vRIO: Efficient Paravirtual Remote I/O

Yossi Kuperman
vRIO: Efficient Paravirtual Remote I/O

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Yossi Kuperman

Submitted to the Senate
of the Technion — Israel Institute of Technology
Sivan 5774 Haifa June 2014
This research was carried out under the supervision of Prof. Dan Tsafir, in the Faculty of Computer Science.

Acknowledgements

First and foremost, I would like to thank my research supervisor Prof. Dan Tsafir for all of his great help and support. Without his assistance and dedicated involvement in every step throughout the process, both from the personal and professional sides, this thesis would have never been accomplished.

My sincere thanks also go to Abel Gordon, who introduced me to the Linux kernel, and whose profound insights were invaluable. In addition, I would like to thank my colleagues at IBM: Eran Raichstein, Eyal Moscovici, Joel Nider and Razya Ladelsky, for their help to assemble all the research parts together and for their productive suggestions along the way.

I’m grateful to the Ministry of Science for their generous financial support.

Last but not least, I would like to express appreciation to my beloved wife, Maayan, who spent sleepless nights with me and was always my support in moments it seemed almost hopeless.

The generous financial help of the Technion and Minster of Science is gratefully acknowledged.
## Contents

List of Figures

Abstract .......................... 1

1 Introduction ...................... 3

2 Related Work ..................... 5

3 Design ............................ 9
   3.1 Paravirtual Protocol .......... 10
   3.2 Network Link ................. 11
   3.3 vRIO Driver Model ........... 11
   3.4 I/O Hypervisor ............... 12
   3.5 vRIO Challenges ............. 13

4 Implementation .................. 15

5 Benefits ......................... 17

6 Evaluation ....................... 19
   6.1 Experimental Methodology ... 19
   6.2 Worst case performance ...... 21
   6.3 Benefits of consolidation ... 26
   6.4 Heterogeneity ............... 29

7 Conclusions ...................... 31
List of Figures

3.1 vRIO architecture — The system is divided into I0client (front-end) and I0host (back-end) ................................................. 10
3.2 Service Consolidation .......................................................... 12

6.1 In vRIO, the workload generator is connect to the VMhost via the IOhost, whereas in the three competing I/O models the generator and the VMhost are connected directly. ........................................... 21
6.2 Latency achieved by Netperf UDP RR (request-response). ............ 22
6.3 The difference in Netperf UDP RR latency between vRIO and the optimum (SRIOV with exitless interrupts) moderately increases due to contention over the remote IOcore. ............................................. 23
6.4 Throughput achieved by Netperf TCP stream. ............................. 23
6.5 Average per-packet processing time in Netperf TCP stream with N = 1 (one VM). ......................................................... 24
6.6 Running Netperf TCP stream in the optimum configuration using 8 cores and 8 VMs demonstrates the price of I/O models allowing for interposing. .............. 24
6.7 Performance of Apache and Memcached under the four I/O virtualization models. .............................................................. 25
6.8 Turning off polling in vRIO and enabling interrupt instead dramatically reduces the performance of ApacheBench. ........................................ 26
6.9 Filebench utilizing a ramdisk block device: (a) one reader; (b) one reader/writer pair; and (c) two reader/writer pairs ................................. 27
6.10 Filebench with one random reader utilizing 128GB SSD block device; each VM assigned a 16GB partition. ........................................ 27
6.11 Performance of the consolidation experiments. ................................ 28
6.12 CPU utilization of the two ELVIS IOcores and the consolidated vRIO core. ................................................................. 28
6.13 Supporting different architectures and hypervisors in vRIO. ............ 30
Abstract

Dedicating host cores for processing the paravirtual I/O of guest virtual machines (VMs) has been shown to greatly improve VM performance. This new virtual I/O model, however, requires the dedication of cores on each physical server, which is wasteful when there is not enough I/O activity. Likewise, the combined computational power of the dedicated per-server cores might not be enough when I/O activity is high. In the context of rack-scale computing, we propose to solve this problem by consolidating the dedicated cores spread across several servers onto one server. In this design, the processing of the paravirtual I/O of all of the VMs is done remotely, on the server that aggregates the dedicated cores. The hypervisor is therefore effectively split into two parts. We call this new model paravirtual remote I/O (vRIO), and we investigate its tradeoffs. The inherent downside of vRIO is that it prolongs I/O latency. We manage to keep it to a maximum of 18% longer than state-of-the-art paravirtualization for network I/O. The latency becomes up to 2x longer if, in addition to the dedicated core, we also migrate (and thus consolidate) high-end local block devices. The benefits are that we achieve (1) comparable throughput by using fewer dedicated cores and (2) superior throughput by using the same number of cores.
Chapter 1

Introduction

The de-facto standard for realizing virtual I/O is through *paravirtualization*, exemplified by KVM’s virtio [29] and VMWare’s VMXNET [34]. In paravirtual I/O, the host presents a virtual I/O device to its guest. The guest is aware of the hypervisor and communicates with it directly through a pre-defined API. The device interface is not modeled after any real device, but rather it is optimized to minimize overhead of guest/host communication. Utilization of the physical I/O devices is improved via time multiplexing. Multiple physical backing devices can be aggregated into the same virtual device, which enables load balancing between channels, and failure masking. Hosts can interpose (control and manipulate) the I/O activity of their guests and thus support such features as live-migration, software-defined networking (SDN), file-based images, replication, snapshots, and security related functionalities such as record-replay, encryption, firewalls, and intrusion detection.

The interposition feature of paravirtual I/O comes with a significant performance penalty for I/O-intensive guests, as already noted in previous work [7, 11, 19, 22, 35]. Traditional paravirtual I/O implementations suffer from two problems. The first is the slowdown of a single guest, mainly caused by exits [1] (context switches from guest to hypervisor). The second is lackluster scalability - when the host has multiple I/O-intensive guests, the competition between these guests causes a significant reduction in throughput and an increase in latency [16].

One way to handle the drawbacks of paravirtual I/O is with PCI passthrough. PCI passthrough has been shown to have superior throughput and latency as compared to paravirtual I/O, because the VMs communicate directly with a physical device, thus bypassing the hypervisor on the data path. However, the performance delivered by PCI passthrough comes at the cost of losing out on other important features. The most significant drawback is that PCI passthrough does not allow for interposition on the data, which is required to control and manipulate the network and block I/O of the VMs.

Another approach that improves on paravirtualization yet also retains the interposition capability is with sidecores [4, 7, 16, 18, 19, 20, 23]. The sidecore approach
leverages the industry trends of increasing number of CPU cores by offloading tasks (such as I/O processing) to a small set of dedicated cores. These dedicated cores are either underutilized when there is not enough I/O traffic to process, or they become the bottleneck when there is too much I/O traffic for the available processing power.

