H-0263 (H0809-006) September 20, 2008
Computer Science

IBM Research Report

Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines

Ben-Ami Yassour, Muli Ben-Yehuda, Orit Wasserman
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines

Ben-Ami Yassour benami@il.ibm.com
Muli Ben-Yehuda muli@il.ibm.com
Orit Wasserman oritw@il.ibm.com

IBM Haifa Research Lab, Haifa, Israel

Abstract

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, but letting VMs access devices directly has a number of particularly interesting benefits, such as not requiring any guest VM changes and in theory providing near-native performance.

In an effort to quantify the benefits of direct device access, we have implemented direct device assignment for untrusted, fully-virtualized virtual machines in the Linux/KVM environment using Intel's VT-d IOMMU. Our implementation required no guest OS changes and—unlike alternative I/O virtualization approaches—provided near native I/O performance. In particular, a quantitative comparison of network performance on a 1GbE network shows that with large-enough messages direct device access throughput is statistically indistinguishable from native, albeit with CPU utilization that is slightly higher.

1 Introduction

I/O virtualization can be implemented in one of three ways: device emulation, para-virtualized ("virtual") I/O drivers, and direct assignment.[1] Emulation means that the host emulates a device that the guest already has a driver for [16].The host traps all device accesses and converts them to operations on a real, possibly different, device, as depicted in Figure 1(a). This approach requires many world switches[2] and has relatively low I/O performance. 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.

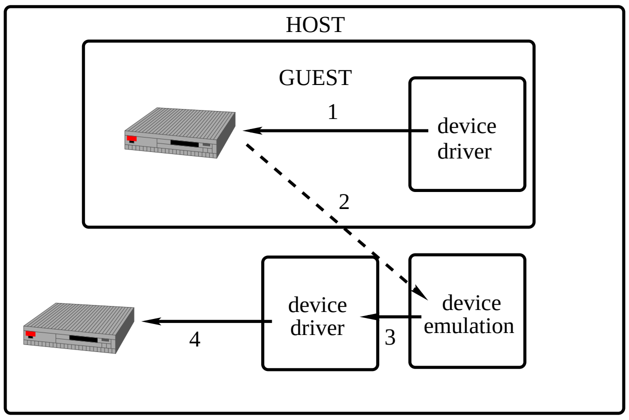
With para-virtualized I/O devices, special, hypervisor-aware I/O drivers are installed in the guest (see Figure 1(b)). By raising the level of interaction from hardware-level operations (e.g., an MMIO access) to high-level operations (e.g., "send a packet"), overhead is reduced and performance improved. All modern hypervisors implement such para-virtualized drivers [1, 9, 13], but their performance is still far from native [14] and they require guest changes, which may or may not be feasible.

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* (see Figure 1(c)). This approach should improve 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. On the other hand, it is not fully compatible with live migration [5, 15], although efforts are underway to address this limitation [20, 7], and it requires dedication of a device to a virtual machine. Self-virtualizing adapters [12, 19, 11], which are starting to become available, solve the adapter sharing problem by presenting a single device as multiple devices to multiple VMs.

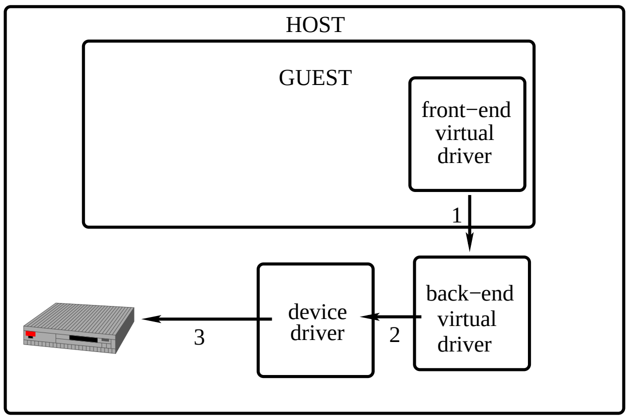
We present the implementation of direct assignment in the Linux/KVM environment in Section 2 and a quantitative comparison and analysis of direct access in Section 3. We survey related work in Section 4 and conclude with a short discussion of what the future holds for direct access in Section 5.

