Paravirtual Remote I/O

Yossi Kuperman
Technion – Israel institute of Technology & IBM Research—Haifa
skyossi@cs.technion.ac.il

Eyal Moscovici
Technion – Israel institute of Technology & IBM Research—Haifa
emos@cs.technion.ac.il

Joel Nider
IBM Research—Haifa
joeln@il.ibm.com

Razya Ladelsky
IBM Research—Haifa
razya@il.ibm.com

Abel Gordon
IBM Research—Haifa & Stratoscale
abel.gordon@gmail.com

Dan Tsafrir
Technion – Israel institute of Technology
dan@cs.technion.ac.il

Abstract

The traditional "trap and emulate" I/O paravirtualization model conveniently allows for I/O interposition, yet it inherently incurs costly guest-host context switches. The newer "sidecore" model eliminates this overhead by dedicating host (side)cores to poll the relevant guest memory regions and react accordingly without context switching. But the dedication of sidecores on each host might be wasteful when I/O activity is low, or it might not provide enough computational power when I/O activity is high. We propose to alleviate this problem at rack scale by consolidating the dedicated sidecores spread across several hosts onto one server. The hypervisor is then effectively split into two parts: the local hypervisor that hosts the VMs, and the remote hypervisor that processes their paravirtual I/O. We call this model vRIO—paraVirtual Remote I/O. We find that by increasing the latency somewhat, it provides comparable throughput with fewer sidecores and superior throughput with the same number of sidecores as compared to the state of the art. vRIO additionally constitutes a new, cost-effective way to consolidate I/O devices (on the remote hypervisor) while supporting efficient programmable I/O interposition.

1. Introduction

Interposition I/O virtualization decouples logical from physical I/O through an indirection layer. The host exposes a virtual I/O device to its guest virtual machines (VMs). It then intercepts ("traps") VM requests directed at the virtual device, and it fulfills ("emulates") them using the physical hardware. This trap-and-emulate approach [28,54] allows the host to *interpose* on the I/O activity of its guests.

The benefits of I/O interposition are substantial [69]: improving utilization of the physical devices via time and space multiplexing; enabling live migration and suspend/resume using the indirection layer to decouple and recouple VMs from/to the physical device; enabling seamless switching between different I/O channels, which allows for device aggregation, load balancing, and failure masking; and supporting such features as monitoring, software-defined networking (SDN), file-based images, replication, deduplication, snapshots, and security related functionalities such as record-replay, encryption, firewalls, and intrusion detection.

SRIOV The virtual I/O indirection layer hinders performance, mainly due to the overhead generated by *exits* [1], namely, the guest-host context switches upon I/O operations, which are being trapped and emulated. This overhead has motivated hardware vendors to develop the Single Root I/O Virtualization (SRIOV) technology [34,52], whereby a physical I/O device can self-virtualize, supporting multiple virtual instances of itself that can be individually assigned to VMs. SRIOV thus bypasses the host, which is largely removed from the critical data path. The result is significantly better performance for I/O intensive workloads [41,44,46,56,72,76]. There is a serious drawback, however, to using SRIOV, as it eliminates the indirection layer and thus only provides multiplexing out of the benefits of virtual I/O listed above. Notably, SRIOV negates live migration, memory overcommitment, metering, and other such features that rely on interposition and that are crucial for VM environments.

Paravirtualization Consequently, real-world applications of virtualization today, including most enterprise data centers and cloud computing sites, seldom use SRIOV. Instead, they realize virtual I/O through *paravirtualization* [11,58,66,70],

ASPLOS '16, April 2–6, 2016, Atlanta, GA, USA..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4091-5/16/04…$15.00.
http://dx.doi.org/10.1145/2872362.2872378

exemplified by VMware VMXNET3 [67], KVM virtio [58], and Xen PV [11]. Paravirtual I/O boosts the performance of the virtualization indirection layer while retaining all the aforementioned benefits of interposable virtual I/O. Paravirtualization achieves this goal by requiring guests to install a host-specific device driver that is purely software based. The latter is not modeled after any real device. Rather, it is optimized to minimize the overheads associated with guest-host communication and context switching.

Sidecores While offering an improvement, paravirtualization still hampers performance due to the exits it induces [13, 22, 40, 44, 57, 74]. Recent studies show that the *sidecore* approach is effective in combating this problem [4, 13, 31, 39, 43, 45]. The host's cores are divided to (1) sidecores dedicated to processing virtual I/O, and to (2) *VMcores* for running the VMs that generate the I/O. Each VM writes its I/O requests to memory shared with the host as is usual for the paravirtual I/O model. Unusual, however, is that VM does not trigger an exit. Instead, the host sidecore polls the relevant memory region and processes the request on arrival. The benefit of this approach is twofold: more cycles to the VMs (whose OSes are asynchronous in nature and do not block-wait for I/O) and less cache pollution, yielding a substantial performance improvement. For example, the Elvis system—which implements paravirtual block and net devices on sidecores—is up to 3x more performant than baseline paravirtualization and is oftentimes on par with SRIOV [31].

The sidecore virtual I/O model has an inherent drawback. A sidecore must continuously poll the relevant memory regions of its associated VMs. Therefore, 100% of the sidecore's cycles are consumed, even when the I/O load generated by these VMs is light. In this case, polling wastes valuable CPU cycles that could have otherwise been used to support other, more productive activity. Conversely, it is also possible that the load generated by the corresponding VMs exceeds the processing capabilities of their designated sidecores, and thus the sidecores might become a bottleneck in the system.

Sidecore Consolidation The most notable benefit of virtualization is, arguably, resource consolidation, enabling multiple OSes to multiplex the physical hardware. Experience shows that multiple virtual servers can be adequately serviced by much fewer physical servers. We contend that the same reasoning holds for sidecores, and that applying it will alleviate the problem of having too few or too many per-server sidecores, making this newer I/O model more cost effective.

Departing from the traditional trap-and-emulate virtualization model, the sidecore paradigm already decouples the host's I/O processing from the VM's core, executing it on a different (side)core. In the context of rack scale computing, we propose to take this paradigm a step further and migrate sidecores to a different server. We call this I/O model Para**v**irtual **R**emote **I/O** (vRIO). In vRIO, hosts are either *VMhosts* or *IOhosts*, consisting of either VMcores or sidecores. VMhosts do not process (and are unaware of) the paravirtual I/O of their VMs. Consolidating the sidecores in IOhosts allows us to achieve superior performance using the same number of CPUs, or comparable performance using fewer CPUs. In exchange for reducing the CPU number, vRIO requires additional networking infrastructure in the rack; we show how this tradeoff yields a cheaper system.

Device Consolidation Supporting the cost-effectiveness of the net/CPU tradeoff is the fact that vRIO introduces a new type of device consolidation for VMs. Splitting the hypervisor into local and remote components makes the remote devices at the IOhost efficiently accessible to local VMs via the paravirtual I/O interface. Note that this feature is complementary to existing device consolidation approaches, such as storage area networks (SANs). The reason: if the VM is configured to use a SAN directly (not as a virtual device), then we lose interposition entirely. Otherwise, the SAN is exposed to the VM as a traditional paravirtual device and so the associated overheads of traditional paravirtualization come into play yet again.

Hypervisor Independence The vRIO model bypasses the local hypervisor, so the vRIO driver is independent of the local hypervisor, thus providing several additional advantages. It is possible, for example, to implement a layer-2 virtual LAN that seamlessly works for virtualized OSes across different processor architectures and hypervisor types. This feature is valuable, as there are nowadays fewer (guest) OSes in widespread use than hypervisors. vRIO also supports bare metal (non-virtual) OSes that install its driver, e.g., they too can be easily incorporated in the aforementioned LAN.

Notably, system-wide services and policies can be implemented for all hypervisor types in one location. Moreover, the IOhost hardware can be specialized for I/O processing.

Minimizing Latency To enjoy the benefits of vRIO, we must make the penalty of adding one hop to the I/O path tolerable. To this end, somewhat surprisingly, we utilize SRIOV. Whereas SRIOV negates interposition in existing I/O models, vRIO is able to use it in a compatible manner. Specifically, the vRIO paravirtual driver installed in the VM generates the same I/O as in traditional paravirtualization. But instead of handing the I/O to the local hypervisor via shared memory, the vRIO driver communicates it to the IOhost through an SRIOV channel. Additionally, vRIO entirely eliminates the overhead of interrupt processing from the virtualization layer by: (1) coupling the SRIOV channel with exitless interrupts [5, 29] at the VMhost; and by (2) polling the Network Interface Controllers (NICs) at the IOhost in accordance to the sidecore paradigm.

Preview of Results vRIO trades off CPUs for NICs and latency for improved resource utilization. We describe supposed vRIO setups that are 8–38% cheaper than the alternative, and we manage to bound vRIO's latency to be at most 1.18x longer than state-of-the-art sidecore paravirtualization for network I/O. The latency is up to 2.20x longer if, in addition to sidecores, we also migrate local block devices to reside in the remote IOhost. The benefit is achieving, e.g., 1.82x the throughput using the same number of sidecores, or 0.92x the throughput using half the number of sidecores.

2. Existing Virtual I/O Models

The vRIO model builds on and extends existing I/O virtualization models. Next, we briefly survey them, highlighting some additional aspects beyond those already mentioned above.

The most naive implementation of virtual I/O devices is *emulation*, whereby the hypervisor implements an interface identical to that of an existing, widely supported hardware device. (For NICs, this is Intel's e1000.) The hypervisor delivers the required functionality by trapping and emulating [28, 54]. Emulation is convenient, because it is hypervisor-agnostic and leaves VMs unaware of being virtualized.

Baseline The cost of emulation is degraded performance, as the physical device was not designed with virtualization in mind, inducing costly exits upon each interaction between the VM and its emulated device. Consequently, installations overwhelmingly prefer *paravirtualization*, which is similar to emulation except that the emulated device does not correspond to any existing hardware but instead is designed to minimize VM-device interactions. Despite being similar in essence across different hypervisors, paravirtual device drivers are hypervisor-specific, because vendors opted to use different interfaces. Thus, VMs must install the appropriate paravirtual device drivers to reap the benefit. In the paper, we use KVM virtio [58] as the state-of-practice representative of paravirtualization. We denote it as the *baseline* configuration.