Rather than partitioning the CPU cores of a single server, we propose to partition the servers of a data center rack; a model which we denote as Paravirtual Remote I/O (vRIO). We thus extend the concept of sidecores by adopting a wider view of the system. By looking at a rack as a logical unit, we can then give entire machines specialized tasks. A server that hosts virtual machines is called a VMhost and contains CPU cores called VMcores. Any machine that offloads its I/O is called an I0client. A specialized server that processes I/O for IOclients is called an IOhost, and is comprised solely of CPU cores that process the I/O called IOcores. By offloading the I/O processing to the IOhost machine, we effectively split the hypervisor into two components: the local hypervisor which is responsible for hosting the virtual machines on the VMhost, and the I/O hypervisor which processes the I/O for all of the IOclients. IOclients communicate with its IOhost using a dedicated channel. vRIO leverages PCI passthrough to realize such a channel.

We have implemented vRIO as a set of drivers on Linux, and evaluate its performance as compared to the currently accepted paravirtual I/O model, as well as a representative of the published state-of-the-art called ELVIS [16]. Our main contribution is a new distributed paravirtual I/O model which is capable of offloading the I/O processing to a remote server. We concentrate the I/O processing of multiple servers of a rack in a single remote server thereby improving CPU utilization of the whole system.

An additional advantage is that vRIO enables consolidating of physical devices. For example, makes remote SSDs efficiently accessible to local VMs via the paravirtual I/O interface. Note that such access is inherently different than the access modes provided when sharing a Network-Attached Storage (NAS) device. If the VM can use the NAS device directly, then we lose the benefits of interposition. And if the NAS device is exposed to the guest as a traditional paravirtual device, then the associated overheads of traditional paravirtualization come into play yet again. Additional benefit, vRIO enables using a single IOhost to serve VMhosts using different local hypervisors, which allows 3rd party software vendors to implement a single solution (e.g SDN) that integrates seamlessly with all the hypervisors. These additional benefits are not necessarily unique to vRIO, but vRIO facilitates their integration. They are explained in section 5.

Our results show that the offloading increases I/O traffic latency by 8µs as compared to state-of-the-art paravirtualization, but is often insignificant since it is a relatively small component of the overall latency. In some cases with extremely fast block devices, the overall latency can be doubled, compared to a local block device. On the other hand, by consolidating IOcores we demonstrate a reduction in CPU utilization while maintaining the same throughput levels, or increase throughput levels while maintaining CPU utilization.
Chapter 2

Related Work

The I/O interfaces between a host platform and a guest virtual machine take one of three forms: either the hypervisor provides the guest with emulation of hardware devices, or the hypervisor provides virtual I/O drivers, or the hypervisor assigns a selected subset of the host’s real I/O devices directly to the guest. Each method has advantages and disadvantages.

Emulation means that the host emulates a device that the guest already has a driver for. The host traps all device accesses and converts them to operations on a real, possibly different, device. This approach requires many world switches (i.e. context switch between guest VM and host hypervisor) and has relatively low I/O performance [30]. On the other hand, no changes are required to the guest OS. Emulation is the default mode of I/O virtualization in all current x86-based virtualization offerings.

Paravirtualization [6] was initially used to overcome obstacles to virtualizing the x86 architecture without the need for deeper software techniques such as binary translation [1]. Specifically, if a paravirtualized guest kernel avoids all use of non-trapping privileged instructions such as POPF, it can run on a trap-and-emulate VMM. Additionally, paravirtualization was used to improve performance using a batched hypercall interface to amortize the guest/VMM crossing overhead.

Paravirtualization has also been used to accelerate virtual machine I/O. For example, instead of giving the guest operating system a virtual device that behaves exactly like an existing physical device (i.e. emulation), the VMM may implement a special-purpose NIC that has been developed specifically for the virtualized environment. Such a paravirtualized NIC side-steps the complexity/performance trade-off that a “real” NIC implemented in silicon must obey. It can present an interface to the guest that allows packets to be sent using fewer device “touches” than would be the case for common physical NICs. KVM’s virtio device architecture, for example, uses a shared memory area to allow the guest’s network driver to communicate asynchronously with the hypervisor [29].

While device paravirtualization delivers performance gains, it requires that the hypervisor vendor provides not just a VMM but also guest drivers for the paravirtualized
device paravirtualization realizes performance benefits only in exchange for taking on the burden of writing both device emulation code and guest drivers, and probably for multiple guest operating systems. Device paravirtualization reduces the number of exits required to complete an I/O transaction against a standard device. vIC [2] presents the design and implementation of a virtual interrupt coalescing scheme for virtual SCSI hardware controllers in a hypervisor. Dong et al. [10] use virtual interrupt coalescing via polling in the guest and receive side scaling to reduce network overhead in a paravirtual environment. Rather than polling for fixed intervals or according to arrival rates, QAPolling uses the system state as determined by applications’ receive queues [8].

Direct device assignment (interchangeably referred to as direct device access”, direct access” or pass-through access”) means that the guest sees a real device and interacts with it directly, without a software intermediary. This approach improves performance with respect to para-virtualization since no host involvement is required. Additionally, no guest modifications are necessary, and the guest can use any device it has a device driver for. In a direct access scenario, it is the guest, not the host, which should handle an interrupt, but delivering an interrupt directly to the guest is not feasible in the general case: the guest might not even be running at the time a device raised the interrupt. There are various approaches and implementations to address the latter scenario. Gordon et al. [15] describe how a hypervisor can carefully manage physical resources, such as the Interrupt Descriptor Table (IDT) and the APIC, to further eliminate most interrupt-related exits. vRIO uses the latter software-based approach to eliminate the interrupt cost incurred by virtualization.

Several studies have used the sidecore approach to offload I/O operations to a dedicated core. vTurbo [36] accelerates I/O processing for VMs by offloading their associated IRQs to a designated core. Scheduling for this designated core is adjusted to achieve timely processing of IRQs. The scheduler on this core is adjusted to achieve timely processing of the IRQs. Virtualization polling engine(VPE) [23] uses dedicated CPU cores to help with the virtualization of I/O devices by using an event-driven execution model with dedicated polling threads. vIOMMU [4] exposes an emulated IOMMU (vIOMMU) to the unmodified guest. By running the vIOMMU on a sidecore, the throughput of the unmodified guest was tripled. While these works offload I/O activity to a sidecore of a single machine (the machine that hosts the VMs), vRIO performs the paravirtual I/O operations in a dedicated core of a remote host.

