Towards Exitless and Efficient Paravirtual I/O

Abel Gordon   Nadav Har'El   Alex Landau   Muli Ben-Yehuda   Avishay Traeger
IBM Research – Haifa
{abelg,nyh,lalex,muli,avishay}@il.ibm.com

Abstract

Virtualization is a prominent technology used in data centers around the world. While many kinds of workloads can run at near-native performance even when virtualized, I/O intensive workloads still suffer from high overhead precluding the use of virtualization in many applications. In this paper we tackle the problem of improving the performance of paravirtual I/O. We propose an exitless paravirtual I/O model, under which guests and the hypervisor, running on distinct cores, exchange exitless notifications instead of costly exit-based notifications. Our initial proof of concept improved throughput by 45% and latency by 25μsec compared to a traditional network paravirtual I/O model. We show that a single hypervisor I/O core can become saturated when serving multiple I/O intensive guests, and further research is required to improve scalability in this scenario.

1. Introduction

In recent years machine virtualization has taken on more and more roles in the modern computing environment, as virtualization hardware and software have been quickly advancing. Virtualization is used for server consolidation, for building huge and flexible compute clouds, and even for hosting users' desktops. The most important benefit of virtualization is flexibility: Virtualization decouples the guest operating-system from the physical hardware; It allows software to control the OS and its resources (CPU, memory, etc.) in a flexible way using a *hypervisor* and various management stacks.

While the main reason for using virtualization is flexibility, the main reason not to use it is the *virtualization overhead*. This overhead is defined as the amount of resources, especially CPU time, that a workload running on a virtual machine (VM) consumes beyond those consumed by the same workload on *bare metal* (a physical machine). While x86 virtualization has been available for many years, it gained momentum when new software [3, 6] and hardware [2, 5, 28] first allowed the performance of many useful guest workloads to achieve levels close to that of bare metal.

Achieving bare-metal performance (and not, say, "just" 20% below it) has become a holy grail of virtualization research. For compute-bound workloads, the above referenced approaches already achieved this goal of bare-metal performance. For workloads that made heavy use of the MMU (memory, paging, process context switching), further research on two-dimensional page tables and huge pages [10] was necessary before near bare-metal performance was achievable.

The biggest remaining virtualization-overhead problem is for I/O-bound workloads, e.g., workloads which are network- or disk-intensive. Three major techniques are common for hosts to virtualize I/O services for their guests: 1. *Emulation* [27], where a familiar device (e.g., a common network card) is emulated. 2. *Paravirtualization* [24, 27], which emulates a new device, designed not to resemble any existing device but to be as efficient as possible when used across the guest-host boundary. 3. *Device assignment* [12, 30], where the host gives a guest (mostly) direct access to a certain physical device.

The performance of these three techniques are in the above order, with device assignment being the best of the three [30]. The performance superiority of device assignment became even clearer when ELI [13], designed for device assignment, recently achieved the coveted bare-metal performance for I/O workloads. ELI removed most of the virtualization overhead by eliminating hypervisor involvement (a.k.a. *exits*) in handling of interrupts from the assigned device. With such Exit-Less Interrupts, the entire I/O critical path became exitless, and could therefore proceed at bare-metal speeds.

Despite the proven performance advantage of device assignment, in many use cases paravirtual I/O is preferred or even required. Device assignment cannot be used if the host wishes to offer a device with no physical counterpart (e.g., a virtual disk stored as a file in the host's filesystem), or if the host wishes to control the guest's use of the device (e.g., process every packet that goes in and out of the guest). Device assignment also requires more expensive hardware

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SYSTOR '12 June 4-6, Haifa, Israel
Copyright © 2012 ACM 978-1-4503-1448-0/12/06…$10.00

(an IOMMU [1] and, if the same device is to be assigned to several guests, SRIOV-supporting devices [11, 22]). Finally, device assignment also complicates VM migration [31] and host-side memory overcommitment [30]. For these and other reasons, most real-world applications of virtualization today choose to use paravirtual I/O.

In this short paper we propose a new technique for significantly boosting the performance of paravirtual I/O. Our proposed method was inspired by ELI [13] in that our goal was also to make the I/O critical path exitless.