Optimum Although more performant than naive emulation, paravirtualization still traps upon every I/O request and thus fails to deliver bare metal (non-virtual) performance [41, 46, 56, 72, 76]. The PCI-SIG has thus standardized the SRIOV extension of PCIe [34, 52], which allows physical I/O devices to *self-virtualize* and support multiple virtual instances of themselves such that each instance can be assigned to a different VM. SRIOV outperforms paravirtualization, as I/O requests generated by guests do not trap. But it does not provide bare metal performance, because device interrupts are still handled by the host [5, 14, 22, 29, 31, 40, 44, 72]. The *exitless interrupts* approach (denoted *Eli*) overcomes this barrier, devising a virtual I/O model in which guests directly receive the interrupts of their SRIOV instances without host involvement [5, 29]. SRIOV+Eli provides bare metal performance for VMs. We thus denote it as the *optimum* configuration.

Despite the hype, using SRIOV is problematic, because it negates many of the key benefits of virtualization that rest on interposition. For example, it conflicts with memory overcommitment and host swapping [6], as DMAs do not tolerate page faults, forcing the host to either pin the entire image of the VM to memory [12, 53], or to become aware of all its DMA target buffers via explicit guest-host interaction [4, 15, 41, 62, 71]. SRIOV likewise conflicts with live migration [38, 51, 64, 77], because the source host cannot decouple the device while it is working, and because the target host might not have this particular device. Other problems include the host not being able to meter, account for, suspend, resume, log, isolate, and manipulate the I/O activity [30]. All the features that interposable I/O provides are missing.

Elvis The *sidecore* approach is an interposable I/O model that provides all the benefits of—but is more performant than—traditional paravirtualization. The approach dedicates cores to poll relevant memory regions, observe changes that reflect I/O requests, and process these requests without inducing the overhead of trapping. The sidecore paradigm has been applied to various tasks such as IOMMU virtualization [4], storage virtualization [13], TCP/IP processing [33], and GPU processing [63]. The *Elvis* system applies this paradigm to our baseline KVM virtio setup [31]. It utilizes Eli to deliver exitless interrupts (in the form of IPIs) from host (side)cores to guest cores. Elvis was shown to be superior to the baseline, oftentimes approaching the optimum. The Elvis source code is publicly available [23]; we use it as a representative of the state-of-the-art of interposable virtual I/O.

The number of sidecores utilized by Elvis is static, so their computational power might be too low or to high, thereby wasting valuable cycles or hindering performance. Conceivably, we could dynamically (de)allocate sidecores in response to the changing load [49]. But this approach is limited for two reasons. First, because sidecores are discrete—it is impossible to allocate a fraction of a sidecore even if only a fraction is required. Namely, if only p of the sidecore's cycles are needed, then $1 - p$ are wasted. The second, more significant limitation of dynamic sidecore allocation is that it is irrelevant when the aggregated need for VM and I/O processing exceeds the capacity of the individual physical server, for example, because of load imbalance. In such cases, we might consider to resort to live migration so as to hand some of the work to another server. But live migration is a costly, coarse-grained activity that noticeably disrupts service, with reduced response times and downtimes that might reach seconds to tens of seconds [7, 17, 36, 47, 59, 68, 73]; this limitation may explain why Amazon's AWS does not support live migration as some anecdotal evidence suggests [55]. vRIO, in contrast, is better suited to accommodate workload imbalance. (In addition to its other benefits, like I/O device consolidation with efficient programmable interposition.)

3. Cost Effectiveness of vRIO

Using vRIO involves hardware tradeoffs; in this section, we assess the associated cost.

Price Trends The vRIO infrastructure allows the number of processors and other devices local to the server to be reduced (by consolidating them in the IOhost), at the cost of increasing

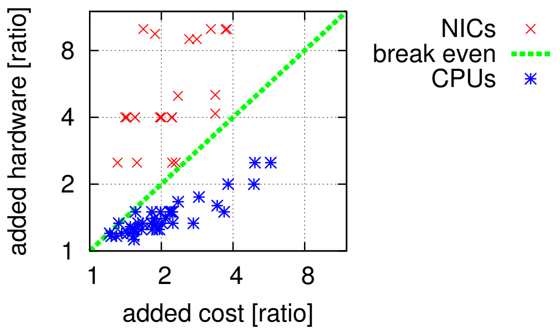

Figure 1. *Within the server, CPU upgrades are costlier than NIC upgrades. CPU data is based on Intel's pricing list [35]. NIC data is based on multiple web sources.*

the number of NICs in that server. Figure 1 shows current trends in NIC and CPU costs that suggest that this tradeoff may be advantageous. The CPU data points in the figure are computed based on the latest pricing list of Intel processors as of this writing [35]. Each data point corresponds to an "adjacent" pair of Xeon CPUs, such that CPUs c_1 and c_2 are adjacent if: (1) the number of cores of c_1 is smaller than that of c_2; (2) the series, version, speed (GHz), and feature size (nm) of the two are identical; and (3) the cache size (MB), power (W), and QPI speed (GT/s) of c_1 are proportionally smaller than or equal to that of c_2. For example, the following two CPUs are adjacent:

c_1: $3,059 **12**-core 2.3GHz E7-8850 v2 24MB 105W 7.2GT/s QPI 22nm
c_2: $4,616 **15**-core 2.3GHz E7-8870 v2 30MB 130W 8.0GT/s QPI 22nm

The (x,y) data points in Figure 1 reflect the relative cost of upgrading from c_1 to c_2 and the resulting relative number of added cores: $x = \frac{\$4,616}{\$3,059} \approx 1.51$ and $y = \frac{15\text{cores}}{12\text{cores}} = 1.25$.

The NIC data points in Figure 1 are equivalently computed based on "adjacent" NIC pairs from Chelsio, Dell, Emulex, HotLava, Intel, Mellanox, and SolarFlare. NICs n_1 and n_2 are adjacent if: (1) the throughout of n_1 is smaller than that of n_2; (2) their vendor, product series, port number, form factor, connector type, and offload capabilities are the same; and (3) the power and PCIe generation and lanes of n_1 are proportionally smaller than or equal to that of n_2. For example, the following two Mellanox NICs are adjacent:

n_1: $560 2x**10**GbE ConnectX-3 MCX312B-XCCT SFP+ PCIe3 x8
n_2: $1121 2x**40**GbE ConnectX-3 MCX314A-BCCT QSFP PCIe3 x8

The corresponding (x,y) data points in Figure 1 likewise reflect the relative upgrade cost and the consequent added bandwidth. In our example, $x = \frac{\$1121}{\$560} \approx 2$ and $y = \frac{80\text{Gbps}}{20\text{Gbps}} = 4$.

Clearly, all CPU data points in Figure 1 are below the main diagonal, whereas NIC data points are above, indicating that augmenting a server's compute hardware involves a premium absent from augmenting its NICs. A similar observation was made regarding the rapidly declining per-port price of network switches and switch modules [16, 75].[1]

One reason underlying the price trends observed in Figure 1 might perhaps be that the server-class processor market is largely dominated by Intel, whereas NICs are manufactured and sold by multiple competing vendors. A second reason could the objective difficulty involved in designing a general purpose computational processor as opposed to a domain-specific one. Regardless of the reason, the observed trends suggest that utilizing CPUs more efficiently (and thus having to purchase less of them) with the help of some additional networking infrastructure could be cost beneficial.

Cost Benefit of vRIO CPU Consolidation We next exemplify the cost tradeoffs made possible with vRIO. Networking infrastructure involves more than just NICs, so the way to check whether vRIO is cost effective is to assess the price of incorporating it in a real rack setup. We use Dell's website, which associates prices with server components, allowing us to configure them online [21] and to compare prices of equivalent Elvis and vRIO systems. We only apply changes to individual servers, possibly rewiring them to an IOhost instead of the switch, as indicated. We constrain the discussion to a single Dell server model for simplicity, although it is likely cheaper to specialize the IOhost. Our comparison applies to racks with 10GbE or 40GbE switches/cabling. We assume 10GbE, as it is more widely deployed [19]. All setups mentioned in this section can be purchased from Dell as specified. We account for the price of *all* the hardware we use, notably networking hardware.

VMhosts can be connected to their IOhost directly or via a switch. Next, we consider the former alternative, which is cheaper, because it allows us to utilize the existing switch and cabling without hampering performance—vRIO supports the same volume of network traffic as its competitors. In §4.6, we consider the drawbacks of this approach and the alternative of connecting VMhosts to IOhosts through a switch.

Consider three Dell PowerEdge R930 Elvis servers, each equipped with two 10Gbps dual-port NICs and four 18-core CPUs, such that 1/3 of the cores serve as sidecores (we demonstrate the use of such a ratio at the end of §5). We calculate the compute-to-network rate based on a study that measured the per-core network throughput granted by 4 cloud providers (Amazon, Google, Rackspace, and Softlayer) and found it to be between 113–380 Mbps when cores concurrently engage in networking [50]. For our R930, this finding translates to at most $4(\text{CPUs}) \times 18(\text{cores}) \times 380\text{Mbps} = 26.72\text{Gbps}$ per server, which is why it is enough to connect to the switch only 3 out of the 4 per-server 10Gbps ports, as shown in Figure 2a.

We contend (and show below) that vRIO allows for efficient sidecore consolidation, thereby providing all the benefits

[1] The observation regarding the declining cost of networking infrastructure is applicable within the rack, as its size is fixed. It may not be applicable to networking infrastructure between racks in growing data centers that house more and more racks [30]. The scope of this work is rack scale.