Tu et al. [33] present a direct device sharing system among multiple virtual machines in a PCIe-based cluster, which enables I/O devices to be shared among virtual machines running on multiple hosts. This architecture could serve as the underlying infrastructure of vRIO instead of the currently used Ethernet channel, given some minor modifications to the vRIO modules. Therefore, this work to be complementary to vRIO.

vRIO is aligned with recent trends in the industry, where products such as the Oracle Fabric Interconnect [26] have started to emerge. Oracle Fabric Interconnect (formerly
Xsigo Systems’ I/O Director) is a hardware and software device that consolidates data center I/O devices. There are no academic publications about the issue, and very few details are available. The main focus of Oracle fabric interconnect is on management of IT infrastructure, like directing I/O traffic from both SAN and LAN, or easily connecting a new I/O device. vRIO focuses on the performance aspects of the I/O path, namely the virtual I/O path in the new consolidated structure, as described in details in the following sections.

New adapters with VXLAN/NVGRE support have been introduced [25], allowing for these functionalities to be executed by the adapter rather than by the CPU. Like vRIO, these cards can interpose on the guest’s I/O. However, they provide limited functionality, while vRIO can run any linux software to inspect or manipulate I/O traffic. Moreover, such a device needs to be installed for each physical host and can’t be shared among multiple hosts, whereas vRIO provides I/O and service consolidation. Despite the differences, the two approaches can complement each other. The application specific hardware can be installed on the IOhost to perform the required functionality. This gives the benefits of both approaches: centralized I/O control and hardware offloading.
Chapter 3

Design

In the traditional paravirtual I/O model, the hypervisor is involved in the I/O path of each virtual machine. It runs a CPU-consum ing software component to process the I/O requests of all the VMs running in the same server. vRIO, motivated by rack-scale computing and emerging high speed, low-latency interconnects, proposes to offload this software component and the IOcores it uses from all the VMhosts located in the same rack to a separate, dedicated machine. By dedicating one machine to the task of virtual I/O processing, we can then free up resources on the VMhosts which can be used to run more VMs. In effect, we are consolidating the virtual I/O processing in order to introduce machine specialization at the rack-scale. In the same way storage appliances demonstrated that consolidation of storage resources in a remote appliance is beneficial, in this paper we demonstrate that consolidation of IOcores in the IOhost is also beneficial.

vRIO can be connected to any existent storage appliance, network switch, storage software or network software in the same way hypervisors are connected today. The goal of vRIO is to provide a more efficient, more scalable and hypervisor-agnostic way to virtualize I/O devices (e.g. vNICs, vHBAs or virtual disks) for the VMs running in an imbalanced rack environment which can be connected to existing storage or network solutions.

A single vRIO IOhost can serve VMhosts using different hypervisors, which allows 3rd party software vendors to implement a single solution that integrates seamlessly with all hypervisors rather than dealing with the complexity of using the different integration frameworks provided by each hypervisor. For example, vRIO can expose vNICs to KVM, VMWare and Hyper-V that shares the same L2 virtual network. This L2 network can be enhanced with firewall or intrusion detection software. In a traditional environment, 3rd party vendors would be forced to invest a lot of effort to integrate a single solution with the APIs of each hypervisor. With vRIO, the I/O interposition framework is shared across all the VMhosts and a single software code-base is required to support multiple hypervisors. This model also applies to other types of solutions such as PCIe flash or SSD based caching for storage. In addition, PCIe flash devices, SSDs and
Figure 3.1: vRIO architecture — The system is divided into IOclient (front-end) and IOhost (back-end).

Hardware accelerators for compression or encryption can all be consolidated in the IOhost to improve utilization by sharing the devices across many servers/hypervisors and to improve performance.

Figure 3.1 illustrates the overall design of the vRIO system, showing the IOclient (virtio front-end) and IOhost (virtio back-end). There can be many IOclients communicating with a single IOhost, but only one is shown here for clarity.

3.1 Paravirtual Protocol

Using paravirtual I/O means that the guest has a special driver installed that exposes the paravirtual devices. The driver design is described in more detail in the "driver model" section below. In our design, we could have selected any one of the many paravirtual protocols that currently exist, but we decided on the virtio [29] protocol which is currently used by KVM. Our choice of protocol is quite isolated from the rest of the system, and does not impact components other than the paravirtual drivers that we provide for the front-end and back-end.

In a conventional implementation of virtio, the guest and the hypervisor communicate by sending I/O requests and responses over a ring buffer stored in shared memory called a virtqueue. In vRIO, we replaced the virtqueue with a network link, which serves as a dedicated communications channel between the guest and the hypervisor. The sole purpose of the network link is to carry virtio I/O requests and responses, and cannot
be used for any other purpose.

### 3.2 Network Link

vRIO is inherently slower than a single machine solution because we are replacing communication over a shared memory buffer (local communication) with a network link (remote communication). Consequently, a large part of our effort was to minimize the negative effects incurred by distancing the I/O processing from the VMhost.

To implement the dedicated communications channel between the guests and IOhost, we use a network card that supports SRIOV. SRIOV allows us to use device assignment, which helps keep latency to a minimum as we don’t require any involvement from the local hypervisor in order to support this link. This decision holds the additional benefit of being hypervisor agnostic, since each guest is responsible for managing the I/O hardware directly. That means each guest could potentially be hosted by a different hypervisor, but still communicate over the dedicated channel as long as the local hypervisor supports device assignment of SRIOV devices. As mentioned earlier, the use of device assignment is not free of disadvantages; by selecting SRIOV, we lose the ability to migrate VMs, at least without additional software support [17, 27, 31, 37]. vRIO can work on top of any interconnect that supports SRIOV such as Ethernet or Infiniband. The only constraint on the interconnect is that it would have sufficient bandwidth to support the I/O activity of the virtual devices.

vRIO allows for the use of a paravirtual device in place of an SRIOV virtual function as the communications channel. This may be required if SRIOV cards are not available. Paravirtual devices add latency to the overall communications, as well as CPU overhead by forcing the hypervisor to be active on the data path.

Looking towards the future, we are consoled by the fact that NIC speeds are increasing, whereas CPU core speeds are remaining relatively constant. This means that the relative latency of the network link will decrease as faster NICs enter the market, thus improving the performance of such a setup.

### 3.3 vRIO Driver Model

Both the front-end (IOclient) and the back-end (IOhost) require new drivers in order to implement vRIO. The front-end contains the drivers which expose the paravirtual devices to the guest. The back-end contains the drivers which process the requests, and communicate with the backing devices. The following paragraphs describe the design and organization of these drivers.

**IOclient** Each type of paravirtual device (i.e. network device, block device) has its own separate driver. These drivers call functions in the generic driver, which contains common functions needed by the paravirtual devices.
The main purpose of the generic driver is to receive commands from the I/O hypervisor. This includes commands to create and destroy virtual devices. The generic driver gets the name of the physical device that is to be used for the communications channel when it is loaded. It then passes this device name to the transport driver, which opens the specified device.