In Section 2 we motivate our work and discuss related research. In Section 3 we outline our proposed model, which we call ELVIS (Exit-Less Virtual I/o System). As we shall explain, the techniques used in ELI are also useful for avoiding some of the exits associated with paravirtual I/O (namely, those related to host-to-guest notifications), while we propose new techniques to avoid the remaining exits, and move the host's work out of the guest's way, to a separate core, a sidecore [15, 16].

In Section 4 we describe our implementation of ELVIS, on the *KVM* hypervisor [14] and on *vhost* [20], an efficient in-kernel implementation of the virtio [24] paravirtual I/O framework.

In Section 5 we present some promising preliminary results: We show that, indeed, the I/O path becomes exitless. For a network-intensive workload on one guest virtual CPU (VCPU), we measure a throughput improvement of 45% over unmodified vhost-net. We show that the throughput improvement does *not* come at the expense of latency (which is actually reduced by around 22 μsec), even when one core handles the I/O of several guests. We demonstrate that multiple guests can effectively share one I/O core, until this core is saturated, though more research in necessary to increase the number of guests that a single I/O core can serve.

2. Motivation and Related Work

Spatial division of cores, as opposed to temporal division, is commonly used by hypervisors because it improves performance on multi-core systems [15, 17, 18]. To obtain the benefits of spatial division, hypervisors exposes paravirtual I/O adapters designed to handle all the I/O requests asynchronously. Using asynchronous I/O models, the guest can continue running on one core while the hypervisor handles the I/O requests on a different core. However, as we show later in Section 5, spatial division of cores still suffers from a significant performance overhead in virtual environments. The main cause of this overhead is the high cost to send notifications between cores.

In a non-virtual environment, software can use a lightweight architectural mechanism, Inter-Processor Interrupts (IPI), to send notifications. In contrast, in a virtual environment, sending an IPI is considered a privileged operation and requires hypervisor intervention. Thus, if the guest sends an IPI, the hypervisor needs to trap and emulate the operation. Trapping the guest execution requires switching to the hypervisor context and returning to the guest context. These switches, known as exits, significantly degrade the performance of virtual machines [2, 8, 16]. In the opposite direction, if the hypervisor sends an IPI to notify the guest, the CPU also forces an exit because interrupts generated by the physical hardware are also considered privileged events. The chain of events for handling paravirtual I/O requests is illustrated in Figure 1(a). First, the guest performs a privileged instruction to request an I/O operation (guest-to-host notification). This instruction forces an exit and switches to the hypervisor context. The hypervisor examines the exit cause, starts processing the I/O request on a separate core and resumes the guest execution. Finally, the hypervisor notifies the completion of the I/O request (host-to-guest notification) by injecting a virtual interrupt to the guest, which requires an additional exit.

SplitX [16] proposes hardware extensions for running virtual machines on dedicated cores, with the hypervisor running in parallel on a different set of cores. The paper proposes a new hardware mechanism for exitless communication between cores, which could be used for paravirtual I/O notifications between guest and host cores. In contrast, ELVIS does not require any hardware modifications and runs on current hardware.

VPE [18] also uses dedicated cores for the hypervisor, but does not remove the costly exits for hypervisor-to-guest notifications. The number of these notifications can be reduced by virtual interrupt coalescing [4].

Previous work [9] showed that using polling in both guest and host, along with other optimizations, can elimate I/O virtualization overheads in a storage controller. As we elaborate in Section 3, ELVIS avoids guest-side polling which adversely impacts performance.

Many papers tried to reduce the overhead of paravirtual I/O. Most of them did so by enhancing the backend driver in the hypervisor or making generic improvements such as avoiding needless copies between the hypervisor and the guest. While these approaches are viable, they are orthogonal to our approach of improving the communication between the frontend driver in the guest and the backend. Mansley et al. [19] propose a hybrid of paravirtual I/O and device assignment where the slow path goes through the hypervisor and the fast path goes directly to the device. Apart from the inherent problems with device assignment (e.g., difficult live migration), this approach does not work when there is no physical device, e.g., when the guest's disk is backed by a file on the host filesystem. Santos et al. [23, 26] instrument Xen's paravirtual drivers and reduce overhead by avoiding packet copies and using a multiqueue NIC. In contrast, our approach does not require special hardware.