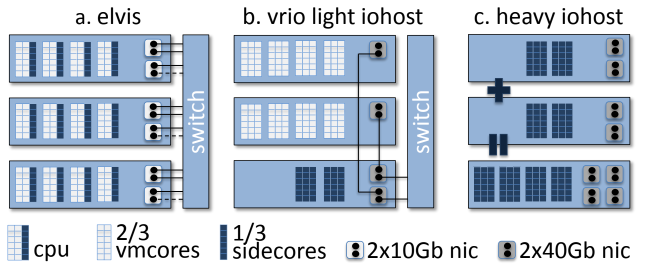

Figure 2. *Elvis sidecores (a) consolidated by vRIO (b), allow an IOhost with only 2 CPUs. Two such "light" IOhosts can be merged into a single "heavy" IOhost (c). If the switch supports 10GbE only, IOhosts are connected to it with 1x40 to 4x10 GbE cables.*

regularly associated with virtualization. Specifically, when assuming that the workload is such that most VMs do not typically utilize *all* their resources *all* the time, we contend that vRIO is capable of delivering comparable performance with, e.g., half the number of sidecores (depending on the workload). Alternatively, vRIO is capable of boosting the performance with the same number of sidecores. In this section, we focus on the former case of halving the number. The corresponding vRIO setup is depicted in Figure 2b; halving allows us to use a "lighter" server as the IOhost with half the CPUs, while leaving the overall number of VMcores identical in both the Elvis and vRIO setups across the three servers.

Assigning the VMs that previously ran on the IOhost to the two VMhosts means that each of them should now handle $1\frac{1}{2}$x the VMs. They have exactly that many additional (VM)cores thanks to the sidecore reconfiguration. But they also need a proportionally higher bandwidth: $26.72\text{Gbps} \times 1\frac{1}{2} = 40.08\text{Gbps}$ per VMhost. The IOhost needs even more: it should handle twice the aggregated throughput of the VMhosts, as every packet transmitted or received by a VMhost must also be (i) received and (ii) retransmitted by the IOhost. The IOhost must therefore handle $2 \times 2(\text{VMhosts}) \times 40.08\text{Gbps} = 160.31\text{Gbps}$. We support these bandwidths by installing one and two 40Gbps dual-port NICs in the VMhosts and IOhost, which provide 80 Gbps and 160 Gbps, respectively.

Notably, the use of 40Gbps NICs between VMhosts and IOhosts is agnostic to whether the switch supports 10GbE or 40GbE. And IOhost ports are connected to the switch using either 40GbE-to-4x10GbE cables (assuming a 10GbE switch) or 40GbE cables (assuming a 40GbE switch); in both cases the number of cables connecting the IOhost to the switch is smaller than the corresponding number in the Elvis setup.

Like bandwidth, the memory of the VMhost is $1\frac{1}{2}$x bigger. We use a rate of 4GB per core, so each Elvis server is equipped with 72 (cores) x 4GB = 288 GB, and each VMhost is equipped with 432 GB. Conversely, the IOhost memory requirements are minimal, because its sole purpose is handling the network activity that flows through it. We equip it with 64GB (minimum for R930), which is more than enough.

component	*price*	*elvis*	*vmhost*	*light iohost*	*heavy iohost*
base	\$6,407	1	1	1	1
18 core CPU	\$8,006	4	4	2	4
8GB DRAM	\$172		2	8	8
16GB DRAM	\$273	18	26		
10Gbps NIC DP	\$560	2			
40Gbps NIC DP	\$1,121		1	2	4
total server price		\$44.5K	\$47.0K	\$26.0K	\$44.2K
total Gbps		40.00	80.00	160.00	320.00
required Gbps		26.72	40.08	160.31	320.63

Table 1. *Dell R930 per-server price, components, and throughput. The CPU is an 18-core 2.5GHz Intel Xeon E7-8890 v3. The NICs pertain to the dual port (DP) Mellanox adapters mentioned earlier; their price includes cable. In the VMhost, we use 2x8GB instead of one 16GB DIMM, as the DIMM number must be even.*

setup	*elvis servers*	*vrio servers*	*elvis price*	*vrio price*	*diff.*
R930 x 3	3	2+1	\$133.4K	\$120.0K	-10%
R930 x 6	6	4+1	\$266.9K	\$232.3K	-13%

Table 2. *Overall price of the Elvis and vRIO setups. The $k + j$ notation specifies the number of VMhosts (k) and IOhosts (j).*

If we take *two* Elvis 3 x R930 setups and transform them to vRIO, we can merge their two "light" IOhosts into a single "heavy" IOhost as depicted in Figure 2c, thereby utilizing five servers instead of six for increased savings. The throughput and prices of individual R930 server types are specified in Table 1. We list the cost and relative difference between the setups we configured in Table 2, which indicates that the vRIO setups are 10% and 13% cheaper.

Cost Benefit of vRIO Device Consolidation A vRIO setup can use the IOhost to further consolidate other devices, in addition to sidecores—SSDs, for example. Such consolidation is *inherently* orthogonal and complementary to other systems that offer storage consolidation, such as SANs, as: (1) if VMs are allowed to use a SAN directly—not through a paravirtual device—then the host forgoes the ability to perform programmable general-purpose interposition; and (2) if the SAN *is* made available to VMs as a traditional paravirtual device, then the associated excessive overheads take effect. vRIO's goal is to alleviate these overheads and allow for performant programmable interposition. Consequently, it makes sense to expose SANs via vRIO in setups that require interposition.

Returning to our example, the R930 can hold up to eight 3.2TB or 6.4TB FusionIO SX300 PCIeSSDs (that cost \$12,706 and \$24,063). If we want to improve the performance of an Elvis setup with such drives, we have to install at least one per server, whereas vRIO allows us to consolidate. As with CPUs, consolidation does not necessarily mean we want to reduce the overall number of drives to get a cheaper setup. We can instead use the same number of drives but make all of them efficiently available to all the servers, thereby in-

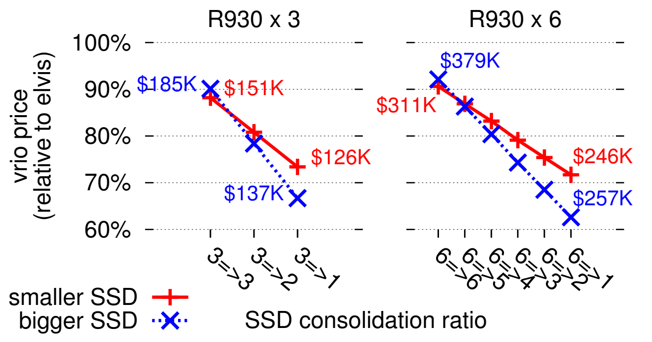

Figure 3. *Price of vRIO relative to Elvis for different drive consolidation ratios (using 3.2TB/6.4TB FusionIO PCIeSSDs).*

creasing the amount of SSD-residing data accessible to each server and boosting overall performance. The SX300 drives deliver up to 2.7GB/sec bandwidth (21.6 Gbps) [60], so consolidating three or six drives requires us to add one or two 2x40Gbps NICs. (The R930 supports up to ten such NICs, so we can add them.)

Let $e \Rightarrow v$ denote the SSD *consolidation ratio*, namely, the number of drives installed in an Elvis setup (e) and in its competing vRIO setup (v). Figure 3 shows the price of the vRIO setups relative to the corresponding Elvis setups, for different consolidation ratios of the aforementioned drives. The cost reduction is between 8%–38%.

4. Design and Implementation

When designing vRIO, our goal is to support sidecore consolidation while trading off as little latency as possible so as to deliver comparable throughput with fewer resources (or better throughput with similar resources) as compared to the state of the art. To this end, we borrow from and extend the three virtual I/O models described in §2.

4.1 Architecture

The vRIO model consists of several components that work together to offload I/O processing from a set of *VMhosts* to their *IOhost*, which houses the VMhosts' sidecores. Usually, VMhosts run guest VMs, but they may also host bare metal OSes as will be discussed later on. We thus use the term *IOclients* to collectively denote the software entities that run on the VMhosts. All I/O processing takes place on the IOhost, on behalf of the IOclients. This transfer of responsibility frees the VMhosts to dedicate more cores to their non-I/O tasks.

vRIO is implemented as a set of drivers in the IOclients and kernel modules in the IOhost. Similarly to the baseline virtio I/O model, vRIO supports both network and block devices. The virtio protocol dictates that guests and host communicate by placing the I/O requests and responses in a shared memory ring buffer (Figure 4a). Elvis follows this protocol, but instead of taking exits on the core of the guest VM, it polls the ring from a different (side)core and allows the VM to continue to compute without interference (Figure 4b). vRIO preserves this architecture but introduces one major change, substituting the shared memory ring with a communications channel (Figure 4c). The latter is exclusively dedicated to encoded I/O traffic that flows between the IOclients and their *I/O hypervisor*—the software that controls the IOhost.

vRIO's dedicated channel is commodity 10Gbps Ethernet, though other interconnects like PCIe [65] or RDMA could be used and would likely provide better performance. Guided by our goal to minimize the latency induced by vRIO, we connect each IOclient to the IOhost through its own SRIOV+Eli instance. Using SRIOV in this manner does not negate interposition, as the I/O hypervisor is free to manipulate the I/O of its clients as it pleases. The only support that vRIO requires from local hypervisors is to assign guests with SRIOV channels. Henceforth, local hypervisors remain uninvolved and unaware of the I/O performed by their guests.

IOclient The IOclient has two driver layers. The first consists of *front-ends*, which expose paravirtual block and net devices to guests, as do virtio and Elvis. The second consists of the *transport* driver, exclusive to vRIO. It is coupled with the aforementioned SRIOV instance and is used to generically support both block and net device types, communicating with the I/O hypervisor as needed. When performing an I/O request, the IOclient hands it to the relevant front-end driver, which in turn hands it to the transport driver. At this point, the request needs to be associated with metadata information, such as the front-end device identifier, type of request, and request size. We directly reuse the virtio protocol—which supplies this information—for this purpose.

The transport driver is responsible for segmenting requests sent to the I/O hypervisor according to the Ethernet protocol. Likewise, it reassembles and decapsulates responses arriving from the I/O hypervisor, before calling the handler functions of the corresponding paravirtual front-end, which notifies the IOclient that a response has arrived.

In virtio and Elvis, the administrator determines the number and type of front-ends assigned to each VM through the local hypervisor. In vRIO, device creation is done via the I/O hypervisor. The transport driver therefore has a secondary role: receiving commands from the I/O hypervisor to create and destroy paravirtual devices in the IOclient.

I/O Hypervisor In the interposable models (virtio, Elvis, vRIO), each front-end device is coupled with an (IO)host back-end, which provides the expected functionality. This emulation layer is where interposition activity takes place, allowing the (IO)host to run services such as block or packet level encryption, SDN, deep packet inspection, intrusion detection, anti-virus, deduplication, and compression.