The transport driver encapsulates all I/O requests generated by the paravirtual devices with the Ethernet protocol, and sends the requests to the I/O hypervisor. The driver also performs the opposite action for incoming responses by stripping the Ethernet-specific information, and calling the handler function in the associated paravirtual device driver. The transport driver is tightly coupled with the underlying physical device, and is responsible for fragmenting and assembling packets that exceed the maximum packet size (MTU) defined by the dedicated communication channel.

**IOhost** The generic module sends commands to the IOclients, instructing them to create or destroy virtual devices. The transport driver acts in the same way as in the IOclient, and is responsible for packet assembly and fragmentation. Once the incoming packet is reassembled, it is passed up to the appropriate driver handling the backing device for further processing. There is a 1-to-1 relationship between virtual devices in the IOclients, and instances of the backing devices in the IOhost.

### 3.4 I/O Hypervisor

Figure 3.2 shows how different IOclients access a range of consolidated services and devices on the IOhost. Since the IOhost has the ability to interpose on each I/O request, it may also run additional services such as block or packet level encryption, SDN, deep packet inspection, intrusion detection, anti-virus, etc. The services would be run on the
IOhost, but would act on all of the I/O from the various VMhosts. Some software could take advantage of this centralized installation by having an overview of the entire system at its disposal. This wider view would enable features such as data de-duplication and compression to be executed on a larger scale, potentially with more effect.

**Polling-mode drivers** In accordance to the sidecore approach, IOhosts poll their Network Interface Controllers (NICs). Thus, with polling at the IOhost and exitless interrupts [15] at the VMhost, vRIO removes the interrupt processing overhead from the virtualization layer.

**Platform independence** The IOhost can support IOclients from different hardware architectures, including those with different endianess (i.e., big-endian, little-endian). For example, we connected an x86-based IOhost that uses a little-endian architecture, to a POWER7 which uses a big-endian architecture. Since the I/O data is opaque to the transport and driver layers, only the vRIO packet headers had to be placed into network byte order to ensure interoperability.

### 3.5 vRIO Challenges

While designing this system, we were constrained by several requirements. We wished to be able to share I/O devices among multiple hosts, but not incur additional latency. We wished to keep the benefits of virtualized I/O, and not lose them as is the case with SRIOV. *Live migration* is also a feature we would like to keep. We have designed vRIO to keep these benefits of paravirtual I/O, while at the same time providing performance that is sometimes comparable to an SRIOV device.

**Latency** By adding the IOhost to the system, another link was added to the I/O path. We then faced the challenge of dealing with the additional latency incurred by this extra hop. After experimenting with the system, we discovered that the major contributor to the additional latency was not the dedicated communications channel as we originally thought, but rather handling the interrupts on the IOhost. To explain what happened, we will generalize a bit about the behavior of I/O devices. In a single machine system, an I/O device will generate one interrupt when the request is sent to the device, and another interrupt when the response is received from the device (interrupt coalescing notwithstanding). Under our model however, the situation is complicated by the addition of the dedicated communications channel between the IOclient and the IOhost. In the transmit path, one interrupt is generated by the dedicated NIC when the request is received from the IOclient, and another interrupt is generated when the request is sent to the physical backing device, for a total of two interrupts per I/O request. It then follows that if the number of interrupts per request is doubled (from one to two) then the amount of CPU time to process these interrupts would also be doubled. Indeed, our experiments revealed that a large portion of the CPU time on the IOhost was being spent just on handling interrupts. By moving to a polling model on the IOhost and using polling mode drivers on both of the physical interfaces, we were
able to reduce the latency to be comparable to single-host paravirtual solution.

**Live Migration** Live migration with SRIOV is a difficult problem. But it can be solved so long as a paravirtual agent can decouple its association to the guest [17, 27, 31, 37], which is the case for vRIO. Live VM Migration starts with notifying the agent that migration is to take place, and that it is to decouple itself from the underlying physical device. The indirection layer abstracts the driver of the dedicated communications channel, allowing the VM to couple with a new driver after migration is completed. It is possible to couple with a driver managing hardware of a different kind or vendor, as long as the abstraction remains the same. This of course would require the correct drivers to be present on the guest system to be able to bind to the new hardware after migration.

**Fault Tolerance** We do not currently have a fault-tolerant system built, but we feel it is important to include a short discussion about how such a feature might look. It is cost-effective to build a system using only a single IOhost, but this incurs the risk of the IOhost failing at some point. Since the IOhost carries responsibility for many hosts, this single point of failure could cause a serious loss of service if it were to fail. To mitigate this risk, it is possible to build redundancy into the system by running multiple IOhosts for a given set of hosts. If for example, we were to run two IOhosts where both are always active, each could act as the backup for the other. In the case of a failure of one of the IOhosts, all I/O traffic could be redirected on-the-fly to the other one. To enable this behavior, each VMhost would have to be programmed with the addresses of both IOhosts, and be able to select between them (similar to how DNS servers are configured for hosts today). Each IOhost would also need to be aware of all of the VMhosts, and support all of the virtual devices for each guest. When a failure occurs, it would be the responsibility of the IOclient to detect the failure and to automatically switch over to the backup IOhost. In this case, the IOhost would then be responsible to handle the I/O traffic of all of the VMhosts, which would incur a performance penalty, but would avoid loss of connectivity to several hosts.
Chapter 4

Implementation

We have implemented vRIO on top of Linux 3.9 as a set of kernel modules based on vhost [32] and virtio code. We chose Linux because it is an open-source operating system which is widely used to host many I/O services such as network storage, software bridges and firewalls. Thus, Linux makes it easy to integrate existing services with vRIO.

Considering the time it takes for the IOcore to process the I/O requests (software) is longer than the time it takes to transmit the I/O requests from the VMhost to the IOhost over a fast interconnect (hardware), we optimized vRIO’s code to minimize latency. In this section we describe all the different optimizations we applied to minimize vRIO’s latency.

**MTU**  To improve the performance of the communications channel, we use jumbo Ethernet frames, up to 8100 bytes in size. The NICs of our IOhost and VMhost are connected via fiber optics, which supports the Data Center Bridging (DCB) [9] extension to the Ethernet protocol, providing lossless Ethernet. We leverage this capability in our vRIO prototype instead of implementing flow control at a higher layer.

**Zero Copy**  vRIO drivers use zero-copy buffers in both the IOclient and the IOhost. When the IOclient initiates a request over a virtual network device, the guest’s network stack allocates a socket buffer. This buffer is used directly by vRIO, which saves the effort of creating a new one and managing the referenced buffers. The handling of a block device request is slightly different. Since it is not a native Ethernet request, vRIO driver must create a socket buffer for it. However, to avoid costly copy operations, the data inside the request is referenced without being copied. Block data must be aligned to sector size, and cannot cross a memory page boundary. This is challenging since each buffer is preceded by the vRIO headers.