ELI [13] achieves 97–100% of bare-metal performance by delivering interrupts sent by physical devices to the guest without hypervisor intervention (exits). ELI focused only

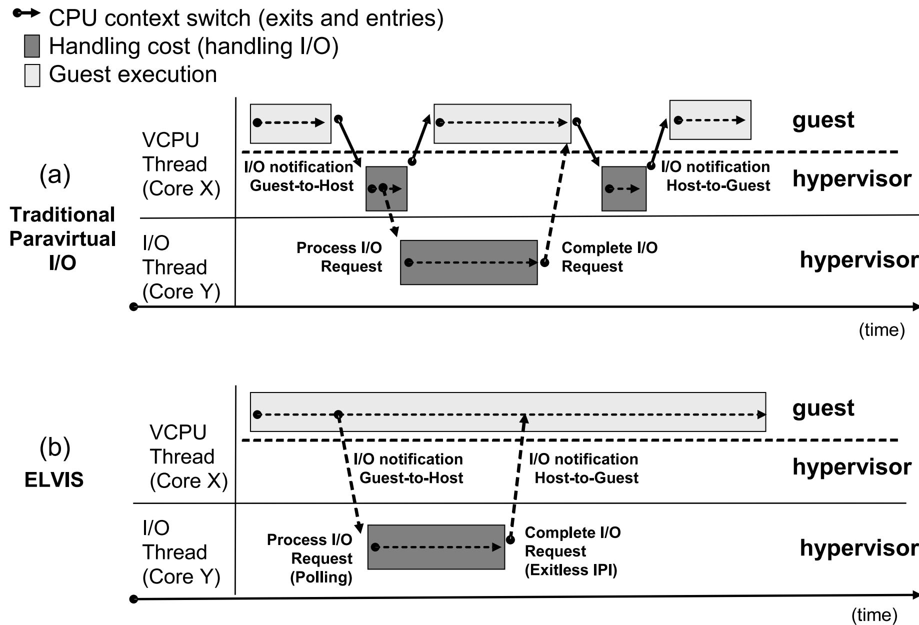

Figure 1. Exits during paravirtual I/O handling.

on physical assigned devices while we focus on paravirtual devices.

3. Exitless Notifications

We propose ELVIS (Exit-Less Virtual I/o System), an exit-less paravirtual I/O model for virtual machines, which takes advantage of spatial division of cores, yet avoids exits caused by inter-core communication.

Different techniques are needed to provide exit-less notifications in the two directions — from guest to host, and from host to guest:

3.1 Guest-to-host Notifications

With ELVIS, when any of the guests wishes to send a notification to the hypervisor, instead of performing a costly privileged instruction, the guest simply writes the notification in a memory area shared with the hypervisor. The hypervisor polls the memory area from a different core and handles the I/O request when needed. This type of polling technique is efficient in virtual environments [9, 18] but requires a fully dedicated core to poll the shared memory area. Dedicating an additional core for each VM is unacceptable and may waste CPU resources. Thus, ELVIS uses a single dedicated core to serve multiple VMs, considering fairness, quality of service and performance. As we show later in Section 5, ELVIS can efficiently serve multiple VMs using a single dedicated core.

For workloads which are not I/O-intensive, the waste inherent in excessive polling may outweigh the benefits of exit-less notifications. It is therefore beneficial to dynamically switch between polling and traditional exit-based guest-to-host notifications. Such switching is often used in the context of interrupt mitigation [21, 25], and has also been used for paravirtual I/O by VMWare's VMXNET3 [29].

3.2 Host-to-guest Notifications

On the opposite direction (host-to-guest notifications), if each guest were to poll for notifications sent by the hypervisor, all the guests would waste a significant amount of cycles that could otherwise be used to run more useful work or just kept unused to reduce power consumption. Due to these disadvantages, polling for notifications in the guest side is impractical.