The vRIO I/O hypervisor consists of a set of *workers*, each running on a different (side)core. Workers service I/O requests from IOclients or from external parties that communicate with them. Requests arrive through IOhost NICs and are directly intercepted by the workers, without going through the TCP/IP stack. A worker that becomes

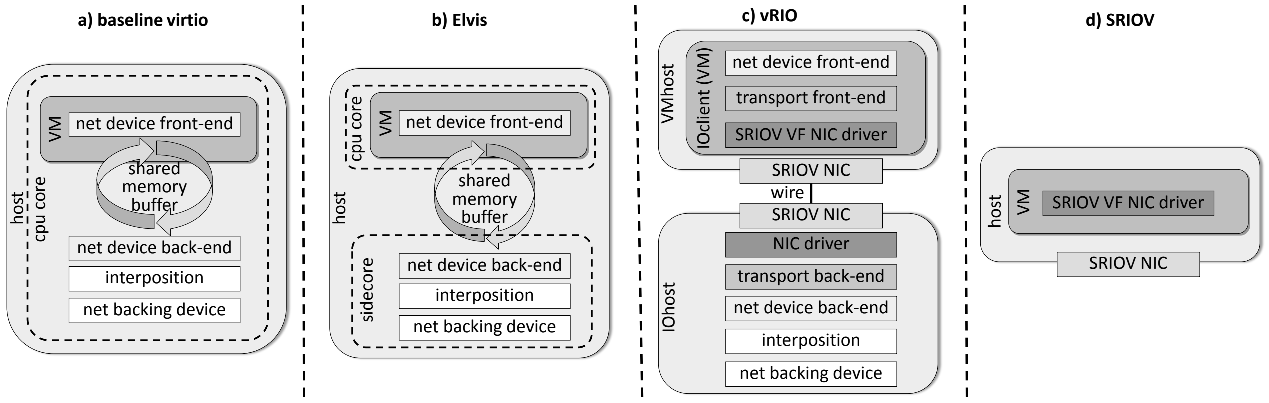

Figure 4. *Side-by-side comparison of the components used for implementing a virtual network device across the four virtual I/O models. The baseline virtio, Elvis, and vRIO are interposable; SRIOV is not. vRIO replaces the shared memory buffers with networking communications. Virtual block devices are not shown but are implemented similarly.*

I/O model	*sync exits*	*guest intrpts*	*intrpt injection*	*host intrpts*	*IOhost intrpts*	*sum*
optimum	0	2	0	0	-	2
vrio	0	2	0	0	0	2
elvis	0	2	0	2	-	4
vrio w/o poll	0	2	0	0	4	6
baseline	3	2	2	2	-	9

Table 3. *Qualitative comparison between the overheads of the different virtual I/O models induced upon a single request-response.*

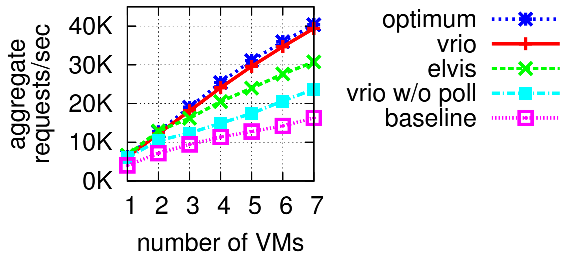

Figure 5. *Turning polling off in vRIO and enabling interrupts substantially degrade the performance of ApacheBench. The performance perfectly correlates with the "sum" column in Table 3.*

idle takes a batch of packets off a relevant NIC receive ring, splits it into sub-batches, and divides the sub-batches between the workers. In a sub-batch, packets are processed in order. For each virtual device *D*, so long as there exists a still-unprocessed packet of *D* designated for processing on the sidecore of worker *W*, then any subsequent requests of *D* will be steered to *W* as well. This policy preserves the order of the original requests and rids network stacks from the need to handle out-of-order packets.

4.2 Interrupts and Exits

Interrupts degrade the performance of virtual setups [5, 29]. But sidecores poll by design, and vRIO carefully uses this property to minimize the overhead. In fact, vRIO completely eliminates the overhead of *virtual* interrupts, partially counteracting the added latency incurred by the extra hop. In this respect, vRIO is superior to Elvis, as will be described next.

Table 3 schematically compares the number of exits and interrupts induced by the virtual I/O models when receiving one network request from an external client and sending back a response. The table contains two vRIO versions—with and without polling at the IOhost. As the comparison is qualitative, we ignore such issues as batching and interrupt coalescing.

Let us go over the table from left to right. Under virtio, a VM induces three synchronous exits: one when sending the response, and two when writing to the end-of-interrupt (EOI) LAPIC register of the virtual hardware after handling the interrupts that notify the VM that (1) a request has arrived and that (2) the response has been sent. The latter interrupts account for the "2" that appears throughout the "guest interrupts" column, as they asynchronously occur in all the I/O models.

Elvis and vRIO do not induce synchronous exits when the virtual NIC is accessed due to their sidecores, whose role is to avoid such exits. Likewise, the optimum avoids these exits as SRIOV provides direct access to the virtual NIC. And none of the models but virtio trigger exits upon EOI because of their use of Eli (which exposes the EOI register directly to the guest by configuring the relevant MSR bitmap to not cause an exit when the guest writes to the EOI register). Similarly, because Eli delivers interrupts of the virtual NIC directly to the VM, all models but virtio avoid the penalty of interrupt injection (4th column in Table 3).

The two virtual device interrupts ("guest interrupts" column) could be the result of two physical interrupts triggered by the backing device, which are handled by the host. The is the case in the baseline and, as it happens, in Elvis ("host interrupts" column). Namely, Elvis's sidecores poll the guest/host shared memory associated with the virtual device (Figure 4b), but they interact with the physical device in the standard interrupt-driven way. In contrast, the optimum and vRIO use SRIOV at the local host and thus do not induce this overhead.

The "IOhost interrupts" column applies to vRIO only. No-poll vRIO treats physical interrupts like Elvis does, letting them drive the IOhost net activity. The number of interrupts is then twice that of Elvis, as every packet reaching the IOhost is received and sent, yielding 4 interrupts per request-response in total. Polling vRIO entirely eliminates this overhead; it polls the NIC, analogously to Elvis's shared memory polling.

Thus, vRIO completely eliminates the overhead of virtual interrupts, allowing it to often outperform Elvis despite the extra hop. Figure 5 supports this analysis by showing that the throughput of ApacheBench is inversely proportional to the "sum" in Table 3 (see experimental details in §5).

4.3 Segmentation and Reassembly

Using Ethernet for VMhost/IOhost communication means that we must segment I/O requests and responses when packet sizes are larger than the maximum transmission unit (MTU), and we must reassemble them at the other end. Of the interposable virtual I/O models, only vRIO suffers from segmentation and reassembly overhead, because shared memory—used by Elvis and the baseline for guest/host communication—does not require segmentation or reassembly.

We optimize the corresponding code paths to reduce this overhead. We configure vRIO to use Ethernet jumbo frames for VMhost/IOhost communication (consisting of 8100 bytes instead of 1500). And we use the TCP segmentation offload (TSO) extension supported by most modern NICs. vRIO works at the raw Ethernet level, but it is nevertheless able to leverage TSO by adding fake TCP/IP headers [20]. This optimization is applicable for chunk sizes as big as 64KB (the maximum buffer size allowed for TCP/IP) and is thus highly effective. Network stacks do not produce packet sizes bigger than 64KB, so the vRIO transport driver only needs to segment block I/O traffic. Reassembly is still done directly by the vRIO software.

4.4 Zero Copying

vRIO copies as little data as possible to minimize CPU consumption and cache pollution. In the IOclient, zero copying is implemented differently for the two front-ends. For the net device, the optimization is straightforward, as the front-end hands a socket buffer (SKB) to the transport driver (T) that already includes all virtio metadata. Then, T adds the aforementioned fake TCP header to the SKB, using existing SKB memory designated for this purpose. It decrements the SKB head pointer to indicate this addition and sends the SKB to the IOhost as explained in §4.3. Receiving is similar: T increments the head pointer and hands the SKB to the front-end.

When the block front-end hands data to T to be *written*, it uses a block I/O buffer, not an SKB. T therefore allocates its own SKB, but it assigns to it pointers of the said block I/O buffer, avoiding a copy. The data is sent, arrives at the IOhost, and is DMAed by the NIC to an IOhost memory buffer. The worker that handles the request reuses this buffer when initiating the corresponding write operation. However, writes to a block device must be aligned to sector size, so the worker uses for zero copy inner portions of the buffer that are aligned, while copying the buffer edges.

When an IOhost *reads*, the data must be copied to the buffers provided by its block system, preventing zero copying.

Zero copy at the IOhost is trickier when handling network messages sent by IOclients. Recall that T utilizes the TSO extension with a fake TCP header to segment messages. The I/O hypervisor must therefore carefully reassemble the original message before it is able to forward it to its destination. The I/O hypervisor starts with an empty SKB. A Linux SKB can map up to 17 fragments, such that each must be contained within a 4KB page. For this reason, vRIO chooses a jumbo frame size of MTU=8100 bytes (rather than the maximal jumbo frame size, which is 9000 bytes). This MTU ensures that each TSO fragment (along with its headers) can be stored in two 4KB pages. At the same time, the maximal size of a single TCP/IP message is 64KB. Thus, the maximal number of per-message TSO fragments is 9, such that 8 of these 9 consist of two 4KB pages and the 9th fragment is smaller than one 4KB page ($64KB - 8100 \times 8 = 736$ bytes). It follows that all the fragments of the original message can be stored in 17 pages as required, ensuring that the I/O hypervisor is able to reassemble the message in a zero copy manner.

4.5 Error Handling

Ethernet is unreliable. Using it as the dedicated communication channel therefore means that vRIO might experience packet loss. For virtual networking I/O, this does not pose a problem, because TCP connections are reliable and retransmit when necessary, and UDP connections might experience loss anyhow. This reasoning does not hold for virtual block I/O, and so vRIO must implement a retransmission mechanism for its block device traffic in order to provide a reliable media.