The virtio request size can vary and can exceed the MTU. Thus, vRIO has to segment the buffers and reassemble them upon arrival. To perform this operation efficiently, vRIO leverages the TCP segmentation offload (TSO) supported by modern NICs. vRIO does not use TCP but by adding fake TCP and IP headers to the L2 frames, vRIO offloads the segmentation task to the NIC.
Chapter 5

Benefits

I/O Core Consolidation Sidecore approach [16, 19] has been shown to improve I/O performance in virtualized environments by 1.2x-3x [16]. In this approach, a VMhost has a set of CPU cores that is dedicated for processing I/O, and cannot be used for any other purpose. In vRIO, we free these cores on the VMhosts by moving the I/O processing to the IOhost instead. This means there are now more CPU resources available on the VMhosts for hosting VMs. The cost is that we must give up one server from the rack to become the IOhost, and service the VMhosts. The aim is to provide services for an entire data center rack, rather than each server servicing itself. In the most simple case, the ratio of IOcores to VMcores remains constant, as well as the total number of physical machines. Let us take the example of a rack with 8 servers, each with 8 CPU cores. With the existing sidecore approach, each server would have a single, dedicated IOcore, with the ratio of 7:1, for a total of 56 VMcores, and 8 IOcores. Under vRIO, we dedicate an entire server to process the I/O of all the other servers. There are still 8 IOcores, but consolidated in a single IOhost, serving 7 VMhosts (with 56 VMcores). We show later in the "Load Imbalance" experiment that this configuration has the benefit of being able to handle larger I/O loads on a single server than were previously possible by using all of the available processing power in the IOhost.

I/O Device Consolidation Servers often include specialized hardware to perform specific tasks that require more performance. A solid-state drive (SSD) may be installed to provide a speed boost on a boot device or another performance critical application such as a database, or perhaps a graphics processor unit (GPU) is needed for speedy calculations that are best to be offloaded from the main CPU. Purchasing such specialized hardware for many servers can be expensive, and may not be able to be utilized to its full capacity. Device consolidation is not a novel feature introduced by vRIO, but is an inherent feature in the design. By virtue of the IOhost being a central location accessible by all VMhosts, it provides a convenient location in which to consolidate the devices. By providing equal opportunity to access the consolidated devices, it allows the devices to be shared among all IOclients, improving the utilization. Consolidation allows us to reallocate resources in a more equal fashion by enabling load balancing at
runtime as loads change between IOclients. IT departments need to provision fewer I/O devices, and would be able to provide access for IOclients that only need them occasionally. Under static allocation, it is likely that casual users would not have access to devices such as a specialized accelerator.

**Bare-metal Virtualization** There are no dependencies between the local hypervisor and the I/O hypervisor components, as long as the drivers on both sides implement the protocol correctly. This fact presents an interesting side-effect of making the I/O hypervisor completely agnostic of the local hypervisor. In fact, we don’t really need the local hypervisor at all, and vRIO can support paravirtual devices in a bare metal IOclient. By installing the same paravirtual drivers on a bare metal system, we can communicate with the I/O hypervisor and support all of the same features that are supported in a fully virtualized environment. This capability enables bare-metal clouds to provide software-based functionality, such as a firewall or an anti-virus, by interposing on the I/O traffic of bare-metal workloads. The software can be hosted by the IOhost without stealing CPU cycles from the target bare-metal machine. In addition, any software running on the IOhost cannot be directly detected nor disabled (such as a virus would wish to do to anti-virus software).
Chapter 6

Evaluation

We implement vRIO as described above and evaluate its performance, strengthens, and weaknesses using micro- and macro-benchmarks.

6.1 Experimental Methodology

**Physical Servers** Our testbed system is comprised of four physical machines: two VMhosts, one IOhost, and one load generator, but not all machines are used in all experiments. Each VMhost is an IBM System x3550 M4, equipped with two 8-cores sockets of Intel Xeon E5-2660 CPU running at 2.2 GHz, 56GB of memory and an Intel x520 dual port 10Gbps SRIOV NIC (allowing a total throughput of 20Gbps per VMhost). The VMhosts’ hypervisors are KVM with QEMU 1.3.0. The VMhosts runs virtual machines allocated with one VCPU and 1GB of memory each. The memory is backed using huge pages of 2MB size to maximize performance. The IOhost is an IBM System x3650 M4 with two 8-core sockets of Intel Xeon E5-2680 series CPU running at 2.7 GHz, 128GB of memory and two Intel x520 dual port 10Gbps SRIOV NICs (with a total throughput of 40Gbps). The load generator is an IBM System x3550 M2, equipped with two quad-core sockets of Intel Xeon 5500 series CPU running at 2.93 GHz, 12GB of memory and an Emulex OneConnect dual port 10Gbps NIC.

All machines run Linux 3.9 as their kernel: guests, hosts, and bare metal. In order to obtain consistent results and to avoid reporting artifacts caused by nondeterministic events, hyperthreading and all power optimizations are turned off, namely, sleep states (C-states) and DVFS (dynamic voltage and frequency scaling).

**Virtual I/O Models** We evaluate, contrast, and compare vRIO against three models of virtual I/O: KVM/virtio [29], ELVIS [16], and SRIOV with exitless interrupts [15] (ELI). In KVM/virtio, a traditional paravirtual I/O device is installed in each guest. The hypervisor traps VM requests and injects virtual interrupts to VMs as required, such that both types of events trigger exits. There is no affinity between activities and cores, namely, interrupts of the physical I/O device, I/O threads of the hypervisor, and VCPUs. Although many studies of paravirtual I/O show improvement beyond this
canonical model of paravirtual I/O, it constitutes the de-facto standard as used in most real-life installations today. KVM/virtio therefore represents in our experiments the state-of-the-practice of paravirtual I/O. It is denoted in our graphs as the baseline.

ELVIS improves upon stock virtio by employing three optimizations: sidecores that process paravirtual I/O requests and thus avoid the associated exits; exitless interrupts that notify VMs about previous I/O requests by injecting the guest with a virtual interrupt without inducing exits to the hypervisor; and adaptive, fine grained scheduling of the paravirtual I/O, which identifies if traffic is latency- or throughput-sensitive and optimizes accordingly. In our experiments, we make use of one socket of each VMhost; one of the eight cores of this socket is designated to be the IOcore. The physical interrupts associated with paravirtual devices are exclusively handled by this IOcore, which then forwards them to the appropriate VMs using exitless IPIs (inter-processor interrupts). The ELVIS code is available from [14]. It represents the state-of-the-art of paravirtual I/O in our experiments.

