physical blocks needed for the new virtual device, this feature is called *delayed allocation* in modern filesystems. When a new file representing a virtual disk is created, the extent tree is empty and no physical blocks are allocated. Regardless of the size of the extent tree, the VF exposes the size of the virtual disk configured by the hypervisor. Blocks will be allocated only on demand, whenever a VM writes to a block that isnt allocated, the hypervisor will allocate the block and update the tree.

The hypervisor then creates a new NeSC VF through the PF. It initializes the VF configuration registers (e.g., the size of the virtual device), writes the extent tree to memory and sets the VF's base extent tree configuration register to point to the in-memory extent tree.

Following the two previous steps, the VF is ready and the virtual device is configured. The hypervisor then connects the VM to the new virtual machine, either by executing a new machine or by notifying an existing machine that a new device is available for its use (virtual device hotplug). The new VF is mapped to a physical address and the hypervisor connects the virtual machine to the VF by mapping its *guest physical address* to the physical address of the VF. The guest OS will probe the PCIe bus and discover the VF.



(a) Read Flow (b) Write Flow

Figure 5.2: Read and write flow in NeSC.

Read flow

The read flow is depicted in Figure 5.2a. The VM's NeSC block driver is attached to the VF and can issue read requests to the device. Large requests are broken down by the driver to scatter-gather lists of smaller chunks. Our NeSC implementation operates at 1KB block granularity (which is, for example, the smallest block size supported by ext4), so the chunks sent by the block driver are broken down by NeSC to 1KB blocks. (more on this in Chapter 6.2) The device then translates each 1024 byte request address through the extent tree mappings of that virtual function and creates a new queue of physical requests to read from physical storage and DMA back to the host memory.

The translation of a vLBA to a pLBA can fail in one of two cases: 1) The pLBA was never allocated due to lazy allocation. Following the POSIX standard, which dictates that unmapped areas inside a file (holes in the file) should read as zeros, NeSC transparently DMAs zeros to the destination buffer in host memory; or 2) The pLBA was allocated, but its mapping was pruned from the extent tree due to memory pressure. NeSC identifies this case when it encounters a NULL node pointer when traversing the tree, and resolves it by sending an interrupt to the host and requesting that the hypervisor regenerates the pruned mappings (e.g., by re-reading the filesystem-specific

mappings from disk).

Write flow

The write flow is shown in Figure 5.2b. Just like read requests, write requests are broken down by NeSC to 1KB chunks.

For each chunk, the device tries to perform vLBA-to-pLBA mapping using the VF's extent tree. If the translation succeeds, the data is written to the persistent physical storage.

Just like in the read case, the translation can fail due to unallocated blocks or pruned tree nodes. Both cases require NeSC to interrupt the hypervisor, which will request the filesystem for the mappings and rebuild the tree. The filesystem might have to allocate new blocks if this is the first time the vLBA is accessed. Once the hypervisor finishes handling the request, it signals NeSC to restart the extent tree lookup, which is now guaranteed to succeed. If, however, the hypervisor cannot allocate more space for the VF (not shown in Figure 5.2b) due to a lack of physical storage or exhausted storage quotas, it signals an error to the PF which, in turn, triggers the VF to send a write failure interrupt to the requesting VM (we note that it is possible to optimize this process by delegating the interrupt directly to the VM [GAH+12]).

5.4 Other design issues

Nested filesystems

A client VM will often manage its own filesystem inside its nested storage device. Given that a virtual device is stored as a file in the hypervisor's filesystem, this use case is commonly referred to as a nested filesystem.

Modern filesystems frequently use journaling for improved fault tolerance. This causes a well-known inefficiency in nested filesystems known as nested journaling [LHW12]. The inefficiency is caused as both the internal and external filesystems redundantly log the internal filesystem's data and meta-data updates. The common solution to this inefficiency is to tune the hypervisor's filesystem to only log meta-data changes for the file at hand and let the VM handle its internal filesystem's data integrity independently.

NeSC naturally lends itself to this common solution. Since NeSC VM clients directly access their data, the hypervisor's filesystem is not aware of the internal filesystem updates, whose integrity it handled by the VM, and only tracks its own meta-data updates.

Direct storage accesses from accelerators

A straightforward extension of NeSC is to export data to accelerators in the system. Traditionally, when an accelerator on the system wants to access storage, it must use the host OS as an intermediary and thereby waste CPU cycles and energy.

While not implemented in the prototype, we note that NeSC can be easily extended to enable direct accelerator-storage communications. This can be easily achieved by modifying the VF request-response interface, which is suitable for block devices, to a direct device-to-device DMA interface (in which offset 0 in the device matches offset 0 in the file, and so on.

This simple change to the VF interface enables accelerators to directly access storage using DMA, without interrupting the main processor and the OS. Such a design directly corresponds with the data-centric server concept presented by Ahn et al. [AKK+15]

Quality of Service (QoS)

Modern hypervisors support different QoS policies for each emulated disk image. Because the hypervisor mediates every disk access, it can easily enforce the QoS policy for each device. NeSC can be extended to enforce the hypervisor's QoS policy by modifying its DMA engine to support different priorities for each VF. Similar techniques are widely used by NICs that support SR-IOV.