[1]A virtual machine might use one, two or all three mechanisms at the same time: Xen driver domains [6], for example, use direct device assignment to provide virtual I/O devices to other VMs.

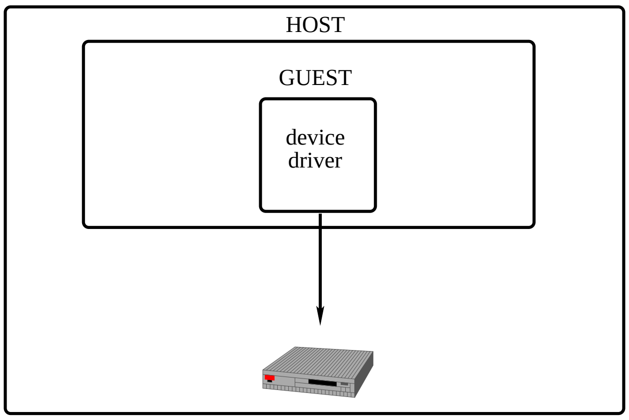
[2]A world switch is a context switch between guest VM and host hypervisor.



(a) Emulation flow: (1) guest writes to register 0x789 of emulated device (2) . . . trap: ahaa, the guest wants to send a packet! (3) send packet (4) write to register 0x789 of real device.



(b) Virtual I/O drivers flow: (1) send a packet (2) send packet (3) write to register 0x789 of real device.



(c) Direct access flow: (1) write to register 0x789.

Figure 1: Different I/O virtualization modes.

2 Direct access in KVM

Generally speaking, system software performs input/output to a hardware device via one of four distinct mechanisms: programmed I/O (PIO, also commonly referred to as port I/O), memory-mapped I/O (MMIO), interrupts, and direct memory access (DMA). The key goal of direct access is to allow a guest OS to access a device *directly*, i.e., without a software intermediary, but some accesses can be intercepted without loss of performance.

2.1 MMIO and PIO

Intel's VT and AMD's SVM virtualization extensions provide mechanisms for the host to be notified whenever a guest VM tries to execute a PIO instruction or perform an MMIO access. Alternatively, the host may let the VM execute PIOs or MMIOs on the device directly. In our initial implementation, PIO and MMIO accesses were trapped by the hypervisor and passed to the userspace component of KVM. This component then validated the accesses and if necessary executed them on the real device. Initial performance results, however, indicated that exits due to MMIO accesses could have a non-negligible performance impact, which led us to implement a "direct-MMIO" mode. In direct-MMIO mode MMIO accesses are *not* intercepted by KVM and are instead executed directly on the device. We note that some PIO accesses could be passed through directly in the same manner, but PIO's are rarely used on the fast-path of a high-speed device[3]. The limited performance benefit of direct-PIO was deemed not to be worth the additional complexity.

2.2 Interrupts

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. In our implementation interrupts are always received by the host. The host acknowledges the interrupt at the IOAPIC, disables the interrupt line so that the interrupt handler will not be called again and injects the interrupt to the guest. Once the guest acknowledges the interrupt and a new interrupt can be serviced, the host re-enables the interrupt line. Note that this mechanism cannot support shared PCI interrupts, since an untrusted guest could potentially delay its acknowledgment forever, thus keeping the (shared) interrupt line disabled. This is a limitation of our approach which we are working to address.

[3]Some guest pio accesses, e.g., to device BARs, could adversely affect the host and always need to be validated.

2.3 Direct memory access (DMA)

In a virtualized environments, guests have their own view of physical memory, which KVM refers to as "guest physical", and which is distinct from the host's "host physical" view of memory [9]. Although there are ways of giving fully-virtualized guests DMA access to portions of host memory without hardware support [10], such approaches can only work for *trusted* guests. Solving the DMA problem for the general case of *untrusted*, non-hypervisor-aware guests, requires hardware support [2].

An I/O Memory Management Unit (IOMMU) on the I/O path between a device and memory validates and translates all device accesses to host memory. With an IOMMU the host can let the guest program the device with guest physical addresses while setting the proper translations (mappings) from guest physical addresses to host physical addresses in the IOMMU translation table used by that device.