Block devices may process different requests simultaneously. It is the responsibility of the guest OS disk scheduler (not its driver) to reorder requests, making sure that each *individual* block has only one outstanding request associated with it, while all subsequent requests for that block are pending [10]. Accordingly, vRIO retransmits block requests that it considers lost without worrying that new requests for the same blocks will arrive from the IOclient.

To support retransmission of block requests, vRIO associates a timeout and a unique identifier with each request (or retransmission). The initial timeout is 10ms, and it is doubled upon each subsequent expiration. When a timeout expires, the request is presumed lost and is retransmitted. After the number of unsuccessful retransmissions exceeds some threshold, vRIO concludes that the request cannot be served and raises a device error. vRIO ignores "stale" responses for requests whose unique identifier differs from the current identifier.

We validated the correctness of this mechanism by artificially dropping I/O requests arriving at the IOhost. Furthermore, in our initial experiments, we encountered loss and retransmission of requests "in the wild". vRIO correctly recovered, but the recovery activity affected performance somewhat. Increasing the vRIO receive ring buffers (Rx) from 512 to 4096 packets in the communication channel NIC at the IOhost eliminated this problem.

4.6 Features and Limitations

Device Consolidation Conveniently, efficient sidecore consolidation implicitly allows for efficient I/O device consolidation, for increased utilization. If IOclients are adequately served by fewer devices, consolidation implies comparable or better performance at a lower or the same cost. This benefit is accentuated by expensive devices such as PCIe SSDs. Rack architects can decide to share devices among IOclients instead of limiting their use to individual physical servers. Recall that this type of consolidation is orthogonal, for example, to SANs or NASs: if a SAN/NAS is used directly by IOclients, then there is no interposition, and if it is exposed as a virtual device, then vRIO's advantages come into play.

Friendliness to Heterogeneity Replacing shared memory with network channels as a means of guest-host communication implies that there are no dependencies between the local and remote hypervisors. A handy side-effect of this feature is that the I/O hypervisor is agnostic to the type of the local hypervisor (and its processor architecture), allowing centralized deployment of services and policies that apply to all.

In fact, vRIO does not require a local hypervisor at all. It provides the same services to bare-metal OSes that install its drivers. Bare-metal clouds can thus provide software-based functionality such as firewalls and anti-virus, without "stealing" CPU cycles from the bare-metal machines. (Any software running on the I/O hypervisor cannot be disabled by the IOclient.)

Live Migration The IOhost's transport interface (T) and its net front-end interface (F) are associated with different MAC addresses. T's address is exclusively used for communicating with the IOhost and is unknown to the outside world; any party that wishes to communicate with the IOclient does so via F's address. The T interface can be implemented using an SRIOV instance (T_{sriov}) as described above. But any other NIC will do, notably traditional virtio (T_{virtio}). This architecture facilitates VM live migration between VMhosts that share an IOhost. It allows F to dynamically switch between channeling traffic via T_{sriov} and T_{virtio}. Once T_{virtio} is used, migration can commence as usual. Then, F in the target host switches back to T_{sriov}. Our vRIO implementation correctly runs using T_{virtio}, T_{sriov}, and any other NIC. But we did not implement the dynamic switch.

Migrating a VM away from its IOhost to an arbitrary server (S) is similarly possible, regardless of whether S supports vRIO. First, F notifies the IOhost to stop encapsulating its incoming traffic with vRIO headers. Then, instead of channeling its outgoing traffic (with headers) to the remote IOhost via T_{sriov}, F switches to channel the traffic (without headers) via shared memory to its local hypervisor. This works since the underlying traffic is traditional virtio in both cases. Once F uses traditional virtio, migration can take place as usual. However, if S is a non-vRIO host, it must locally provide the same services previously provided by the IOhost.

Paravirtual block devices (D) migrate as usual: if D is an interface to distributed storage, then this storage will be available in S; otherwise, D's content should be copied to S.

Fault Tolerance vRIO setups as in Figure 2 make the system less fault tolerant. If the IOhost fails, VMhosts cease to be reachable. Reachability in the face of failure can be ensured by connecting VMhosts to their IOhost through the rack switch and configuring the switch to channel IOclient traffic (F addresses) to the appropriate IOhost; this setting requires a costlier switch that can handle the added bandwidth, but it allows for a typical rack arrangement. Alternatively, reachability can be obtained by connecting VMhosts to a secondary fallback IOhost, which requires additional cables and matching ports that would increase price and complexity.

Recall that the IOclient corresponds to F's MAC address. Thus, assuming reachability, the network can recover from an IOhost failure by falling back on regular virtio. Virtual block device activity can similarly recover if D is backed by distributed storage; otherwise, if the storage resides exclusively on the IOhost, losing it is akin to losing a local drive.

Energy An inherent downside of the sidecore approach is that polling consumes energy. In principle, this cost can be reduced [4] by trading off some latency and utilizing the CPU's monitor/mwait capability, which enables the core to enter a low-power state until a monitored cache range is modified [8]. This optimization is outside the scope of this work.

5. Evaluation

Methodology We do not have the resources to conduct an evaluation with setups as described in §3. Instead, we focus on the individual (side)core and compare the performance it delivers in the different I/O models. Our testbed system consists of up to 7 servers: two VMhosts, one IOhost, and four load generators. The VMhosts are IBM System x3550 M4 machines, each with: two 8-core Intel 2.2GHz Xeon E5-2660 CPUs; 56GB memory; and an Intel x520 dual port 10Gbps SRIOV NIC, allowing 20Gbps per VMhost. The IOhost is an IBM System x3650 M4 with: two 8-core Intel 2.7GHz Xeon E5-2680 CPUs; 128GB memory (we made sure the IOhost only uses a small fraction of it); and two NICs identical to that of the VMhosts, supporting 40Gbps. The load generators are IBM System x3550 M2 machines, each with: two 4-core 2.93GHz Intel Xeon 5500 CPUs; 12GB memory; and an Emulex OneConnect dual port 10Gbps NIC. VMhost hypervisors are KVM/QEMU, hosting VMs configured with one VCPU and 1GB of memory each, backed by 2MB pages. All machines run Linux 3.9: guests, hosts, and bare metal. We turn off hyperthreading and all power

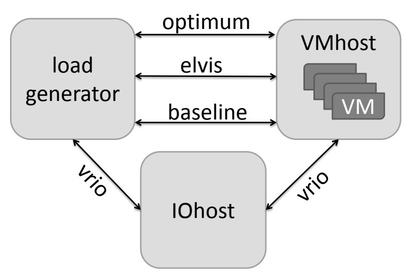

Figure 6. *Simplest experimental setup.*

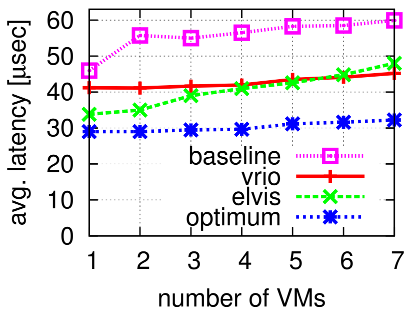

Figure 7. *Netperf RR latency.*

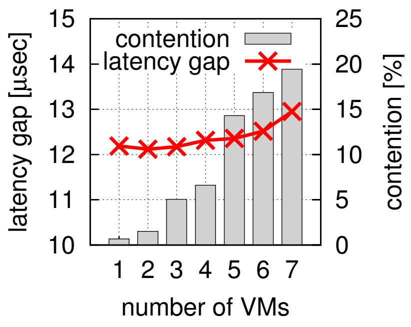

Figure 8. *Netperf RR/vRIO contention.*

optimizations—sleep states (Cstates) and dynamic voltage and frequency scaling (DVFS)—to avoid reporting artifacts caused by nondeterministic events.

Although IOhost specialization is an inherent vRIO advantage, our IOhost is different from the VMhosts because we did not have enough identical machines available. Note, however, that we only use a small fraction of the IOhost in our experiments and carefully equalize the number of cores available to each competing setup, as explained in detail below.

We evaluate vRIO against the I/O models we have considered thus far (§2)—the KVM/virtio baseline as the state of practice [58], Elvis as the state of the art [31], and SRIOV+Eli as the non-interposable optimum [5, 29]. We use the benchmarks: (**1**) Netperf UDP RR (request-response), a standard tool to measure network latency [37], repeatedly sending one byte and waiting for a byte response; (**2**) Netperf TCP stream, a tool to measure the maximal throughput sent over one TCP connection—we use a 64B packet size to stress the I/O models, as bigger sizes quickly saturate the network links and hide the differences; (**3**) Apache [24, 25], an HTTP web server driven by ApacheBench [9]; (**4**) Memcached [26], an in-memory key-value storage server driven by Memslap [3]; and (**5**) Filebench, a filesystem and storage benchmark that generates both micro and macro workloads (as specified later) [48].

We execute each experiment 5 times and present averages. The standard deviation for all the Elvis, vRIO, and optimum models is less than 2% of the average, and it is less the 5% for all the baseline experiments, which are less stable due to nondeterministic scheduling of vhost threads.

Latency The most troubling aspect of vRIO is the impact of adding a hop to the I/O path. We evaluate this impact in the *least* favorable conditions for vRIO—when no interposition is involved. With interposition, the relative weight of vRIO overheads is reduced. For example, if interposition takes 100μs per-packet and vRIO adds 10μs, then performance drops by less than 10% regardless of the I/O model we compare against. We initially use our simplest experimental setup, depicted in Figure 6. For all I/O models but vRIO the setup consists of one VMhost directly communicating with one generator. With vRIO, the two communicate indirectly via the IOhost. Only one 8-core CPU is used on the VMhost.

Using Netperf RR, we measure the latency of the I/O models as a function of N—the number of VMs in the VMhost that run the benchmark. For each given N, we constrain the number of active cores to be $N+1$ for Elvis, vRIO, and the baseline: for Elvis, the extra core is in the VMhost serving as a sidecore; the baseline is similar, but Linux uses the core to run I/O threads and VCPUs as it pleases; and for vRIO, the extra (side)core resides on the IOhost. In the optimum SRIOV/Eli setup, we use only N cores, as an extra core would stand idle with only N VMs, seeing it has no host I/O threads or interrupts to process. (Later, we conduct another experiment where we equalize the number of cores for the optimum setup.)