SRIOV with exitless interrupts is the highest performance virtual I/O configuration we evaluate. It is only applicable for experiments that involve a NIC. Each VM is directly assigned one virtual instance of the physical NIC via SRIOV, and it thus communicates its I/O requests to the NIC directly. The NIC, in turn, sends its interrupts directly to the VM via exitless interrupts, removing any hypervisor involvement from the virtual I/O path and thus any associated virtualization overhead. Doing away with hypervisor involvement, this configuration inherently cannot support I/O interposition. It consequently serves as the theoretical upper bound for virtual I/O models that do support interposition. Accordingly, it is denoted in our graphs as the optimum.

Networking Setup Unless stated otherwise, the networking topology setup we use in our experiments is depicted in Figure 6.1. It involves three physical machines: load generator, VMhost, and IOhost. Configurations other than vRIO do not make use of the IOhost. The load generator is directly connected via its two 10Gbps ports to the IOhost and the VMhost, respectively, with a Maximum Transmission Unit (MTU) of 1500 bytes on each of the two links. In the baseline and ELVIS configurations, for optimal performance, we use the macvtap Linux device to bridge the VMhost’s physical NIC with the backend of the paravirtual NICs of the guests that run on the VMhost.

In the vRIO I/O path, the VMhost is connected to the IOhost using its second 10GbE port (the first port is connected to the load generator). Guests in the vRIO configuration are assigned with an SRIOV instance of the NIC, which is connected to the IOhost. Similarly to the baseline and ELVIS configurations, we use macvtap to bridge the IOhost’s physical NIC with the backend of the paravirtual NICs of the IOclients.

Measurements Each experiment is executed 5 times. We ignore the first run, make sure the variance across the remaining 4 is negligible, and present their average.
Figure 6.1: In vRIO, the workload generator is connect to the VMhost via the IOhost, whereas in the the three competing I/O models the generator and the VMhost are connected directly.

6.2 Worst case performance

We evaluate the performance of vRIO as compared to the other virtual I/O models in the least favorable conditions with respect to vRIO, namely, when there is no I/O interposition activity involved. As interposition activity increases, the relative weight of vRIO overhead is reduced. For example, if interposition takes 100µs of per-packet processing time and vRIO induces an additional 10µs latency, then performance drops by less than 10%. On the other hand, if interposition takes only 10µs, than performance might drop as much as a factor of 2.

Network latency Our first experiment focuses on the most worrying consequence of using vRIO—its impact on latency. To this end, we use the popular benchmark Netperf UDP RR (request-response), which models a latency sensitive workload by repeatedly sending a UDP packet with a payload of only one byte and waiting for a matching byte response. The results are depicted in Figure 6.2, showing the average request latency as a function of the number of VMs in the VMhost that run the benchmark.

Let us first explain the connection between the number of VMs that partake in each experiment and the number of cores that we assign to that experiment. Let $N$ denote the number of VMs (displayed along the X axis). Given a specific $N$, we set the number of cores in that experiment to be $N + 1$ for the baseline, ELVIS, and vRIO configurations. For ELVIS, the extra core resides in the VMhost and serves as the IOcore. For vRIO, the extra core serves the same purpose, but it resides on the IOhost. For the baseline, the extra core resides in the VMhost like in the ELVIS case, and the Linux scheduler decides on which core to run each of the VCPU and I/O threads. Interrupts generated by the NICs were balanced across the cores in use. We assign only $N$ cores to the optimum, 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 and vary the number of VMs.)
Examining Figure 6.2, we see the optimum configuration enjoys nearly perfect scalability, with 30–32\(\mu\)s latency on average per request-response. The results show a similar slope for vRIO, meaning vRIO scales similarly. But absolute numbers show that vRIO latency is about 12\(\mu\) higher—the price of the added hop.

Comparing vRIO to the optimum is not “fair”; the real competitor of vRIO is ELVIS, because the two allow for interposition whereas the optimum does not. Initially, when \(N = 1\) (only one VM) and the system is underloaded, the latency of vRIO is 8\(\mu\)s longer. This gap pertains to the 18% prolonged latency relative to state-of-the-art paravirtualization that we report in the abstract of this paper. As for network workloads, it is the worst-case performance of vRIO relative to ELVIS that we have observed. As the load increases and more VMs are added, the gap between the two shrinks, until vRIO becomes faster at \(N = 6\).

The underlying reason as to why vRIO outperforms ELVIS under higher loads rests on how they differ in handling interrupts coming from the I/O device. ELVIS allows such interrupts to occur in the regular way, and it handles them as they fire. It utilizes its IOcore exclusively to poll for VM requests. Conversely, vRIO eliminates physical interrupts altogether by polling the I/O device. We experimentally demonstrate this realization later on in a subsequent paragraph. We believe, however, that this deficiency is not inherent to the ELVIS design, and that ELVIS could be modified to be more scalable by utilizing its IOcore to poll the device in addition to its VMs.

Let us return to comparing vRIO to the optimum, in more detail. A deeper look into the difference between the latencies reveals that it is in fact steadily growing with \(N\), from about 12\(\mu\)s to 13\(\mu\)s, as depicted in Figure 6.3 (left Y axis). We find that this increase is a result of contention over the remote IOcore of vRIO. This finding is exemplified by Figure 6.3 (right Y axis), which shows the fraction of traversing packets (from both directions) that had to wait before being processed by the IOcore, which is tightly correlated to the increase in the depicted latency gap.
Network throughput  Our next set of experiments evaluates the impact of vRIO on throughput oriented applications. We use the Netperf TCP stream for this purpose, which measures network performance by maximizing the amount of data sent over a single TCP connection, simulating an I/O-intensive workload. We set the packet size to be 64 bytes so as to stress the vRIO model and highlight the overheads it incurs. Bigger packet sizes quickly saturate the network links which reach their maximal bandwidth, hiding the differences between the various I/O models.

The results are shown in Figure 6.4. All four I/O models demonstrate linear scaling, with ELVIS and the optimum achieving nearly identical performance. vRIO is a close second, with a 5–8% lower throughput across the board. Analyzing the reason for this performance penalty, we find that it is due to the added processing time incurred by the paravirtual vRIO driver, which is not present in ELVIS and the optimum. We quantify this added compute time in Figure 6.5. The figure depicts the average per-
packet processing time in the corresponding $N = 1$ configurations. Since vRIO spends an average of 9% more cycles on each packet, it is unable to reach the same level of throughput.

To conclude the throughput analysis, we run an additional experiment where we assign an equal number of cores to each I/O model ($N = 8$), which equates to all cores in one VMHost socket, except for vRIO, which uses 7 VMcores plus one remote IOcore. The outcome is shown in Figure 6.6. The new result is associated with the 8 VMs optimum bar on the left; the rest already appear in Figure 6.4 ($N = 7$). As expected, utilizing 8 cores and 8 VMs in the optimum (SRIOV) configuration yields superior throughput. This is the price we pay for having the ability to interpose on the I/O of the VMs.