Unified buffer cache

Different VMs often share storage blocks with one another. To minimize buffer duplication across VMs' buffer caches, common hypervisors use their own buffer cache as a unified buffer cache by forcing VMs to minimize their individual buffer caches (specifically, balloon drivers in each VM reclaim physical pages; the resulting memory pressure forces the VM to free buffered pages). As NeSC enables VMs to directly access storage, it prevents the hypervisor from maintaining a unified buffer cache. Instead, NeSC relies on memory deduplication to eliminate buffer duplication across guest VMs'. Notably, memory deduplication is supported by all modern hypervisors (e.g., TPS in VMware, KSM in kvm and "Memory CoW" in Xen).

Chapter 6

NeSC Architecture

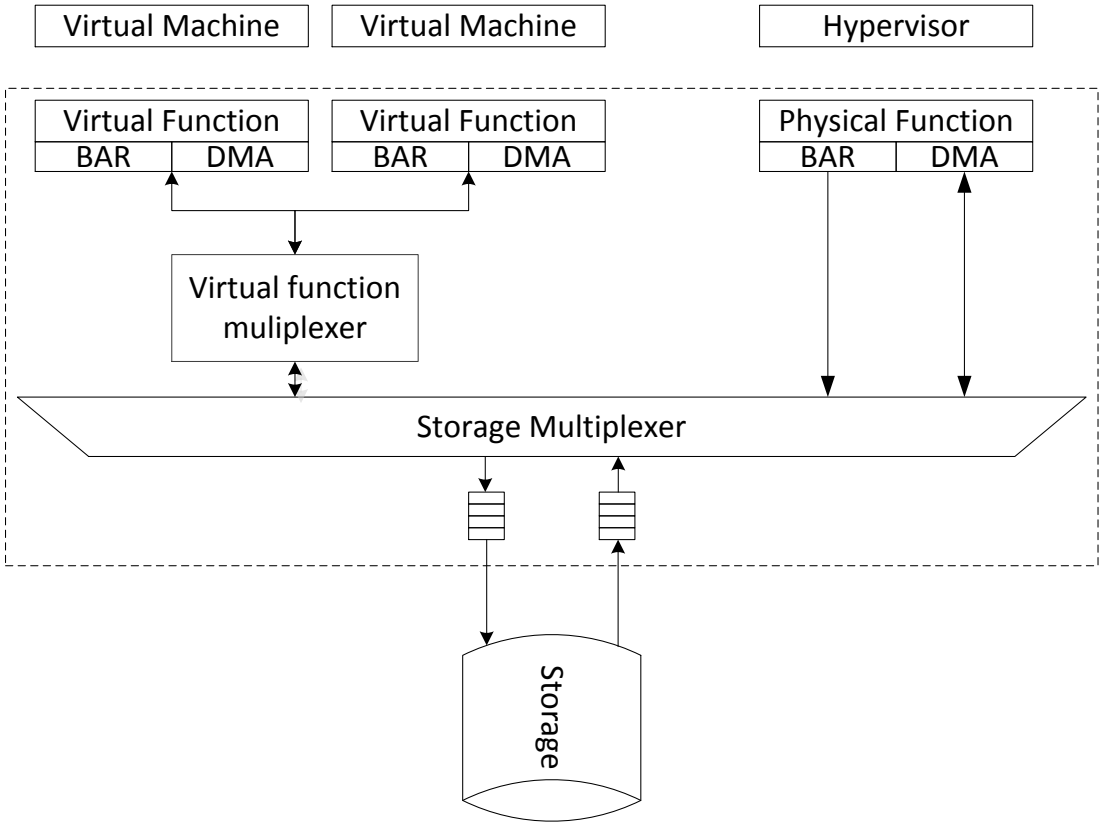


Figure 6.1: An outline of the NeSC architecture. The PF and VFs each have a set of control registers and request/response queues. The core NeSC microarchitecture multiplexes the activity of the individual VFs.

As an SR-IOV device, NeSC presents itself as multiple devices on the PCIe interconnect. The architecture must maintain a separate context for each PCIe device (PF and VFs), but a key tenet of the microarchitecture is multiplexing traffic from the different devices.

Figure 6.1 outlines the core design of the NeSC architecture. For each function

(PF and VF alike), NeSC maintains a set of control registers and request/response queues. All requests sent to the different functions are multiplexed through the address translation and request service units. Similarly, all traffic between the host and the device is multiplexed through a single DMA engine.

6.1 PCIe Interface

PCIe Transactions

The PCIe interconnect is based on point-to-point topology, with separate serial links connecting every device to the root complex (host). Communication between devices to themselves, and between devices to the root complex is done by sending and receiving PCIe requests (configuration, I/O or memory read/write). These requests form a *Transactional Layer Packet* (TLP) and carry the request info (type, address, size etc.). Whenever the CPU accesses the device's address space, a TLP is generated and sent by the root complex to NeSC. Every request received by the NeSC device is labeled with the ID of the NeSC function to which it was sent. This enables the multiplexed facilities to associate requests with a specific function (according to the SR-IOV specification, the ID of the PF is 0). PCIe addressing is hierarchical, and each entity on the interconnect is identified by a triplet: the PCIe *bus ID*, the ID of the device on the bus (*device ID*), and the function inside the device (*function ID*). In PCIe parlance, addresses are referred to as *bus:device:function* triplets or *BDF* for short. Since PCIe addresses of the different NeSC functions share the same bus and device IDs, associating each request with its originating function ID is sufficient for accurate bookkeeping. The BDF triplet is originated by the PCIe root complex and is unforgeable by a virtual machine. The VM has access only to a specific VF, therefore the request generated by the root complex will contain the BDF concurrent to that VF.