Instead, ELVIS uses inter-processor interrupts (IPIs) to send notifications from the hypervisor's I/O thread to the guest. Normally, these interrupts would cause guest exits, and their associated performance penalties. We therefore use the Exit-Less Interrupts (ELI) technique [13] to deliver these notifications directly to the guest. ELI asks the processor to deliver all interrupts to the running guest, with the guest's interrupt descriptor table (IDT) shadowed so that only the intended interrupts are actually delivered to the guest and the rest cause an exit to the hypervisor. The ELI paper focused on assigned devices and on the interrupts they generate — but we can also use the same mechanism for delivering an IPI directly to the guest.

Figure 1(b) illustrates that, combining guest-to-host polling based notifications with host-to-guest exitless IPI based notifications, ELVIS handles I/O without exits on the critical path.

4. Proof of Concept

To measure the impact of exitless notifications on paravirtual I/O performance, we implemented a proof of concept of ELVIS within the KVM hypervisor. The KVM hypervisor [14] is implemented as a Linux kernel module that extends the kernel with hypervisor capabilities, driven by a QEMU [7] user process. KVM offers two different implementations for paravirtual network I/O devices: (1) a user-space implementation, part of QEMU (*qemu-virtio-net*); and (2) a Linux kernel module implementation (*vhost-net*). Both implementations share the same guest driver. We based ELVIS on vhost-net because it performs significantly better than qemu-virtio-net [20]. While we focus on KVM and vhost-net, ELVIS principles are also applicable to other hypervisors and other paravirtual I/O implementations.

To take advantage of multi-core systems and handle I/O asynchronously, KVM creates multiple threads. For each VM, KVM creates a thread per VCPU to run the guest code and an additional thread to handle I/O requests (I/O thread). Thus, a uniprocessor guest running I/O intensive workloads will have two active threads that can run simultaneously on different cores. While the guest and KVM use shared ring buffers to transmit and receive data efficiently [24], they still need to notify each other when new buffers are ready to be processed. The guest sends notifications to KVM executing a programmable I/O instruction (PIO). PIO is a privileged instruction, therefore causing an exit. KVM sends notifications to the guest in the form of virtual interrupts generated by the paravirtual I/O device. Unfortunately, due to x86 hardware virtualization limitations, virtual interrupts cannot be delivered to the guest while it is still running. KVM first forces an exit to stop the execution of the guest and then injects the corresponding virtual interrupt.

We applied the ELVIS model to KVM: First, we modified vhost-net to poll for notifications sent by the guests and neutralized the usage of the costly PIO instructions. To keep fairness and maximize performance across multiple VMs, we also modified vhost-net to use a single thread to handle multiple devices (i.e., multiple VMs) instead of a thread per device. Otherwise, multiple vhost-net I/O threads would be competing with each other for CPU cycles. Finally, we changed KVM to deliver virtual interrupts using IPIs and ELI. This last mechanism allows vhost-net to send exitless notifications to the running guests.

5. Evaluation

We analyzed ELVIS' impact on virtual network I/O measuring throughput and latency. Performance-minded applications would typically dedicate whole cores to guests (single VCPU per core) thus we limited our evaluation to this case. Our test machine was an IBM System x3550 M2, which is a dual-socket, 4-cores-per-socket server equipped with Intel Xeon X5570 CPUs running at 2.93 GHz. The system included 24GB of memory and an Emulex OneConnect 10Gbps NIC. We used another similar remote server connected directly by 10Gbps fiber as the target for I/O transactions. We set the Maximum Transmission Unit (MTU) to its default size of 1500 bytes; we did not use jumbo Ethernet frames. The host ran Linux 3.1, with IOMMU enabled, and Qemu 0.14.0. The guest ran Linux 3.1, and its memory was backed by huge (2 MB) pages in the host.