The results, shown in Figure 7, indicate that the optimum enjoys close to perfect scalability, with 30–32μs latency on average per request-response; vRIO exhibits a similar slope, yet its latency is about 12μs higher—-this is the cost of the added hop. A closer examination of the difference between the latencies of vRIO and the optimum reveals that it slowly increases with N from about 12μs to 13μs, as depicted in Figure 8 (left y-axis). The small increase is due to "contention" over the remote sidecore of vRIO, as exemplified by Figure 8 (right y-axis), which shows the fraction of packets that had to wait at the IOhost before being processed.

The real competitor of vRIO, however, is Elvis, as both support interposition whereas the optimum does not. Initially ($N=1$), the latency of vRIO is 8μs longer than Elvis's. This gap pertains to the 1.18x increased latency we report in the introduction, and it is the highest performance degradation induced by vRIO that we observed for network workloads. As the number of VMs increases, the gap shrinks until vRIO becomes faster than Elvis at $N=6$ due to the manner by which vRIO reduces interrupts activity (§4.2).

The tail latency is shown in Table 4. The results are mixed, with Elvis and vRIO fairing better at different percentiles: Elvis has lower 99.9% and 99.99% latency, but vRIO has a lower 99.999% and maximal latency.

Throughput The Netperf stream throughput is shown in Figure 9. Elvis and the optimum achieve similar performance, and vRIO is 5–8% lower. The degradation is due to the added processing time incurred by the vRIO driver, quantified in

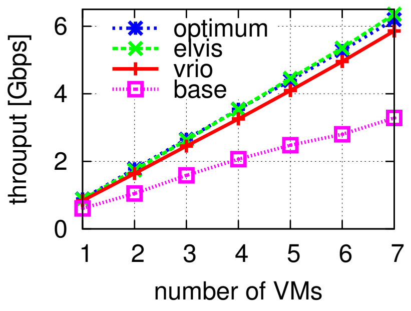

Figure 9. *Netperf stream throughput of vRIO is 5–8% smaller than the optimum.*

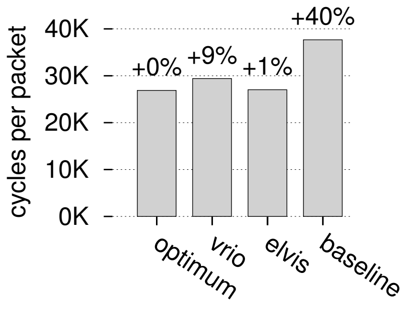

Figure 10. *Netperf stream's per-packet processing time with one VM (N=1).*

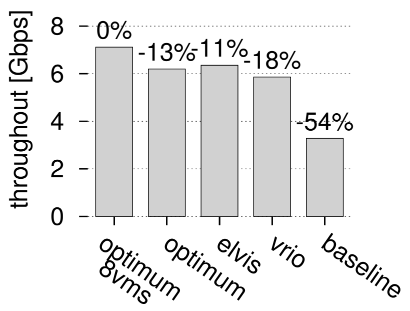

Figure 11. *Optimum with N+1=8 cores ("8vms") shows the cost of interposition.*

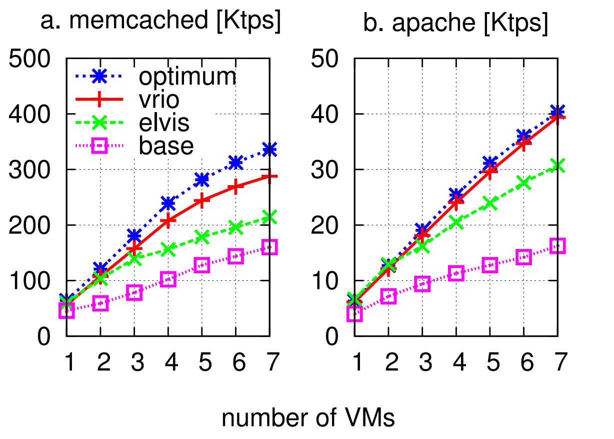

Figure 12. *Benchmarks performance (kilo transactions/second).*

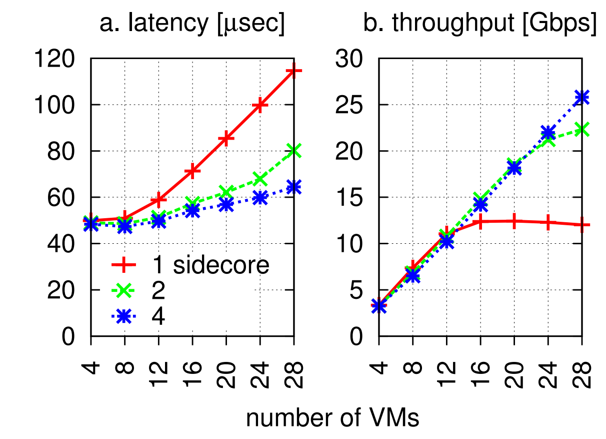

Figure 13. *vRIO's IOhost scalability—latency and throughput.*

percentile	*optimum*	*elvis*	*vrio*
99.9%	35	53	60
99.99%	42	71	156
99.999%	214	466	258
100%	227	480	274

Table 4. *Tail latency in microseconds for one VM.*

Figure 10. Since vRIO spends 9% more cycles on processing each packet, its throughput is lower.

The number of VMs in our experiments so far was fixed (N), so the optimum used one less core. Next, we equalize the number of cores by comparing the $N = 7$ throughput results (Figure 9) to that of the optimum setup utilizing 8 cores to run $N = 8$ VMs. Figure 11 shows the outcome. As expected, this setup yields superior throughput, highlighting the price paid for having the ability to interpose on the I/O of the VMs.

Macrobenchmarks Figure 12 displays the performance of Memcached and Apache. The results—showing that vRIO approaches the optimum whereas Elvis falls behind—coincide with that of the latency experiment for higher N values (Figure 7). As noted in §4.2, when load is high, Elvis is inferior to vRIO, because it needs to handle additional physical interrupts, whereas vRIO polls the device and thus avoids the associated overheads. Notably, the cost of interrupts is substantial despite the fact that both the hardware (NIC) and software (OS) employ interrupt coalescing [5, 29].

IOhost Scalability When consolidating the I/O processing of several VMhosts onto one IOhost, scalability is a concern. We assess vRIO's scalability by serving *all* the machines at our disposal (see beginning of §5) with one IOhost. We configure the two 16-core VMhosts to simulate four 8-core VMhosts: each physical VMhost has two 8-core CPUs and two 10Gbps ports, so we partition it to two logical VMhosts such that each CPU is exclusively associated with a single port.

In this experiment, we systematically increase the load by adding one VM to each of the four logical VMhosts. In the first run there are 4 VMs (one per VMhost), in the second there are 8, and in the final run there are 28 (seven VMs per VMhost). Each VMhost is connected to its own load generator, yielding a setup where the IOhost collectively serves eight multicore machines divided to four VMhost/load-generator pairs.

Figure 13a shows the results of this experiment with Netperf RR and an IOhost that utilizes 1, 2, and 4 sidecores. More sidecores reduce the observed latency. Yet, even with 4 sidecores, the latency increases. This degradation, however,

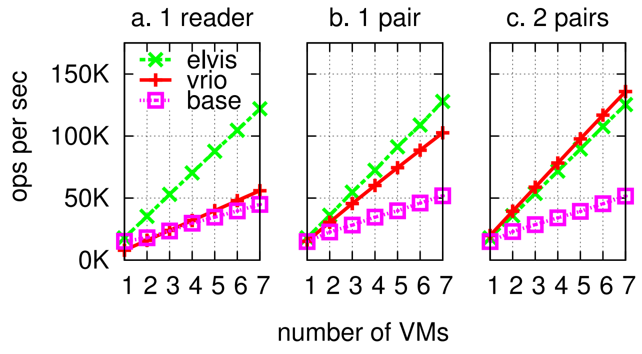

Figure 14. *The vRIO Filebench/ramdisk results improve with increased concurrency until, counterintuitively, it outperforms Elvis.*

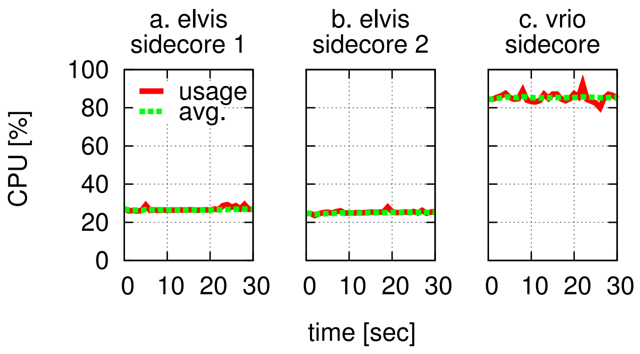

Figure 15. *The CPU utilization and running average indicate that vRIO makes better use of its sidecore cycles as compared to Elvis.*

turned out to be due to NUMA effects on the load generators. Each generator has two 4-core CPUs with a single PCIe bus that is directly connected to CPU 0. Initially, this CPU generates the load. Since we designate one core for interrupt handling only, $N = 16$ is the first run that uses a core from CPU 1, causing the drop in performance. Beyond that, every additional core being used on CPU 1 further increases the average DRAM latency and hence the overall performance.

We repeat the experiment using Netperf stream (Figure 13b) and find that throughput scales linearly while sidecores are not saturated, and that a sidecore gets saturated when servicing about 13 VMs, processing around 13Gbps. Indeed, the curves are converged for N=4,8,12 (and then for N=16,20,24). We conclude that handling VMs from different VMhosts on the same sidecore does not affect performance; only the *number* of VMs is significant, regardless of where the VMs are hosted.

Making a Local Device Remote Suppose we want to accelerate VM storage performance using fast I/O block devices, but we are unwilling to purchase a device for each physical server, or we want all servers to enjoy the devices of the other servers. Further suppose that the devices must be interposable (unlike, say, SAN), so they cannot be used directly by the VMs. With vRIO, these goals can be easily achieved by placing the device(s) at the IOhost rather than locally. We next evaluate the performance implications.