Configuration Space

PCIe devices have a set of registers referred to as configuration space. Configuration space registers are mapped to memory locations. Device drivers must have access to the configuration space, and operating systems typically use APIs to allow access to device configuration space. Communication with a PCIe device is handled through *base address registers* (BARs). BARs have a predetermined size, and are mapped to the system's logical address space when the PCIe interconnect is scanned (typically by the system's firmware). Since each BAR is assigned a logical bus address, the hypervisor can map the BARs directly to clients' virtual address space, and to its own.

Although NeSC assigns a BAR for each function (physical and virtual alike), the control registers for all functions can be mapped to a single physical structure inside the device. The PCIe specifications make it possible to define control registers as offsets into a device's BAR. When a certain offset inside a BAR is read from or written to, the

PCIe device internally translates the offset to an address in a shared structure. Since our prototype can support up to 64 VFs (and the additional PF), the NeSC prototype uses a single 130KB SRAM array (2048B per function).

The set of VF-specific control registers are listed as follows (the PCIe specification mandates a number of additional standard control registers, omitted for brevity).

1. **DMAListEntry** (8B) This register is where the driver writes the nodes of a scatter-gather list. Each node that is written here is immediately appended to a list NeSC maintains and a new node can override the old value. The driver sends the scatter-gather list to NeSC by writing all the nodes to this register one by one.

2. **ExtentTreeRoot** (8B) This register contains the base address (in host memory) of the root node of the extent tree associated with a VF. It is set by the hypervisor when a new VF is created.

3. **MissAddress, MissSize** (8B, 4B) These two registers are set by NeSC when a translation of a write request sent to the VF misses in the extent tree. After setting these values, NeSC sends an interrupt to the hypervisor so it will allocate additional physical space to the file associated with the VF.

4. **RewalkTree** (4B) This register is used by the hypervisor to signal NeSC that the physical storage allocation has succeeded. When the hypervisor writes a value of 1 to a VF's RewalkTree register, NeSC reissues the stalled write requests to the extent tree walk unit.

In addition to the NeSC-specific control registers listed above, each VF also exposes a set of registers for controlling a DMA ring buffer [Lov10], which is the de facto standard for communicating with devices. We omit those for brevity.

6.2 Kernel Drivers

As mentioned in 2.2, a device driver communicates with the NeSC device through the PCIe interconnect. When the block layer decides to commit read/write requests to the underlying storage, it sends the requests to the driver in the form of bio structures. The driver will translate the request to NeSC read/write requests and return when the request is full filed.

NeSC is controlled by two device drivers, one is used by the hypervisor to access the PF, and one is used by virtual machines to access the VF.

The VF Driver

A VM that wants to access the NeSC device, installs the NeSC VF device driver. When a request is sent to the driver by the block layer, the driver will map the request to a

scatter-gather list and send the nodes to NeSC one by one. The nodes in the list are DMA requests for NeSC to execute. Each node in the list consists of:

- *dma address* The bus address of the current request. NeSC will DMA to/from this address.
- *offset* The offset from the beginining of the bus address.
- *length* The amount of bytes to transfer. Each node is limited to represent up to 4K bytes which is exactly the size of a page on the system.

while sending the list, NeSC will start executing the requests one by one and when its done, NeSC will send a finished interrupt to the driver.

PF Driver

The PF driver's interface with the block layer is the same as the VF, and through it the hypervisor installs the root filesystem on NeSC. The hypervisor uses the PF driver to manage SR-IOV capabilities. Through the driver the hypervisor creates, deletes and manages VFs. When a new VF is created, the PF driver is responsible of assigning the ExtentTreeRoot register to point to the correct base address of the extent tree.

During NeSC operation, if a VF tries to write to an unassigned block (as seen in 5.3), an interrupt will be sent by NeSC and will be handled in the PF driver. The driver will allocate and assign new blocks to the VF extent tree and signal the VF to continue with the write request.

6.3 The NeSC virtual function multiplexer

In this section we describe the implementation of the NeSC virtual function multiplexer, which is depicted in Figure 6.2. The main role of the virtual function multiplexer is to process access requests from the multiple VFs, translate their addresses to physical storage blocks, and respond to the client VM. Access requests are sent by the VM's block driver. The driver typically breaks large requests into a sequence of smaller 4KB requests that match the system's page size. Requests sent from different client VMs are stored in per-client request queues. NeSC dequeues client requests in a round-robin manner in order to prevent client starvation (a full study of client scheduling algorithms and different quality-of-service guarantees is beyond the scope of this paper).

Client requests address their target storage blocks using vLBAs, or logical block addresses as viewed by the virtual device. These addresses must be translated to pLBAs using the per-VF extent tree. The client requests are therefore pushed to a shared vLBA queue for translation, along with the root pointer to their VF's extent tree.