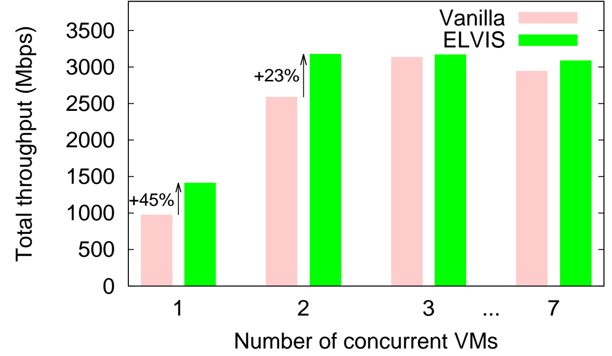

Figure 2. Total throughput (from all VMs) of Netperf TCP Stream, when run from 1 to 7 VMs serviced by one I/O core.

We used Netperf TCP-stream to measure throughput improvements. This benchmark opens a single TCP connection to the remote machine, and makes as many rapid write() calls of a given size as possible. We configured the benchmark to perform 64 byte writes, which fully loaded the tested machine's CPUs (so that throughput can be compared), but did not saturate the remote machine.

As described in Section 4, the unmodified version of vhost-net creates a thread per I/O device. Unfortunately, vhost-net threads continue running on a CPU as far as they have work to perform. This implementation does not consider fairness among VMs when multiple vhost-net threads need to share the same CPU in a non-preemptible kernel (default in server setups). If multiple vhost-net threads run on the same core, they can starve each other. In this case, only one VM might achieve good performance while the others experience extremely low performance. To avoid this problem and make a fair comparison with ELVIS, we improved the implementation of vhost-net to periodically release the core when needed, so that each vhost-net thread gets an equal and proportional part of the CPU. Our improvement has already been accepted into the Linux kernel, and will be part of future releases.

Figure 2 compares the results we obtained with the improved vhost-net (*vanilla*) and with ELVIS. To verify that ELVIS is able to serve multiple VMs using a single I/O thread running on a dedicated core, we ran the experiment using 1 through 7 VMs. In all the configurations we used one core per VM and one additional core to run all the vhost-net I/O threads. We can see that ELVIS improved throughput by 45% and 23% for 1 and 2 VMs respectively. As is evident

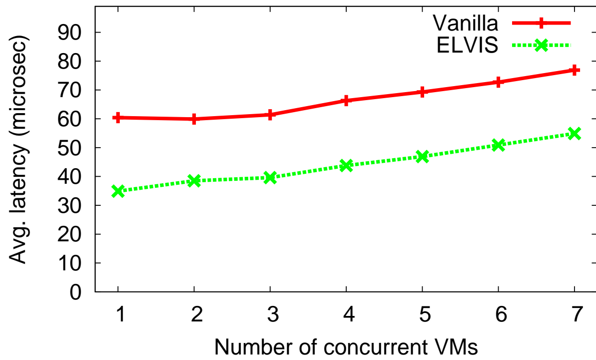

Figure 3. Average latency as measured by Netperf UDP-RR, when run from 1 to 7 VMs. With ELVIS, latency is lower, and remains low even when multiple VMs are serviced by one I/O core.

in the figure, the single vhost-net I/O core (used by both vanilla vhost-net and ELVIS) was saturated at a throughput of around 3100 Mbps, leading to a plateau for 3 VMs and more, where both ELVIS and vanilla could no longer scale. This scalability problem would not be an issue for less bandwidth-intensive workloads (as we demonstrate below), and can also be alleviated by allocating more than one I/O core. Moreover, we believe that with further research, the I/O capacity of the single I/O core can be significantly increased, thereby significantly increasing the number of guests which could be served by a single I/O core.

As expected, ELVIS was able to reduce the average number of exits per second from 120,000 to only 800 when running a single VM and from 32,000 to 800 per core when running 7 VMs. Most of these remaining 800 exits per second are not related to I/O — for example, 500 of them are related to timer interrupts.

The number of notifications (exits for vanilla) per core decreases as the number of VMs increases. From 1 to 2 VMs throughput more than doubled because the single I/O core was batching and coalescing multiple notifications: While the I/O thread processes requests for the first VM, the incoming requests for the second VM are batched. For a single batch, the I/O thread sends only one notification. This side-effect improved throughput at the cost of higher latency. Additionally, when running 3 VMs or more the total throughput remains constant and therefore each VM performs less work, further decreasing the number of notifications per core.