Since the Linux kernel already contained support for setting-up and programming Intel's VT-d IOMMU, all that was required to handle DMA in our implementation—other than fixing the odd bug or three—was to "hook up" the kernel's VT-d code to the KVM guest memory mapping routines. We did it in such a way that any host physical address mapped by the guest at a given guest physical address has the same guest physical address to host physical address mapping in the the IOMMU translation table for any device the guest has direct access to. There is nothing inherently specific to VT-d in our implementation: any isolation-capable IOMMU supported by Linux could be plumbed into KVM in the same manner.

Willman, Rixner, and Cox presented four policies for deciding when to create or remove IOMMU mappings [18]: *single use*, *shared mapping*, *persistent mapping* and *direct mapping*. The first three policies require a para-virtualized interface for the guest to map and unmap its memory, and thus are not appropriate for fully-virtualized (unmodified) guests. They also have a non-negligible performance overhead [3, 18]. *Direct mapping*, on the other hand, is transparent to the guest and requires minimal CPU overhead. Therefore, our implementation implements direct mapping.

Having said that, we do note that direct mapping requires pinning the guest's entire memory (no memory over-commit) and provides no protection inside a guest (intra-guest protection), only between different guests (inter-guest protection). Overcoming its limitations is part of our on-going work, as discussed in Section 5.

3 Performance results

3.1 Experimental setup

We compared the performance of direct access versus native Linux, KVM's emulated e1000 NIC and para-virtualized virtio network driver (referred to as "virtio" below).

Our setup consisted of two Lenovo M57p machines with the Intel Q35 chipset (which includes VT-d). Each machine had a 2.66GHz dual-core Intel Core 2 Duo CPU with 4GB of memory. The machines were connected directly with a 1GbE cable. One Lenovo machine ran native Linux (Ubuntu 7.10 for x86_64) and the other ran Linux (Ubuntu 7.10 for x86_64) with KVM, with a single virtual machine running Fedora Core 8 (64 bit), with 1GB of memory. All runs, native and virtualized, used the on-board e1000e PCI-e NIC.

Both the hosts and the guest ran a Linux kernel with VT-d support, based on the Linux 2.6.27-rc4 KVM git tree[4]. On the host running the VM we used the kvm-userspace git tree[5], again with added VT-d support.

We ran both the `iperf` [17] and `netperf` [8] benchmarks, and measured throughput and CPU utilization. For the sake of brevity we only present the netperf results, but the iperf results were substantially similar. The VM and the native machine alternated sender and receiver roles. The sender (running `netperf`) was run with the following parameters: `-H <address> -l 60`, and the receiver (running `netserver`) was run with no command line arguments.

When running native, Linux used both cores. When running a virtual machine, the virtual machine was given 1 virtual CPU and was not pinned to a specific core. We note that the upper bound for CPU utilization is therefore 200% (100% $\times$ 2 cores).

We ran the following setups:

- The baseline for comparison was native Linux (no virtualization) with VT-d disabled (`intel_iommu=off` specified in the kernel's command line).
- Virtual machine using the *emulated* e1000 device.
- Virtual machine using KVM's para-virtualized, virtio-based network driver virtnet (*"virtio"*).
- Virtual machine with *direct access* to the on-board e1000e adapter.

We repeated each setup 5 times, measuring in each case throughput and CPU utilization. The values presented in graphs are the averages.

3.2 Performance results and analysis

As can be seen in Figure 2, the overall throughput in the case of direct-access was 99.7% of native, and is 260% better then emulation. The throughput of virtio is 93% compared to direct access, and the CPU utilization of virtio is 314%(!) of direct access.

Santos et al. recently also showed that a sizable gap remains between virtual I/O drivers and native [14], using Xen's state-of-the-art virtual I/O drivers. Although it is less pronounced in 1GbE environments, this performance gap is inherent in the architectural differences between virtual I/O drivers and direct access. We expect that in a 10GbE environment, where the CPU is likely to be the bottleneck, direct access will perform significantly better than virtio, since it requires several times less cycles to push (or receive) a packet.

The results for network receive (Figure 3) are roughly the same as for send, except the differences between virtio and direct access are less pronounced. We note however that in a multiple VM scenario, where received packets need to be dispatched to the right VM, direct access—which does the dispatching in hardware—has an advantage over virtio which has to do the dispatching in software.