NeSC includes a dedicated *translation unit* (discussed below) that dequeues pending requests from the vLBA queue, translates their vLBAs to pLBAs and pushes the

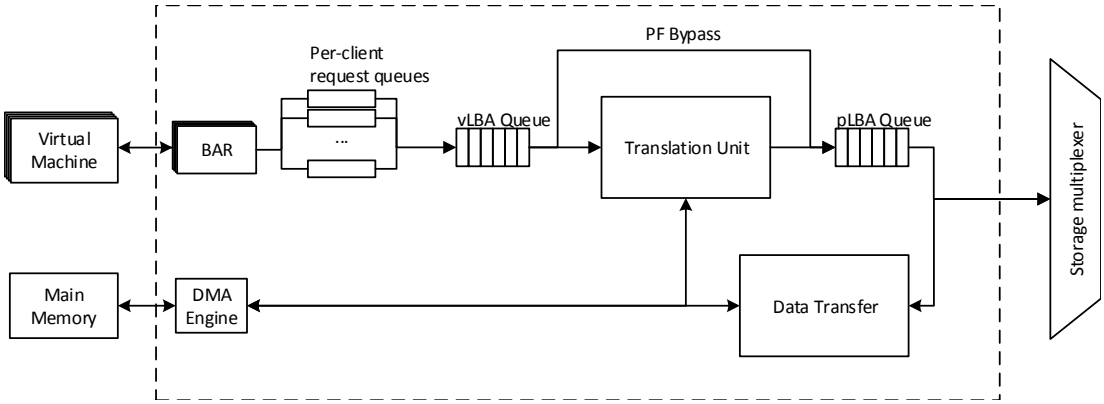


Figure 6.2: High-level view of the NeSC virtual function multiplexer design. The microarchitecture multiplexes requests from the different virtual functions.

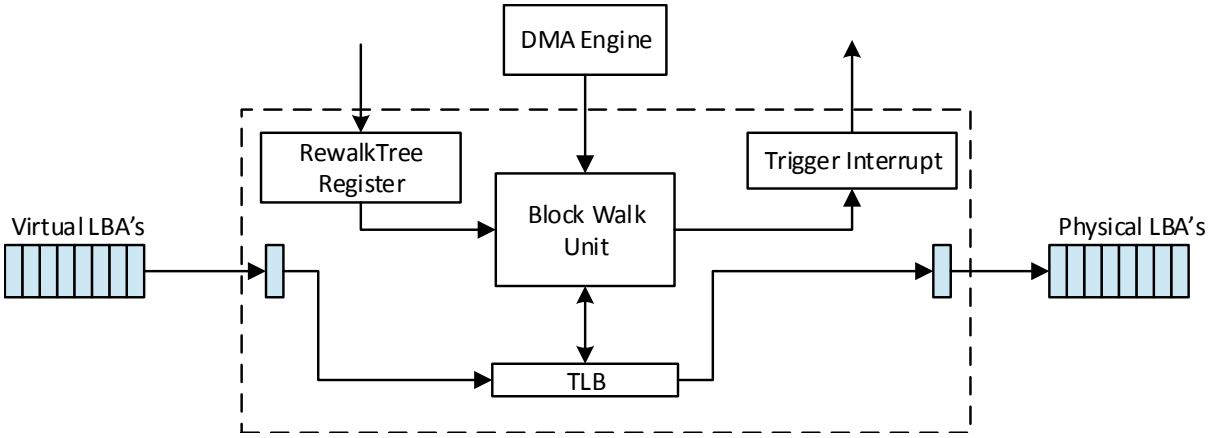


Figure 6.3: The vLBA-to-pLBA translation unit.

translated requests to a pLBA queue. A *data transfer* unit then manages direct accesses to the persistent storage device. For write requests, the data transfer unit simply writes the data to storage and generates an acknowledge response message that will be sent to the client. For read requests, the data transfer unit reads the target data from the physical storage and prepares a response message. The message is then DMAed to the client VM's destination buffer in host memory.

While we only discuss the processing of client VM requests, NeSC also includes a dedicated out-of-band (OOB) channel to process hypervisor requests sent through the PF. The OOB channel is needed so that VF write requests whose translation is blocked will not block PF requests. The addition of the OOB is, however, simple since PF requests use pLBAs and need not be translated. The OOB channel, therefore, bypasses the NeSC flow preceding the pLBA queue and does not affect the the address translation unit.

The vLBA-to-pLBA translation unit

The translation unit translates the vLBA addresses used in client VM requests to pLBA addresses, which can be used to access the physical storage. The translation uses the extent tree associated with a request's originating VF.

Figure 6.3 illustrates the translation unit and its components. These include a *block walk unit* that traverses the extent tree and a *block translation lookaside buffer* (BTLB) that caches recent vLBA mappings. Furthermore, the unit may send interrupts to the hypervisor when a mapping is not found (e.g., when a client VM writes to an unallocated portion of its virtual device), and it is charged with observing the VF's *RewalkTree* registers through which the hypervisor signals that the mapping was fixed and the extent tree can be re-examined.

Block walk unit

This unit executes the block walk procedure by traversing the extent tree. When a client VM request arrives at the unit, the root node of the associated extent tree is DMAed and the unit tries to match the vLBA to the offsets listed in the root node's entries (be they node pointers or extent pointers). The matched entry is used as the next node in the traversal, and the process continues recursively until an extent is matched. For read operations, an unmatched vLBA means that the client VM is trying to read from an unmapped file offset (a hole in the file) and zeros must be returned. If a match is not found on a write operation, the unit sets the VF's *MissAddress* and *MissSize* control registers, interrupts the host, and waits until the hypervisor signals that the missing blocks were allocated (using the *RewalkTree* control register).

The block walk unit is designed for throughput. Since the main performance bottleneck of the unit is the DMA transaction of the next level in the tree, the unit can overlap two translation processes to (almost) hide the DMA latency.

Block translation lookaside buffer (BTLB)

Given that storage access exhibits spatial locality, and extents typically span more than one block, the translation unit maintains a small cache of the last 8 extents used in translation. Specifically, this enables the BTLB to maintain at least the last mapping for each of the last 8 VFs it serviced.