**Macrobenchmarks** To evaluate the performance of vRIO on real applications, we use Apache [12] and Memcached [13] to this end. Apache is a popular web server.
Figure 6.7: Performance of Apache and Memcached under the four I/O virtualization models.

We drive it using the ApacheBench [5] (also called “ab”) which is distributed with Apache. It assesses the number of concurrent requests per second that the web server is capable of handling. We use 2 concurrent requests per VM for a 4KB file.

Memcached is a high-performance in-memory key-value storage server. It is used by web applications for caching results of slow database queries, thus improving overall performance and scalability. We used the Memslap benchmark [3] (part of the libmemcached client library), which sends a random sequence of memcached get (90%) and set (10%) requests to the server and measures the request completion rate. We use 8 concurrent requests per VM.

Figure 6.7 shows the results. For Apache, vRIO approaches the optimum whereas ELVIS falls behind. For Memcached, vRIO is noticeably less performant than the optimum but is still better than ELVIS. As we explained in the case of latency (Figure 6.2), ELVIS is inferior to vRIO because it needs to handle the physical interrupts; vRIO polls the device and in so doing avoids the associated overheads. We now experimentally demonstrate that this explanation is correct by additionally running the Apache benchmark with our non-polling version of vRIO. Figure 6.8 displays the results. We see that the non-polling version yields worse performance than ELVIS, highlighting the fact that interrupts are to blame.

The non-polling vRIO is inferior to ELVIS largely because it must handle twice as many interrupts. The difference arises because an interrupt is generated on both the transport interface and the physical backing interface for each transaction, effectively doubling the number of interrupts that must be served by the IOhost. Even with features such as interrupt coalescing enabled, the overwhelming number of interrupts quickly becomes a limiting factor, reducing throughput and increasing latency at the IOhost. This understanding served as a main motivator for our polling implementation of vRIO.
Figure 6.8: Turning off polling in vRIO and enabling interrupt instead dramatically reduces the performance of ApacheBench.

6.3 Benefits of consolidation

Block workloads Assume that we want to accelerate the storage performance of the VMs in our workload using a fast I/O block device. In this paragraph we evaluate the performance penalty of placing this device in the IOhost instead of in the VMhost. To eliminate the bottleneck created by a physical drive and allow the VMs to achieve their maximum throughput we simulate the device using ramdisk. We expose a 1GB device to each VM, we run within the VM read and write threads performing 4KB random I/O using Filebench benchmark [24] (version 1.4.9.1). A reader/writer pair consists of a writer thread and a reader thread. We use one reader, one pair and two pairs in each VM. To avoid benchmarking the guests’ filesystem cache, we open the virtual block device with the O_DIRECT flag, so that all I/O requests would pass the guest-host boundary. In this setup, we test only vRIO, ELVIS, and baseline; we do not test the optimum because there is no such thing as a SRIOV ramdisk.

We estimate the latency of accessing a remote block device by measuring performance of a single reader (similar results apply for one writer). The Figure 6.9a shows the throughput achieved by each of the 3 configurations. vRIO scales better than the baseline but worse than ELVIS. This result refers to the 2.2x prolonged latency that we report in the abstract of this paper. The results shown in Figure 6.9b–c are for one reader/writer pair and two reader/writer pairs, with one pair yielding better results in ELVIS and two pairs yielding better results in vRIO. The reason for the latter counterintuitive result turned out to be the number of involuntary context switches within ELVIS guests, which was up to two orders of magnitude more in comparison to vRIO. The relatively low latency which the local ramdisk provides combined with the relatively high number of CPU cycles required to process each ramdisk request caused the threads to contend over the CPU, which wasted a lot of cycles and caused
the throughput to drop.

We have utilized ramdisk to approximate the overhead incurred by vRIO on future faster I/O devices. When applied to SSDs that are currently available, the difference between the configurations drops to below 17% for one VM and 5% for 7 VMs. The results are shown in Figure 6.10.

**Improved Utilization**  A main benefit of vRIO is IOcore consolidation. We first demonstrate this benefit for a semi-intensive I/O workload, which does not require all the computational power that an IOcore provides. Different than the experimental setup used until now, this experiment consists of two VMhosts, each running five guest VMs, utilizing 5+1 cores (N=5). All VMs internally run the Webserver personality of Filebench, 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 configuration: (1) ELVIS with two IOcores, one per each VMhost; (2) vRIO with only a single “consolidated” IOcore situated in the IOhost and servicing both VMhosts; and (3) the baseline servicing the five VMs with six cores (N+1). In the baseline and ELVIS configurations, the VMs have a 1GB virtual block device backed by a ramdisk on the VMhost. In vRIO, the block device is backed by a ramdisk on the IOhost.

Figure 6.12a–b shows the CPU utilization of the two ELVIS IOcores in their respective VMhosts. Both cores are underutilized, collectively spending about 150% of the 200% CPU time available to the IOcores on polling without doing anything useful. Figure 6.12c shows the utilization of the single, consolidated vRIO IOcore. In this case, we see that only about 15% CPU cycles go to waste. Figure 6.11a shows the throughput achieved by the three configurations. vRIO is significantly better than the baseline but 14% below ELVIS. This outcome constitutes a typical consolidation tradeoff in virtual environments: sacrificing some performance (-14% throughput) to get significant savings in physical resources (one consolidated IOcore instead of two).

**Load Imbalance** Our second experiment demonstrates the performance benefit of consolidated resources under unbalanced load conditions. Assume we have at our
disposal two IOcores, and now we need to decide where to position them in our system. In the vRIO case, we consolidate both in the IOhost, whereas in the ELVIS case we partition them between the two VMhosts, one for each. To simulate imbalance, we assume that only one VMhost currently requires service (the Webserver personality of Filebench as in the previous experiment), while the other VMhost is idle or is engaged in an activity that requires little I/O. To increase the imbalance, we also assume that the Webserver makes use of I/O interposition for the sake of seamless encryption. We use AES-256 as the encryption algorithm and invoke it through standard Linux APIs. In vRIO, since the two IOcores are consolidated and thus accessible to all VMhosts, we are able to process the encryption with both IOcores. Conversely, in ELVIS, the VMhost can only make use of one, local IOcore. Figure 6.11b shows the outcome: vRIO provides a 82% improvement as compared to ELVIS. This performance boost is an inherent benefit of vRIO in an unbalanced workload.

6.4 Heterogeneity

In our final set of experiments, we demonstrate that vRIO enables I/O interposition in a way that is hypervisor agnostic, keeping the guest’s hypervisor off the I/O path. We run Netperf TCP stream experiments (16KB per message) with a Linux IOclient as: (1) a guest hosted on VMwareESXi 5; (2) a guest hosted on KVM; and (3) a bare metal operating system. The results are shown in Figure 6.13b. All three configurations attain line rate on the network interface, and they have comparable CPU utilization on both the IOcore and VMcore. This experiment shows that the IOhost processes the I/O regardless of the underlying hypervisor.