It is well-known that network CPU overhead is relative to application buffer sizes. We looked at the effect different application buffer sizes had on the throughput and CPU utilization of direct access. As can be seen in Figure 4, throughput increases as message sizes increase, and the CPU utilization decreases. We note however that with virtio the difference is more pronounced: virtio works a lot harder for small messages and a bit harder for larger messages. This leads us to conclude that the smaller the application buffer size, the more compelling it is to use direct access.

Last but not least, we analyzed the performance gap between native and direct access. The key observation is that direct access gets rid of the virtualization overhead for the I/O path, but there re-

[4]changeset `ce094fc0d25cb364bce6f854dffc6849876ab89a`.
[5]changeset `e82e58b1e889010b531dac616d0b94f76de66b09`.

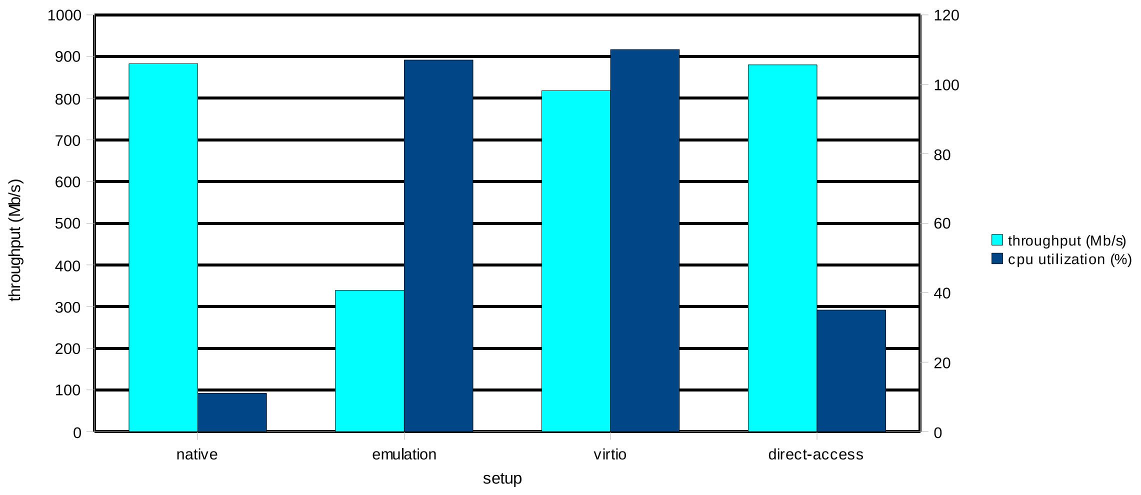


Figure 2: Performance comparison: network send.

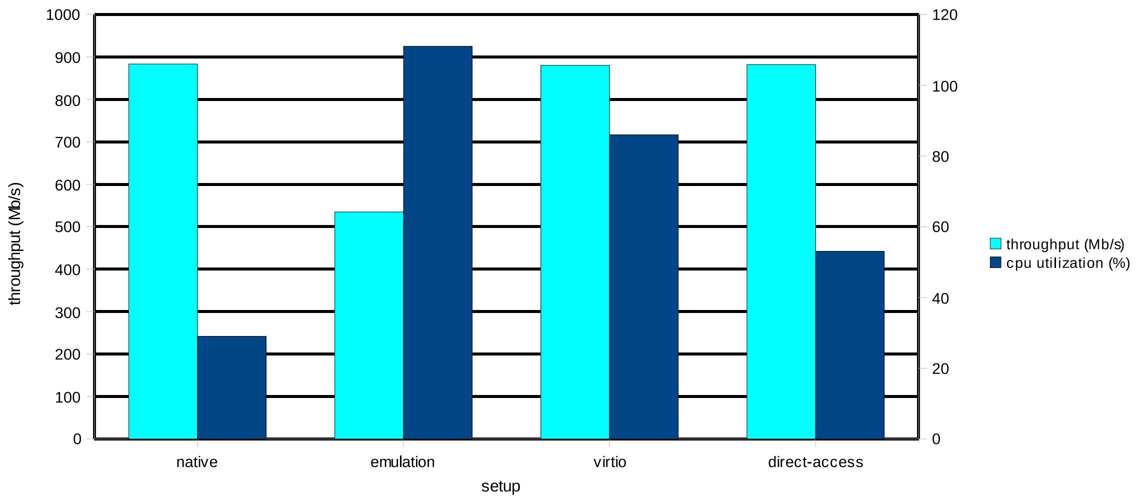


Figure 3: Performance comparison: network receive.