Before the translation unit begins the block walk, it checks the BTLB to see whether the currently translated vLBA matches one of the extents in the BTLB. If a match is found, the unit skips the block walk and creates a new pLBA for the output queue. On a miss, the vLBA will be passes to the block walk unit and, after the translation completes, the mapping will be inserted to the BTLB (evicting the oldest entry).

Finally, the BTLB cache must not prevent the hypervisor from executing traditional storage optimizations (e.g., block deduplication). Consequently, NeSC enables the PF

(representing the hypervisor) to flush the BTLB cache in order to preserve meta-data consistency.

In summary, the NeSC implementation effectively balances the need to multiplex different VFs with the application of traditional optimizations such as caching and latency hiding. The following sections describe our evaluation of the proposed NeSC design.

Chapter 7

Methodology

Host machine	
Machine model	Supermicro X9DRG-QF
Processor	Dual socket Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz (Sandybridge)
Memory	64GB DDR3 1333 MHz
Operating system	Ubuntu 12.04.5 LTS (kernel 3.5.0)
Virtualized system	
Virtual machine monitor	QEMU version 2.2.0 with KVM
Guest OS	Linux 3.13
Guest RAM	128MB
Filesystem on NeSC volume	ext4
Prototyping platform	
Model	Xilinx VC707 Evaluation board
FPGA	Virtex-7 (XC7VX485T-2FFG1761C)
RAM	1GB DDR3 800MHz
Host I/O	PCI Express x8 gen2

Table 7.1: Experimental platform

Microbenchmark	
GNU dd [cor]	read/write files using different operational parameters.
Macrobenchmarks	
Sysbench File I/O [Kop04]	A sequence of random file operations
Postmark [Kat97]	Mail server simulation
MySQL [mys]	Relational database server serving the SysBench OLTP workload

Table 7.2: Benchmarks

Experimental system

Table 7.1 describes our experimental system, which consists of a Supermicro X9DRG-QF server equipped with a Xilinx VC707 evaluation board.

Emulating IOV

We implemented NeSC using a VC707's Virtex-7 FPGA. Since this revision of the Virtex7 FPGA only supports PCIe Gen2, we had to emulate the self-virtualizing features rather than use the SR-IOV protocol extension. We have emulated the SR-IOV functionalities by dividing the device's memory-mapped BAR to 4KB pages. The first page exports the NeSC PF, and subsequent pages export complete VF interfaces. Upon QEMU startup, the hypervisor maps one of the VF interfaces (offset in the BAR) into the physical address space of the VM. A multiplexer in the device examines the address to which each PCIe packet (TLP) was sent and queues it to the appropriate VF's command queue. For example, a read TLP that was sent to address 4,244 in the NeSC device would have been routed by the multiplexer to offset 128 in the first VF.

Furthermore, since the emulated VFs are not recognized by the IOMMU, VMs cannot DMA data directly to the VC707 board. Instead, the hypervisor allocates trampoline buffers for each VM, and VMs have to copy data to/from the trampoline buffers before/after initiating a DMA operation.

We note that both SR-IOV emulation and trampoline buffers actually impede the performance of NeSC and thereby provide a pessimistic device model. Using a true PCIe gen3 device would improve NeSC's performance.

We further note that we do not emulate a specific access latency technology for the emulated storage device. Instead, we simply use direct DRAM read and write latencies.

We used the QEMU/KVM virtual machine monitor. Since the VC707 board only has 1GB of RAM, we could only emulate 1GB storage on the NeSC device. In order to prevent the entire simulated storage device from being cached in RAM, we limited the VM's RAM to 128MB. We validated that this limitation does not induce swapping in any of the benchmarks.

Finally, in order for NeSC to truly function as a self-virtualizing device, we implemented the VF guest driver, which is a simple block device driver, and the hypervisor PF driver, which acts as both a block device driver and as the NeSC management driver for creating and deleting VFs.

Benchmarks

Table 7.2 lists the benchmarks used to evaluate NeSC. We first evaluated read/write performance metrics (e.g., bandwidth, latency) using the *dd* Unix utility. In addition, we used a common set of macrobenchmarks that stress the storage system. All applications used the virtual device through an underlying ext4 filesystem.

Chapter 8

Evaluation

This section reports the evaluation results of the NeSC prototype. We examine the performance benefits of enabling VMs to directly access NeSC, and compare its performance to those of virtio and device emulation. Our baseline storage device is the NeSC PF, which presents the hypervisor a raw storage device with no file mapping capabilities. The baseline measurement is done by the hypervisor without any virtualization layer.

Raw device performance

We begin by examining the raw device performance observed by the guest VM when accessing a virtual NeSC device. A file on the hypervisor's filesystem is used to create a VF, which is then mapped to the guest VM. These results are compared to mapping the PF itself to the guest VM using either virtio and device emulation. The baseline (marked *Host* in the figures) is the performance observed when the hypervisor directly accesses the PF block device (without virtualization). In all configurations, we examine the performance of reads and writes to the raw virtual device, without creating a filesystem.

When a guest accesses the device using virtio or device emulation, each access request is processed by the guest I/O stack and storage device driver, delivered to the hypervisor, and is then processed by the hypervisor's I/O stack and its own device driver. When directly assigning a NeSC VF to the guest, requests pass down the guest I/O stack directly to the device. In contrast, in the baseline evaluation (i.e., *Host* in the figure) the requests only pass down the hypervisor I/O stack.

The performance itself was measured using *dd* [cor] for different block sizes.