We simulate the device using ramdisk, exposing 1GB to each VM. We use three instances of Filebench: (1) a reader, (2) a reader and a writer ("pair"), and (3) two readers and two writers ("2 pairs"), such that the readers and writers perform 4KB random I/O within their VM. To avoid benchmarking the guests' filesystem cache, we open the virtual block device with O_DIRECT, so all I/O requests pass the guest-host boundary. We do not benchmark the optimum setup, because there is no such thing as an SRIOV ramdisk.

Figure 14 depicts the results. The reader effectively measures latency (inverse of throughput). It shows that vRIO scales somewhat better than the baseline but worse than Elvis. This result refers to the 2.2x latency reported in the introduction. vRIO improves with one pair, and, counterintuitively, outperforms Elvis with two pairs. The reason is the number of involuntary context switches in Elvis guests, which is two orders of magnitude higher as compared to vRIO. The relatively low latency of the local ramdisk combined with the relatively high number of CPU cycles required to process each request results in threads contending over the CPU, degrading throughput.

We utilized a ramdisk to approximate the overhead incurred by vRIO on future, faster I/O devices. When applied to SATA SSDs available to us, the reader's baseline and vRIO throughput become 75%–95% and 83%–95% relative to Elvis.

Improving Utilization Resources consolidation is suitable when VMs make partial use of them. In our context, it means that their I/O is semi-intensive and so they do not require all the computational power of their sidecores. Our next experiment consists of two VMhosts, each running five VMs, utilizing 5+1 cores (N=5). All VMs run Filebench's Webserver I/O personality, modeling a block device workload generated by a typical webserver. The workload includes 30K files of variable sizes with a mean size of 28KB. Each webserver has 4 threads performing open, read, and close operations while updating a log file. We empty the hypervisor page caches before each run. We evaluate three setups: (1) Elvis with two sidecores, one per VMhost; (2) vRIO with a single "consolidated" sidecore at the IOhost serving both VMhosts; and (3) the baseline serving the five VMs with six cores (N+1) on each VMhost. All VMs have a 1GB ramdisk block device, but in vRIO it is remote rather than local.

Figures 15a–b show the CPU used by the Elvis sidecores. Both are underutilized, spending together about 150% CPU on useless polling. Figure 15c shows that the consolidated vRIO sidecore is more effective. Figure 16a shows the resulting throughput. We see that vRIO is significantly better than the baseline but is 8% below Elvis. This outcome constitutes a typical consolidation tradeoff in virtual environments: sacrificing some performance (-8%) to get significant savings in physical resources (halving the sidecore number).

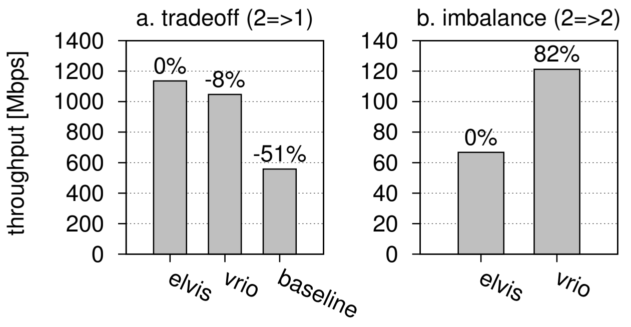

Figure 16. *With sidecore consolidation, we can use fewer or the same resources and get comparable (a) or better (b) performance.*

Load Imbalance Resource consolidation is most suitable to handle load imbalance. Suppose our budget consists of two sidecores, and we need to decide where to place them. In vRIO, we consolidate them at the IOhost, whereas in Elvis we allocate one per VMhost. Suppose that only one VMhost currently requires service (the Webserver personality of Filebench) while the other is idle or engaged in activity that requires little I/O. To increase the imbalance, the Webserver makes use of I/O interposition for seamless encryption. We use AES-256 as the encryption algorithm and invoke it through standard Linux APIs. In vRIO, two sidecores can process the VMhost's I/O and encryption, whereas in Elvis the VMhost can only make use of its one local sidecore. Figure 16b shows the outcome: vRIO provides an 82% improvement over Elvis with the same sidecore budget. This performance boost is an inherent benefit of vRIO.

Heterogeneity Finally, we demonstrate that vRIO enables hypervisor agnostic I/O interposition. We run Netperf stream within a Linux IOclient that is: (1) a VM hosted on VMware ESXi 5; (2) a VM hosted on KVM; and (3) a bare metal OS. All setups attain line rate and have comparable CPU utilization in both sidecore and VMcore. We perform another test involving an IBMPOWER 710 with a 1Gbps Intel 82571EB NIC to show that vRIO is also hardware platform agnostic. We install the vRIO driver on the host OS, making it a bare-metal client that involves no virtualization. We run the same test on an x86 guest after installing a 1Gbps NIC in it so as to have comparable setups. Here too both servers attain line rate and have comparable CPU usage (under 10%). We conclude that it is possible to seamlessly run vRIO services on the IOhost for multiple hardware platforms and hypervisor types.

6. Related Work

The sidecore approach is a parallelization technique to accelerate the execution of virtual machines by polling relevant memory regions via host cores external to the VMs. Rather than taking exits on the cores that run the VMs, the host sidecores continuously observe the requests of VMs and service them accordingly, thereby (1) eliminating the direct and indirect overheads of exits, and (2) offloading the corresponding work to sidecores, freeing VM cores to process other activities. In essence, the sidecore approach offloads work from guest cores to host cores while replacing the costly trap-and-emulate guest/host communication mechanism with simple, much faster load/store operations directed at the coherent memory caches.

The sidecore approach was first introduced in 2007 by Kumar et al. [27, 39]. They used it to accelerate network interrupts and page table management processing of paravirtual guests. In 2009, Liu and Abali reintroduced the same concept, calling it virtualization polling engine (VPE) and using it to additionally accelerate the receive and transmit network I/O paths [45]. In 2012, Ben-Yehuda et al. implemented a block device on a sidecore [13]. In 2013, Har'el et al. proposed Elvis [31]—used in this paper—combining the sidecore paradigm with exitless interrupts [5, 29] and making it applicable to any virtio device (block, as well as net) such that it linearly scales with the number of cores (VMs).

All of the aforementioned sidecore work was done using paravirtualization, whereby the guest VM code is changed to explicitly cooperate with a host that runs on sidecore, rather than on the same core with regular trap-and-emulate exits. Conversely, Amit et al. exposed a virtual IOMMU ("vIOMMU") to unmodified VMs, and they were able to substantially improve performance by running the vIOMMU code on a sidecore, keeping the VMs unaware of the fact they are not using the physical IOMMU [4].

Landau et al. proposed a hardware extension ("SplitX") that allows the hypervisor and its unmodified guests to run on disjoint sets of cores, such that *every* exit occurring on a VM core is delivered to, and is serviced by, a host sidecore [40].

The insight underlying vRIO is that, logically, the functionality of sidecores is very similar to that of I/O devices, and therefore it could be advantageous to consolidate them. Thus, different than all the above studies, vRIO takes the sidecore approach a step further by migrating sidecores to another server, reaping the benefits of consolidation as outlined above. In this respect, vRIO is reminiscent of supercomputer architectures that distinguish between I/O nodes and compute nodes [2, 18, 32, 42, 61]. But such architectures—if used to run VMs—either hinder programmable interposition or induce the overheads of traditional interposable virtual I/O.

7. Conclusions

Given that sidecores are an effective way to optimize for virtual I/O, we show that it makes sense—in terms of price and performance—to consolidate them remotely across the network rather than to spread them across the rack.

Acknowledgments

We thanks the anonymous reviewers for their feedback. This research was funded in part by the Israeli Ministry of Science and Technology (grant #3-9779) and the European Union's Seventh Programme (grant agreement No. FP7-609828).

References

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, 2006. `http://dx.doi.org/10.1145/1168857.1168860`.

[2] N.R. Adiga, G. Almasi, G.S. Almasi, Y. Aridor, R. Barik, and many others. An overview of the BlueGene/L supercomputer. In *ACM/IEEE Supercomputing (SC)*, pages 1–22, 2002. `http://dx.doi.org/10.1109/SC.2002.10017`.

[3] Brian Aker. Memslap - load testing and benchmarking a server. `http://docs.libmemcached.org/bin/memslap.html`. libmemcached 1.1.0 documentation. Accessed: Jan 2015.

[4] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011. `http://www.usenix.org/events/atc11/tech/final_files/Amit.pdf`.

[5] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Bare-metal performance for virtual machines with exitless interrupts. *Communications of the ACM (CACM)*, 59(1):108–116, Jan 2016. `http://dx.doi.org/10.1145/2845648`.

[6] Nadav Amit, Dan Tsafrir, and Assaf Schuster. VSwapper: A memory swapper for virtualized environments. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–366, 2014. `http://dx.doi.org/10.1145/2541940.2541969`.

[7] M.R. Anala, M. Kashyap, and G. Shobha. Application performance analysis during live migration of virtual machines. In *IEEE International Advance Computing Conference (IACC)*, pages 366–372, 2013. `http://dx.doi.org/10.1109/IAdCC.2013.6514252`.

[8] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2008. `http://dx.doi.org/10.1109/IPDPS.2008.4536358`.

[9] Apachebench. `http://en.wikipedia.org/wiki/ApacheBench`. Accessed: Jan 2015.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, chapter 37. Arpaci-Dusseau Books, LLC, 0.90 edition, 2015. `http://pages.cs.wisc.edu/~remzi/OSTEP/file-disks.pdf`.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003. `http://dx.doi.org/10.1145/945445.945462`.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 49–65, 2014. `https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf`.

[13] Muli Ben-Yehuda, Eran Borovik, Michael Factor, Eran Rom, Avishay Traeger, and Ben-Ami Yassour. Adding advanced storage controller functionality via low-overhead virtualization. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 187–194, 2012. `https://www.usenix.org/legacy/events/fast12/tech/full_papers/Ben-Yehuda2-2-12.pdf`.

[14] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 423–436, 2010. `http://www.usenix.org/events/osdi10/tech/full_papers/Ben-Yehuda.pdf`.

[15] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)*, pages 9–20, 2007. `https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=9`.

[16] Jason Blosil, David Fair, and David Fair. 10GbE – key trends, drivers and predictions. SNIA presentation: `http://www.snia.org/sites/default/files/SNIA_ESF_10GbE_Webcast_Final_Slides.pdf`, July 2012. (Accessed: Aug 2015).