We measured ELVIS's latency improvement using Netperf UDP Request-Response, which sends a UDP packet and waits for a reply before sending the next. Figure 3 presents the results. We can see that ELVIS reduced latency by 25μsec compared to vanilla vhost-net when only a single VM was running. With multiple VMs ELVIS reduced the average latency per VM by 22μsec. This improvement was possible because ELVIS' single thread model, as opposed to multiple threads per VM, combined with exitless notifications significantly reduced the time it takes to detect and handle the I/O requests sent by the guests. Compared to the TCP stream benchmark we previously analyzed, UDP Request-Response did not saturate the I/O core and scaled very well.

6. Conclusions and Future Work

In this paper we argued that a high-performance paravirtual I/O system must be exitless. We proposed a new model for exitless notifications between guests and the hypervisor running on distinct cores. We described our initial implementation of this model and evaluation results which showed improvement of up to 45% in throughput and 25μsec reduction in latency over the baseline virtualized system.

However, our evaluation demonstrated that more research is necessary to make exit-less paravirtual I/O efficient for more workloads, especially when running multiple VMs concurrently. In particular we plan to evaluate (and increase) the number of VMs that a single I/O thread can handle, to find the right balance to dynamically create and destroy I/O threads depending on workload characteristics and the number of active VMs, and to dynamically switch between polling and exit-based notifications. We also plan to extend this work to disk I/O.

Acknowledgments

The research leading to the results presented in this paper is partially supported by the European Community's Seventh Framework Programme ([FP7/2001-2013]) under grant agreement #248615 (IOLanes).

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2006.

[3] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. The evolution of an x86 virtual machine monitor. *ACM SIGOPS Operating Systems Review*, 44(4):3–18, 2010.

[4] I. Ahmad, A. Gulati, and A. Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conference (ATC)*, 2011.

[5] AMD Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming, 2011.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC)*, pages 41–46, 2005.

[8] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2010.

[9] M. Ben-Yehuda, E. Borovik, M. Factor, E. Rom, A. Traeger, and B.-A. Yassour. Adding advanced storage controller functionality via low-overhead virtualization. In *USENIX Conference on File & Storage Technologies (FAST)*, 2012.

[10] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2008.

[11] Y. Dong, Z. Yu, and G. Rose. SR-IOV networking in Xen: architecture, design and implementation. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2008.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[13] A. Gordon, A. Nadav, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2012.

[14] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)*, 2007.

[15] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[16] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[17] G. Liao, D. Guo, L. Bhuyan, and S. 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)*, 2008. ISBN 978-1-60558-346-4.

[18] J. Liu and B. Abali. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM International Conference on Supercomputing (ICS)*, pages 225–234, 2009.

[19] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope. Getting 10 Gb/s from Xen: safe and fast device access from unprivileged domains. In *Conference on Parallel Processing (Euro-Par)*, 2007.

[20] Michael S. Tsirkin. vhost-net and virtio-net: need for speed. `http://www.linux-kvm.org/wiki/images/8/82/Vhost_virtio_net_need_for_spe%ed_2.odp`, 2010. (KVM Forum 2010).

[21] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)*, 15:217–252, 1997. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/263326.263335. URL `http://doi.acm.org/10.1145/263326.263335`.

[22] PCI SIG. Single root I/O virtualization and sharing 1.0 specification, 2007.

[23] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10Gbps using safe and transparent network interface virtualization. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, 2009.

[24] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review (OSR)*, 42 (5):95–103, 2008.

[25] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *Anual Linux Showcase & Conference*, 2001. URL `http://portal.acm.org/citation.cfm?id=1268488.1268506`.

[26] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference (ATC)*, 2008.

[27] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, 2001.

[28] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38 (5):48–56, 2005. ISSN 0018-9162.

[29] VMWare Inc. Esx server 2 - architecture and performance implications. Technical report, VMWare, 2005.

[30] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical Report H-0263, IBM Research, 2008.

[31] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symposium (OLS)*, pages 261–268, 2008.