NeSC latency

Figure 8.1 shows the latency observed for read (top) and write (bottom) operations, using request sizes varying from 512B to 32KB. The figure shows that the latency obtained by NeSC for both read and write is similar to that obtained by the host when directly accessing the PF. Furthermore, the NeSC latency is over 6× faster than virtio

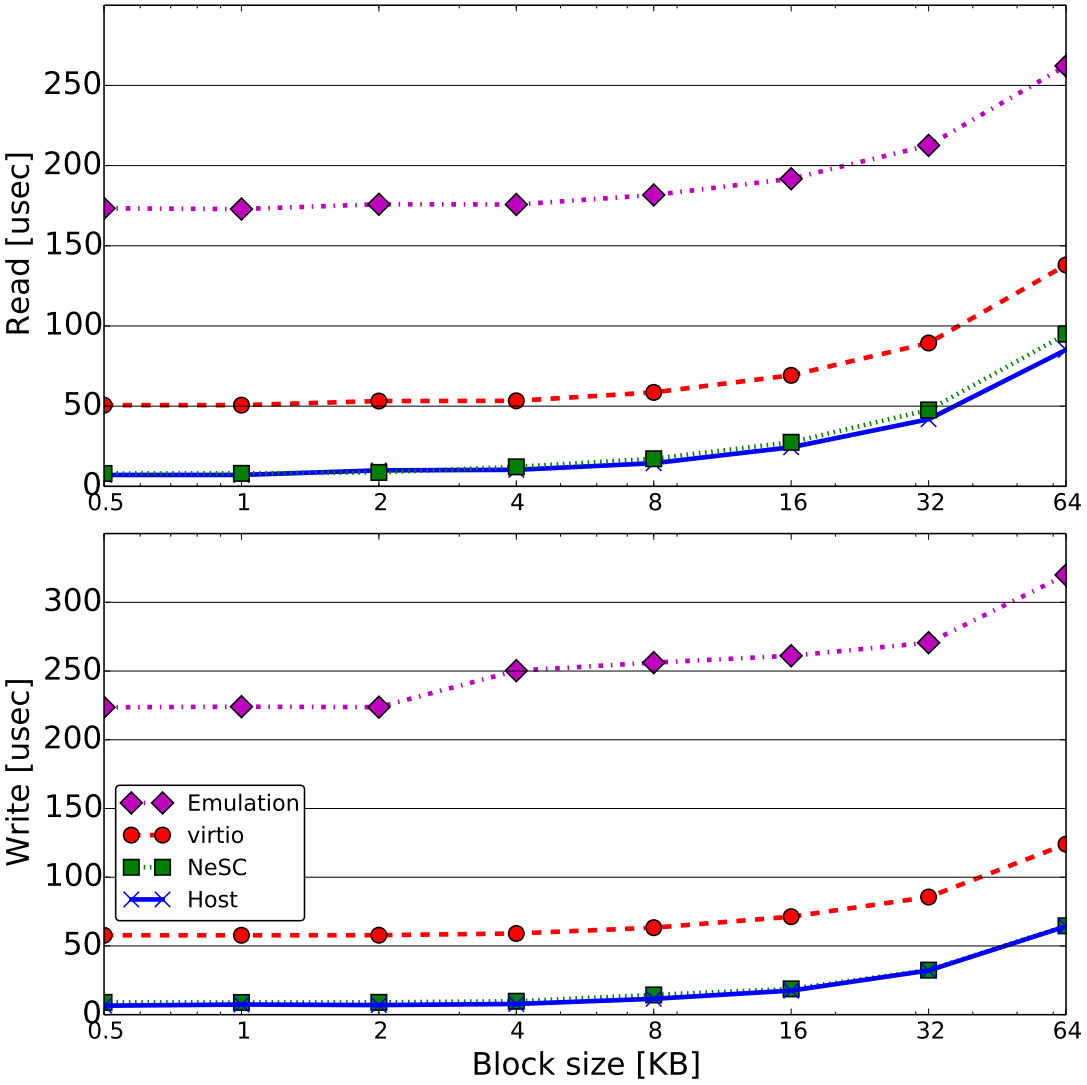


Figure 8.1: Raw access latency observed for read (top) and write (bottom) operations for different block sizes.

and over 20× faster than device emulation for accesses smaller than 4KB.

NeSC bandwidth

Figure 8.2 shows the bandwidth delivered for read (top) and write (bottom) operations, using request sizes varying from 512B to 32KB.

We begin by examining the read bandwidth. The figure shows that for reads smaller than 16KB, NeSC obtained bandwidth close to that of the baseline and outperforms virtio by over 2.5×. Furthermore, for very large block sizes (over 2MB), the bandwidths delivered by NeSC and virtio converge. This is because such large accesses ameliorate the overheads incurred by VM/hypervisor traps (vmenter/vmexit on Intel platforms).