mains residual overhead for the CPU and MMU virtualization. The increased CPU utilization of direct access as compared to native comes from guest exits. MMIOs and DMAs do not cause exits in our implementation, and PIOs do not occur on the fast path. Therefore, all of the guest exits are either due to interrupts or due to "generic" virtualization exits such as page faults. Interrupt coalescing and switching to polling the adapter can reduce or even eliminate the interrupt exits overhead, and the advances in CPU and MMU virtualization (e.g,. the introduction of nested paging [4]) will continue reducing the generic virtualization overhead, to the point where we expect direct access to be virtually indistinguishable from native.

4 Related work

In an earlier work we discussed the design considerations of IOMMU support in hypervisor environments [2], focusing on *para-virtualized* virtual machines in the Xen hypervisor environment using the IBM Calgary IOMMU. In this work we focus on *fully-virtualized* virtual machines in the KVM environment with Intel's VT-d IOMMU.

In a follow-on work we presented the performance penalties associated with direct access and IOMMUs [3], again focusing on a para-virtualized Xen environment, para-virtualized mapping strategies and the Calgary/CalIOC2 family of IOMMUs. Willman, Rixner and Cox [18] extended that work and pre-

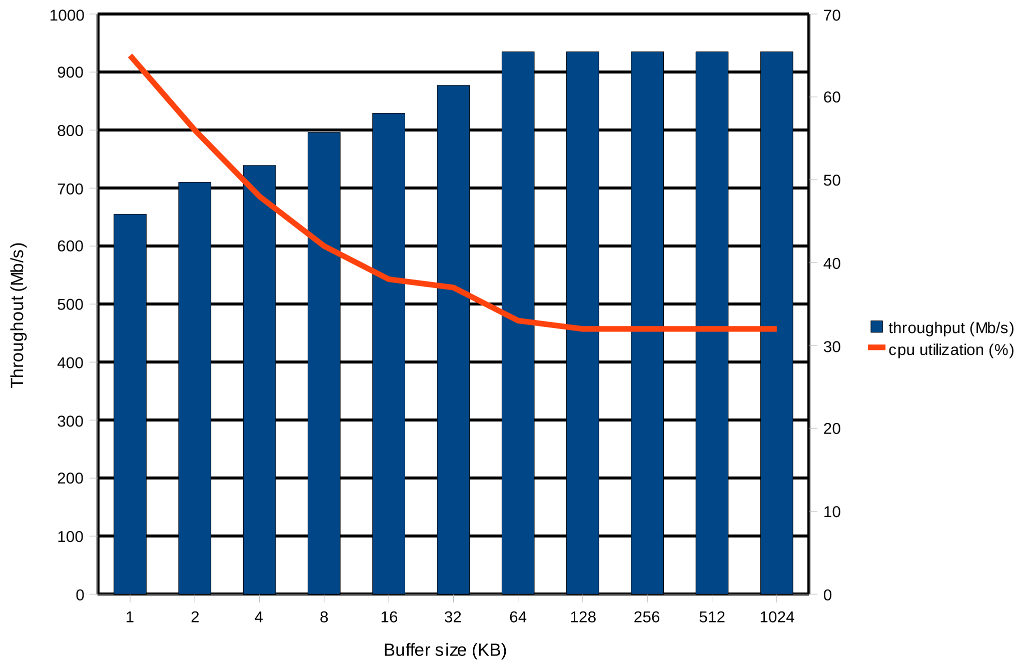


Figure 4: Effect of buffer size on throughput and CPU utilization.

sented the four different mapping strategies mentioned in Section 2.3. Their evaluation however was done in a simulated environment using AMD's GART rather than an isolation-capable IOMMU. Our results are from a full implementation of direct access using Intel's VT-d isolation-capable IOMMU. Additionally, we compare and contrast these results with the emulation and para-virtualized modes of I/O virtualization.