[17] Vojtech Cima. An analysis of the performance of block live migration in openstack. URL `http://tinyurl.com/live-migration-openstack`, 2014. Accessed: Jan 2016.

[18] Peter F. Corbett and Dror G. Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems (TOCS)*, 14(3):225–264, Aug 1996. `http://dx.doi.org/10.1145/233557.233558`.

[19] Crehan Research. Another year of robust growth and record shipments for branded data center switches. `http://www.crehanresearch.com/wp-content/uploads/2015/03/CREHAN-2014-Data-Center-Switching-CR.pdf`, Mar 2015. (Accessed: Aug 2015).

[20] B. Davie and J. Gross. A stateless transport tunneling protocol for network virtualization (STT). `http://tools.ietf.org/id/draft-davie-stt-04.txt`, Sep 2013. Network Working Group; Internet-Draft; draft-davie-stt-04. Accessed: Jan 2016.

[21] Dell Inc. Dell PowerEdge R930 4-socket rack server. `http://www.dell.com/us/business/p/poweredge-r930/pd`, Jul 2015. Prices as of 31 July, 2015.

[22] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–10, 2010. `http://dx.doi.org/10.1109/HPCA.2010.5416637`.

[23] Virtual I/O acceleration—the Elvis source code. https://github.com/abelg/virtual_io_acceleration, 2013. (Accessed: Mar 2015).

[24] The Apache HTTP server project. http://httpd.apache.org. Accessed: Jan 2015.

[25] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. *IEEE Internet Computing*, 1(4):88–90, Jul 1997. http://dx.doi.org/10.1109/4236.612229.

[26] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, Aug 2004. http://dl.acm.org/citation.cfm?id=1012889.1012894.

[27] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High-performance hypervisor architectures: Virtualization in HPC systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, 2007. http://www.csm.ornl.gov/srt/hpcvirt07/papers/paper10.pdf.

[28] Robert P Goldberg. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974. http://dx.doi.org/10.1109/MC.1974.6323581.

[29] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–422, 2012. http://dx.doi.org/10.1145/2150976.2151020.

[30] James Hamilton. AWS innovation at scale. https://www.youtube.com/watch?t=113&v=JIQETrFC_SQ, Nov 2014. (Accessed: Aug 2015).

[31] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual I/O system. In *USENIX Annual Technical Conference (ATC)*, pages 231–242, 2013. https://www.usenix.org/conference/atc13/technical-sessions/presentation/har%E2%80%99el.

[32] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, and many others. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, March 2012. http://dx.doi.org/10.1109/MM.2011.108.

[33] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. When slower is faster: On heterogeneous multicores for reliable systems. In *USENIX Annual Technical Conference (ATC)*, pages 255–266, 2013. https://www.usenix.org/conference/atc13/technical-sessions/presentation/hruby.

[34] Intel. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html, Jan 2011.

[35] Intel Corporation. Intel processor pricing effective June 07, 2015. http://tinyurl.com/intel-cpu-prices-2015-07, June 2015. (Accessed: Aug 2015, via http://www.intel.com/go/processor_pricing).

[36] Hai Jin, Li Deng, Song Wu, Xuanhua Shi, and Xiaodong Pan. Live virtual machine migration with adaptive, memory compression. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, 2009. http://dx.doi.org/10.1109/CLUSTR.2009.5289170.

[37] Rick A. Jones. A network performance benchmark (Revision 2.0). Technical report, Hewlett Packard, 1995. http://www.netperf.org/netperf/training/Netperf.html. Accessed: Jan 2015.

[38] Asim Kadav and Michael M. Swift. Live migration of direct-access devices. In *USENIX Workshop on I/O Virtualization (WIOV)*, pages 95–104, 2008. http://usenix.org/events/wiov08/tech/full_papers/kadav/kadav.pdf.

[39] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting VMMs for multicore systems: The *sidecore* approach. In *Workshop on Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007. http://www.ideal.ece.ufl.edu/workshops/wiosca07/Paper3.pdf.

[40] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011. http://www.usenix.org/events/wiov11/tech/final_files/Landau.pdf.

[41] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, 2004. https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/levasseur/levasseur.pdf.

[42] Jianwei Li, Wei-Keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *ACM/IEEE Supercomputing (SC)*, page No. 39, 2003. http://dx.doi.org/10.1109/SC.2003.10053.

[43] Guangdeng Liao, Danhua Guo, Laxmi Bhuyan, and Steve R King. Software techniques to improve virtualized I/O performance on multi-core systems. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 161–170, 2008. http://dx.doi.org/10.1145/1477942.1477971.

[44] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2010. http://dx.doi.org/10.1109/IPDPS.2010.5470365.

[45] Jiuxing Liu and Bulen Abali. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM International Conference on Supercomputing (ICS)*, pages 225–23, 2009. http://dx.doi.org/10.1145/1542275.1542309.

[46] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX Annual Technical Conference (ATC)*,

pages 29–42, 2006. http://usenix.org/legacy/event/usenix06/tech/liu.html.

[47] Dražen Lučanin, Foued Jrad, Ivona Brandic, and Achim Streit. Energy-aware cloud management through progressive SLA specification. In *International Conference on Economics of Grids, Clouds, Systems, and Services (GEOCON)*, pages 83–98. Springer, 2014. http://dx.doi.org/10.1007/978-3-319-14609-6_6.

[48] J. Mauro, S. Shepler, and V. Tarasov. Filebench. http://sourceforge.net/projects/filebench/. Accessed: Oct, 2012.

[49] Eyal Moscovici, Yossi Kuperman, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Dynamic sidecore allocation. In preparation.

[50] David Mytton. Network performance at AWS, Google, Rackspace and Softlayer. https://blog.serverdensity.com/network-performance-aws-google-rackspace-softlayer, Apr 2014. (Accessed: Aug 2015).

[51] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. CompSC: Live migration with pass-through devices. In *ACM International Conference on Virtual Execution Environments (VEE)*, pages 109–120, 2012. http://dx.doi.org/10.1145/2151024.2151040.

[52] PCI-SIG. Single root I/O virtualization and sharing 1.1 specification. http://www.pcisig.com/specifications/iov/single_root/, Jan 2010.

[53] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 1–16, 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf.

[54] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM (CACM)*, 17(7):412–421, Jul 1974. http://dx.doi.org/10.1145/361011.361073.

[55] Jesse Proudman. Live migration is a perk, not a panacea. URL https://www.blueboxcloud.com/insight/blog-article/live-migration-is-a-perk-not-a-panacea, Oct 2014.

[56] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 179–188, 2007. http://dx.doi.org/10.1145/1272366.1272390.

[57] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-Switch: A scalable software virtual switching architecture. In *USENIX Annual Technical Conference (ATC)*, pages 13–24, 2013. https://www.usenix.org/conference/atc13/technical-sessions/presentation/ram.

[58] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM Operating System Review (OSR)*, 42(5):95–103, 2008. http://dx.doi.org/10.1145/1400097.1400108.

[59] Felix Salfner, Peter Tröger, and Andreas Polze. Downtime analysis of virtual machine live migration. In *International Conference on Dependability (DEPEND)*, pages 100–105, 2011. http://tinyurl.com/DEPEND11-vm-downtime.

[60] SanDisk Corp. Fusion ioMemory SX300 spec. http://www.fusionio.com/load/-media-/302x3j/docsLibrary/SX300_DS_Final_v3.pdf, 2014. (Accessed: Aug 2015).

[61] K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *ACM/IEEE Supercomputing (SC)*, page No. 57, 1995. http://dx.doi.org/10.1145/224170.224371.

[62] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, 2007. http://dx.doi.org/10.1145/1294261.1294294.

[63] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 485–498, 2003. http://dx.doi.org/10.1145/2451116.2451169.

[64] R. Takano, H. Nakada, T. Hirofuchi, Y. Tanaka, and T. Kudoh. Cooperative VM migration for a virtualized HPC cluster with VMM-bypass I/O devices. In *IEEE International Conference on E-Science (e-Science)*, pages 1–8, 2012. http://dx.doi.org/10.1109/eScience.2012.6404487.

[65] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. Secure I/O device sharing among virtual machines on multiple hosts. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 108–119, 2013. http://dx.doi.org/10.1145/2485922.2485932.

[66] VMware, Inc. VMware ESX server 2 - architecture and performance implications. Technical Report ESX-ENG-Q305-122, VMware, 2005. https://www.vmware.com/pdf/esx2_performance_implications.pdf.

[67] VMware, Inc. Performance evaluation of VMXNET3 virtual network device. http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf, 2009.

[68] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *International Conference on Cloud Computing*, pages 254–265. Springer-Verlag, 2009. http://dx.doi.org/10.1007/978-3-642-10665-1_23.

[69] Carl Waldspurger and Mendel Rosenblum. I/O virtualization. *Communications of the ACM (CACM)*, 55(1):66–73, Jan 2012. http://dx.doi.org/10.1145/2063176.2063194.

[70] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and

networked applications. Technical Report 02-02-01, University of Washington, 2002. http://www.cs.washington.edu/tr/2002/02/UW-CSE-02-02-01.pdf.

[71] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)*, pages 15–28, 2008. https://www.usenix.org/legacy/event/usenix08/tech/full_papers/willmann/willmann.pdf.

[72] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 306–317, 2007. http://dx.doi.org/10.1109/HPCA.2007.346208.

[73] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 17–17, 2007. http://usenix.org/event/nsdi07/tech/full_papers/wood/wood.pdf.

[74] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. vTurbo: Accelerating virtual machine I/O processing using designated turbo-sliced core. In *USENIX Annual Technical Conference (ATC)*, pages 243–254, 2013. https://www.usenix.org/conference/atc13/technical-sessions/presentation/xu.

[75] Frank Yang. 10GBASE-T economics. http://www.commscope.com/Blog/10gbase-t-economics/, 2012. (Accessed: Aug 2015).

[76] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical Report H-0263, IBM Research, 2008. http://tinyurl.com/IBM-TR-H-0263.

[77] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symposium (OLS)*, pages 261–268, 2008. https://www.kernel.org/doc/ols/2008/ols2008v2-pages-261-267.pdf.