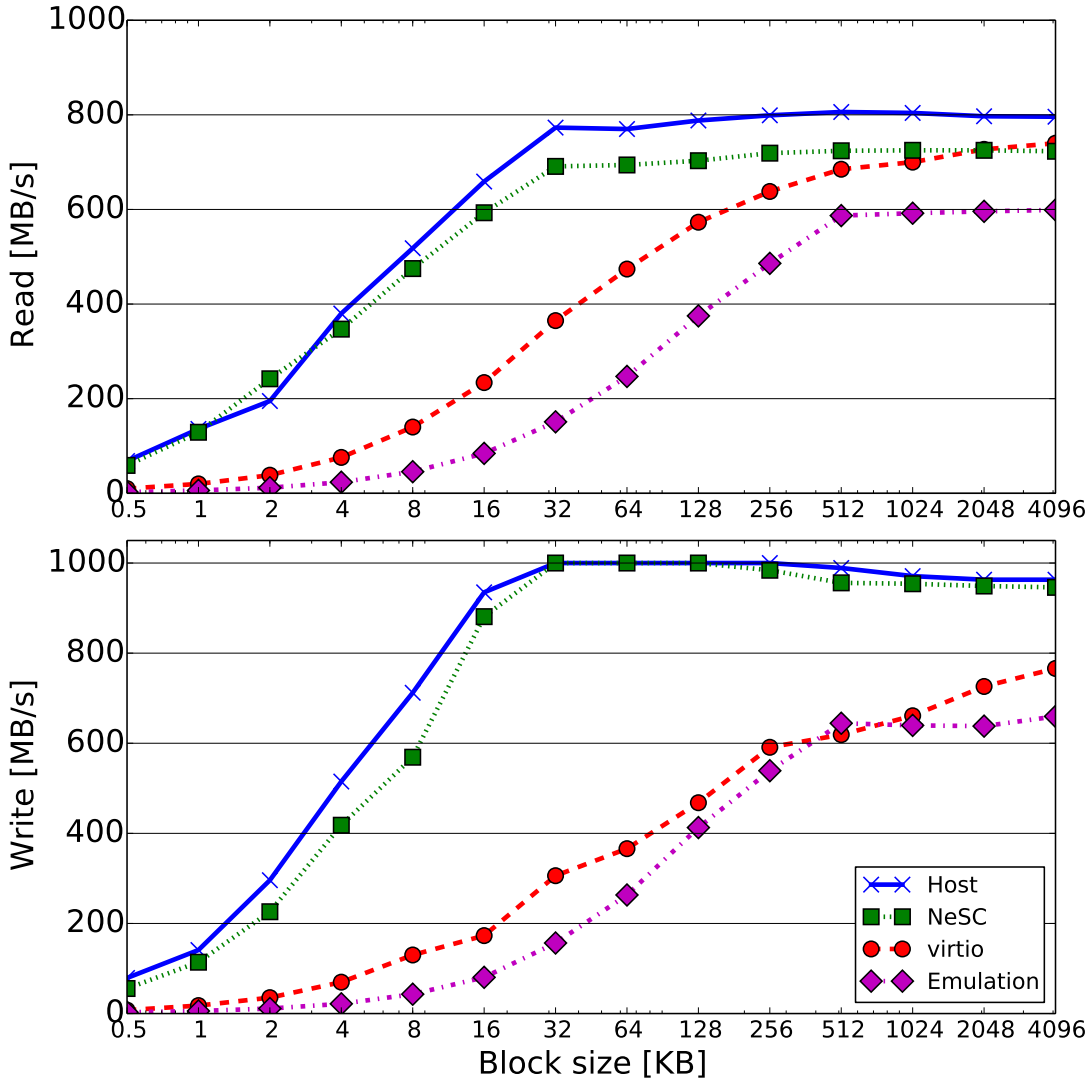


Figure 8.2: Raw bandwidth observed for read (top) and write (bottom) operations for different block sizes.

When examining the write bandwidth, we observe that NeSC delivers performance similar to the baseline for all block sizes. This result is better than that achieved for read bandwidth, for which NeSC is ~10% slower than the baseline for blocks of size 32KB and larger. Moreover, NeSC's write bandwidth is consistently and substantially better than virtio and emulation, peaking at over 3× for 32KB block sizes.

Filesystem overheads

We next examine the overheads incurred by filesystem translations. To this end, we compare the access latency observed by the guest VM when accessing the raw device with that observed when accessing an ext4 filesystem on the virtual device.

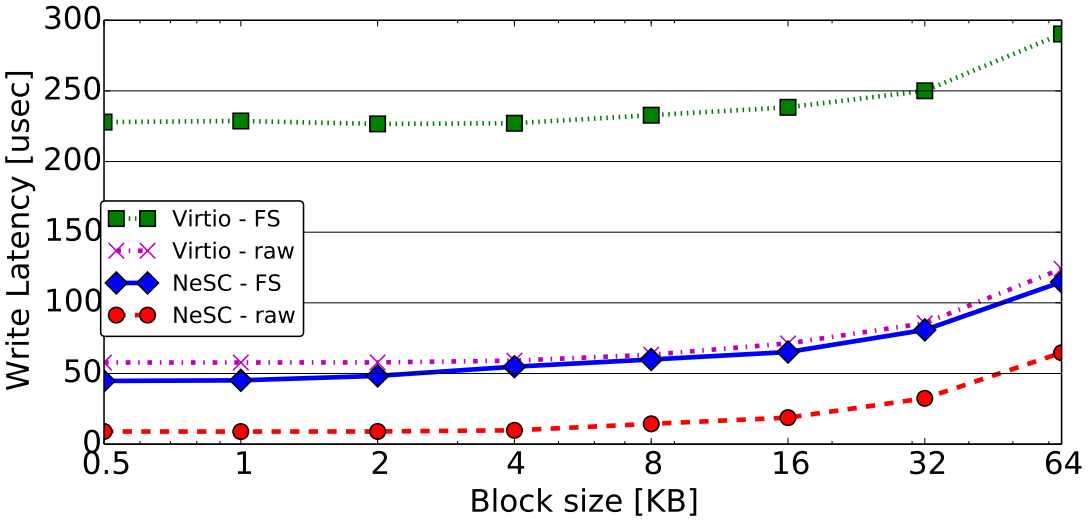


Figure 8.3: Filesystem overheads.