We perform an additional test involving an IBMPOWER 710 (8231-E1C) with 64GB memory and 1Gbps Intel 82571EB Ethernet to show that vRIO is also hardware platform agnostic. We install the vRIO driver on the host operating system, 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 throughput. The results are shown in Figure 6.13a. Comparing the results holds little meaning since there are many variables involved. However, both servers attain line speed on the network link and have comparable CPU usage at under 10%. We conclude that it is possible to run vRIO services on the IOhost that serves multiple hardware platforms, unmodified and unaware of the IOclient’s hardware architecture.
Figure 6.13: Supporting different architectures and hypervisors in vRIO.
Chapter 7

Conclusions

The industry is moving towards higher density servers and rack-scale solutions to help cut costs in the data center. Economies of scale have come into play, and costs are now commonly calculated per-rack, rather than per-server [21, 28]. Since the rack is becoming the new server, our goal is to show that server specialization can make hardware utilization of a rack become more efficient. Given that sidecores are a prominent way to optimize virtual I/O, we show that it makes sense to consolidate these sidecores in a single server, rather than to spread them across the rack. Especially since new interconnects being introduced are of increasing higher speeds and lower latencies it reduces the relative cost of accessing the remote server.

We describe the implementation of the split hypervisor design, as well as the performance evaluation of the design, highlighting the tradeoffs between CPU utilization and I/O performance. Based on the reported results, we feel that this design is feasible to be implemented in a data center, and can improve the utilization of expensive hardware, allowing the density to be increased.

We find that the main concern is latency, but we propose and analyze some realistic techniques to keep latency to a minimum. We only evaluate one particular kind of interconnect, but better interconnects will be available in the future, which may make this idea even more appealing.

In the case of an unbalanced load, the split hypervisor design can improve throughput and increase I/O device utilization by allowing equal access to the hardware. So in a case where I/O processing would have been limited by the available hardware, vRIO can increase the limits through statistical multiplexing. By increasing the scale of the problem by consolidating all I/O processing for several machines, we note that in the average case, the likelihood of all VMs requiring peak performance simultaneously is low, and therefore we can reduce the amount of required hardware.
Bibliography


Single Root I/O Virtualization (SRIOV) is a technique that reduces the overhead of virtualization, by directly mapping virtual devices to physical devices, reducing the need for virtualization software. This allows for improved performance, as the virtual machine does not have to go through the hypervisor to access the device.

In this thesis, we focus on SRIOV and its impact on virtual machine performance. We propose a new design for virtual machine networks, which utilizes SRIOV to improve performance. Our design includes a new hypervisor architecture that provides better performance and scalability.

We also propose a new design for virtual machine networks, which utilizes SRIOV to improve performance. Our design includes a new hypervisor architecture that provides better performance and scalability.

The proposed design includes a new hypervisor architecture that provides better performance and scalability. The hypervisor is designed to be more efficient and scalable, allowing for better performance in virtual machine networks.

Our design includes a new hypervisor architecture that provides better performance and scalability. The hypervisor is designed to be more efficient and scalable, allowing for better performance in virtual machine networks.
Paravirtual I/O (vRIO) enables the host to influence the input/output operations of the guest machines, for sharing hardware resources among the VMs. Studies of the host cores intended for execution of virtual machines, the host cores themselves, and network interfaces indicate that taking this model a step further, we propose to have virtualized guest processors (VM hosts) or IO hosts that can host a virtual vRIO, which is referred to as a virtual host interface (VHIO). We have investigated the benefits and drawbacks of vRIO in this context of service-oriented computing (SOC) and have taken a model that is based on this virtual interface. We have successfully founded a virtual interface model for both the X86 and POWER servers that run on ESX and KVM hypervisors. The virtual interface is independent of both the hardware and the hypervisor, for example, for X86-64-based systems.

Reviewing the literature and existing studies, we have focused on the following aspects of vRIO: (1) improving hardware utilization by multiplexing requests from multiple virtual machines; (2) using the virtual interface for seamless or on-demand communication to physical or virtual devices; (3) virtualized input/output (I/O) interfaces for unobtrusive and efficient resource sharing; (4) flexible network communications; (5) integration of virtualization and software-defined networking (SDN); (6) mirroring and snapshots; (7) network security and detection.


tক্ষুদ্র

ক্ষুদ্রতর মানসিক লেখা উপরের ক্ষুদ্রতর মানসিক লেখা উপরের ক্ষুদ্রতর মানসিক লেখা উপরের ক্ষুদ্রতর মানসিক লেখা।

1. Shiferaw Getnet et al. (2015) show that the virtual input/output interface (vRIO) is not a good choice for multiplexing.

2. The method of handling I/O requests and maintaining network interfaces is efficient and low-cost.

3. The effect of different virtualization mechanisms on performance is studied, and a model is proposed that combines software-defined networking (SDN) and software-defined networking (SDN) for efficient and low-cost network virtualization.

4. This model is a step further in the field of virtualization and software-defined networking (SDN) for efficient and low-cost network virtualization.

5. The model is based on the virtualization of the network interface (NI) and the software-defined networking (SDN) for efficient and low-cost network virtualization.
המחקה מתאימה לשילוח פרסים. בенькלו להוות מחשבה.

תודה

מאגר נתונים ביצועי ולהבנת(QL) השילוח של, פרופ' ד"ר פרליר, על ברamateיה לכל אדם.

תודה. שיו שפתיאל השילמה זה איזו. ד"ר ברנאו לשילוח לאלא מורד. שבירי על את זה שלランקס.ランックス פנטן רוגר

עד ברנאו לשילוח לאלא מורד. שבירי על את זה שלランקס.ランックス פנטן רוגר

ערכו בין מחמתי העמוסות מחוץ לא ישראלpei עזר. בנסק. באית גם לזרות לאלים

חלף המחקה הממסרות של צי שולש על ממק גם ע mostra מילודת החממות לארוךーズ.

אני ơnיך המח להאו את הפינות העומדוכספים תנדバラ.

אגרזנו בהב. ברנאו השילוח לאלא מורד. שבירי על את זה השילוח לאלא מורד. שבירי על את זה השילוח לאלא מורד. שבירי על את זה השילוח לאלא מורד. שבירי על את זה השילוח לאלא מורד.

בנד.
קלט/פלט ירח-וייטסיאלי מודחק

היבר על מחקר

לשם מחקר חלקי על תודריך ירח-וייטסיאלי
מניסחי לחידוטו במדע המחשב

יוסי קופמן

רומח לס RECEIVER - مركز טכנולוגיה לישראלי
שטר מודעות - חיפה - יוני 2014
קולט/פלט פרמא-壓縮및 מרורק

יוסי קופרמן