To the best of our knowledge, direct access using Intel's VT-d IOMMU was first implemented by the Xen developers. There are numerous implementation differences between the Xen and KVM implementations which stem from the different hypervisor architectures. For example, unlike our implementation which made use of Linux's VT-d support, the Xen implementation required re-implementing VT-d in the Xen hypervisor itself. Additionally, as far as we know no detailed technical evaluation of the Xen direct access support has been published, but we expect that a full analysis of the different I/O virtualization modes Xen supports would roughly parallel our results, except that Xen's para-virtualized I/O drivers are somewhat more mature than the KVM alternatives [14].

5 Conclusions and Future work

It is evident from the results presented in Section 3 that direct access with an IOMMU provides excellent I/O performance for untrusted fully-virtualized virtual machines. However, performance is not everything. Direct access is fundamentally about bypassing the virtualization abstraction layer, and by bypassing this layer we lose some of the benefits of virtualization, such as support for live migration [5, 15]. Several approaches have been proposed for combining direct access with live migration (e.g., Zhai, Cummings and Dong proposed bonding [20] and Huang et al. proposed adapter hardware changes [7]) but each of the proposed approaches has different limitations.

Another limitation of direct access using direct mapping is that it requires pinning all of the guest's memory. A para-virtualized interface for selective IOMMU mapping ("pvdma" in KVM parlance) will allow us to avoid pinning unneeded memory, but is also likely to incur a significant performance cost [3, 18]. It is our belief that improving "pvdma" to the point where it is as performant as direct mapping is possible, and we are actively pursuing it.

To conclude, direct access is a valuable alternative approach for I/O virtualization today, which provides near-native performance, as demonstrated by our implementation of direct access for KVM. Hardware advances such as self-virtualizing adapters, interrupt re-mapping, and more efficient CPU and MMU utilization are likely to continue making direct access an attractive I/O virtualization choice. Ultimately, we believe that the future of I/O virtualization is a combination of direct access on the software side coupled with self-virtualizing, intelligent adapters on the hardware side. With the right combination of software and hardware, direct access can provide native performance while also providing all of the benefits of software-based methods for I/O virtualization.

Acknowledgments

The authors would like to express their appreciation for the efforts of their collaborators on the KVM direct access project: Amit Shah of Qumranet and Allen M. Kay and Weidong Han of Intel. Additionally, the authors would like to thank the KVM developers for giving us a true Linux-based hypervisor to experiment with.

References

[1] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[2] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing iommus for virtualization in Linux and Xen. In *OLS '06: The 2006 Ottawa Linux Symposium*, pages 71–86, July 2006.

[3] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn. The price of safety: Evaluating iommu performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 9–20, July 2007.

[4] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, New York, NY, USA, 2008. ACM.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005.

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

[7] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda. Nomad: migrating os-bypass networks in virtual machines. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 158–168, New York, NY, USA, 2007. ACM Press.

[8] S. D. Jones R., Choy K. Netperf. HP Information Networks Division, Networking Performance Team, http://www.netperf.org, 2001.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[10] J. Levasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, page 2, Berkeley, CA, USA, 2004. USENIX Association.

[11] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *USENIX '06: Proceedings of the 2006 USENIX Annual Technical Conference*, page 3, Berkeley, CA, USA, 2006. USENIX Association.

[12] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC '07: Proceedings of the 16th international symposium on high performance distributed computing*, pages 179–188, New York, NY, USA, 2007. ACM Press.

[13] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[14] J. R. Santos, Y. Turner, J. G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In

USENIX '08: USENIX Annual Technical Conference, pages 29–42, June 2008.

[15] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 377–390, 2002.

[16] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on vmware workstation's hosted virtual machine monitor. In *USENIX '01: USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[17] A. Tirumala and J. Ferguson. Iperf 1.2 - the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf, 2001.

[18] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX '08: USENIX Annual Technical Conference*, pages 15–28, 2008.

[19] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 306–317, 2007.

[20] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *OLS '08: The 2008 Ottawa Linux Symposium*, pages 261–268, July 2008.