Figure 8.3 shows the write latency when accessing the device with and without a filesystem, for NeSC and virtio virtualization (for brevity, we omit the results obtained using an emulated device since they were much worse than virtio). We only show results for writes because those are more prohibitive for NeSC, since writes may require the VF to request extent allocations from the OS's filesystem.

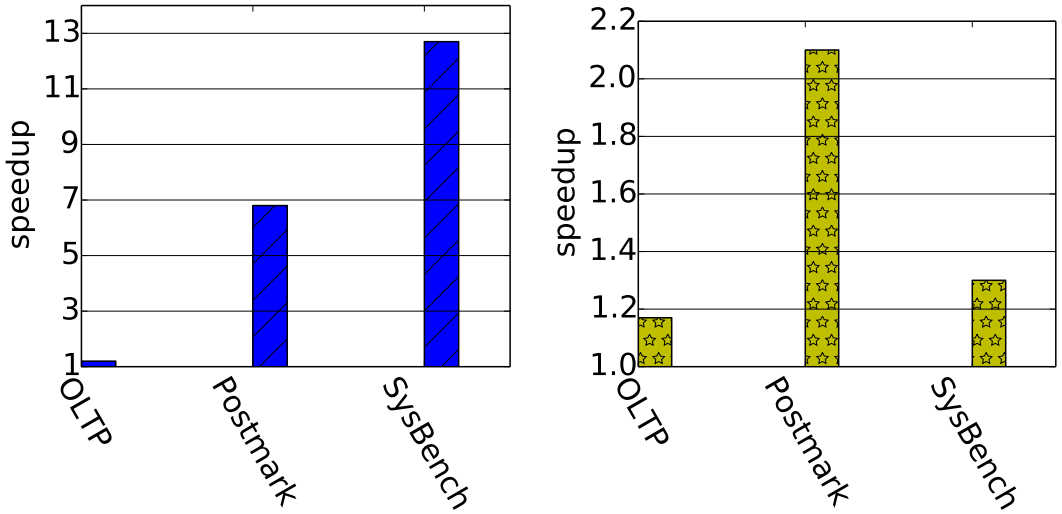
The figure demonstrates the performance benefits of NeSC. While the filesystem overhead consistently increases NeSC's write latency by $\sim 40\mu s$, the latency obtained using NeSC is similar to that of a raw virtio device. Using virtio with a filesystem incurs an extra $\sim 170\mu s$, which is over $4\times$ slower than NeSC with a filesystem for writes smaller than 8KB.

We note that the similar latencies observed when using a filesystem with NeSC and when using a raw virtio device indicate that NeSC effectively eliminates the hypervisor's filesystem overheads.

Application performance

We finally examine how the raw performance delivered by NeSC translates to application level speedups. To this end, we compare the performance of the applications (listed in Figure 7.2) when running in a guest Linux VM whose storage device is mapped to a NeSC VF, to a virtio device and to a fully emulated device.

The virtual storage device is mapped to the VM in the following manner. The hypervisor creates an ext4 filesystem on the raw device using the NeSC PF. The virtual storage device is stored as an image file (with ext4 filesystem) on the hypervisor's filesystem, and the hypervisor maps the file to the VM using either of the mapping facilities: virtio, emulation or a NeSC VF.



(a) Applications speedups when using NeSC over device emulation.

(b) Application speedups when using NeSC over virtio.

Figure 8.4: Application speedups over other storage virtualization methods.

Figure 8.4 shows the application speedups obtained using a NeSC VF device over device emulation and virtio. For the MySQL OLTP benchmark, the figure shows that mapping the virtual disk using a NeSC virtual device improves performance by 1.2× over both device emulation (Figure 8.4a) and virtio (Figure 8.4b).

The performance improvements provided by NeSC are even more substantial for Postmark and Sysbench File I/O. Postmark enjoys more than 6× speedup over device emulation and more than 2× over virtio. Sysbench File I/O, on the other hand, enjoys a dramatic 13× speedup over device emulation, and 1.3× over virtio.

In summary, the evaluation of the NeSC prototype demonstrates its substantial performance benefits over state-of-the-art storage virtualization methods. The benefits are consistent for both read and write microbenchmarks and for common storage benchmarks.

Chapter 9

Conclusion and open questions

The emergence of multi-GB/s storage devices has shifted storage virtualization bottlenecks from the storage devices to the software layers. System designers must therefore delegate the virtualization overheads in order to enable virtualized environments to benefit from high-speed storage.

In this paper we presented the *nested, self-virtualizing storage controller* (NeSC), which enables files stored on the hypervisor-managed filesystem to be directly mapped to guest VMs. NeSC delegates filesystem functionality to the storage device by incorporating mapping facilities that translate file offsets to disk blocks. This enables NeSC to leverage the self-virtualization SR-IOV protocol and thereby expose files as virtual devices on the PCIe interconnect, which can be mapped to guest VMs.

We prototyped NeSC using a Virtex-7 FPGA and evaluated its performance benefits on a real system. Our comparison of NeSC to the leading *virtio* storage virtualization method, shows that NeSC effectively eliminates the hypervisor's filesystem overheads, as filesystem accesses to a NeSC virtual device incur the same latency as accesses to a raw virtio device. Furthermore, we have shown that NeSC speeds up common storage benchmarks by 1.2×–2×.