Live Migration of Direct-Access Devices

Asim Kadav and Michael M. Swift
Computer Sciences Department, University of Wisconsin-Madison

Abstract

Virtual machine migration greatly aids management by allowing flexible provisioning of resources and decommissioning of hardware for maintenance. However, efforts to improve network performance by granting virtual machines direct access to hardware currently prevent migration. This occurs because (1) the VMM cannot migrate the state of the device, and (2) the source and destination machines may have different network devices, requiring different drivers to run in the migrated virtual machine.

In this paper, we describe a lightweight software mechanism for migrating virtual machines with direct hardware access. We base our solution on *shadow drivers*, which efficiently capture and reset the state of a device driver. On the source hardware, the shadow driver continuously monitors the state of the driver and device. After migration, the shadow driver uses this information to configure a driver for the corresponding device on the destination. We implement our solution for Linux network drivers running on the Xen hypervisor. We show that the performance overhead, compared to direct hardware access, is negligible and is much better than using a virtual NIC.

1 Introduction

Virtual machine migration, such as VMware VMotion [8] and Xen and KVM Live Migration [10, 3], is a powerful tool for reducing energy consumption and managing hardware. When hardware maintenance is required, running services can be migrated to other hardware platforms without disrupting client access. Similarly, when hardware is underutilized, management software can consolidate virtual machines on a few physical machines, powering off unused computers.

Virtual machine migration relies on complete hardware mediation to allow an application using a device at the source of migration to be seamlessly connected to an equivalent device at the destination. In the case of disk storage, this is often done with network disk access, so both the virtual machines refer to a network-hosted virtual disk. In the case of network devices, this is often done with a virtual NIC, which invokes a driver in the virtual machine monitor [17], in a driver virtual machine [5, 7, 12], or in the host operating system [9, 13]. Complete mediation provides compatibility, because the virtual machine monitor can provide a uniform interface to devices on all hardware platforms, and provides the VMM with access to the internal state of the device, which is managed by the VMM or its delegated drivers and not by the guest.

However, complete mediation of network access is a major performance bottleneck due to the overhead of transferring control and data in and out of the guest virtual machine [6, 11]. As a result, several research groups and vendors have proposed granting virtual machines *direct access* to devices. For example, Intel's VT-d provides safe access to hardware from virtual machines [1]. However, direct device access, also called passthrough I/O or direct I/O, prevents migration, because the device state is not available to the VMM. Instead, a real device driver running in the guest manages the device state, opaque to the VMM.

In this paper, we leverage shadow drivers [14, 15] to migrate virtual machines that directly access devices. The shadow driver executes in the guest virtual machine to capture the relevant state of the device. After migration, the shadow driver disables and unloads the old driver and then initializes a driver for the device at the migration destination. In addition, the shadow driver configures the device driver at the destination host.

We have a prototype implementation of shadow driver migration for network device drivers running in Linux guest operating systems on the Xen hypervisor. In experiments, we find that shadow drivers have little impact on passthrough-I/O performance.

In the following section, we describe the architecture of I/O virtualization. Following that, we describe the architecture of our solution, then the implementation of passthrough-I/O migration, followed by a brief evaluation and conclusions.

2 I/O Virtualization

Virtual machine monitors and hypervisors must provide guest virtual machines access to hardware devices. With *virtual I/O*, operations in a guest VM are intercepted by the VMM and carried out by either the hypervisor [17], a host operating system [9, 13], or a driver executing in

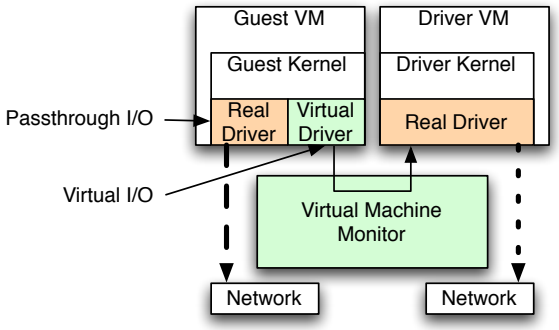

Figure 1: I/O paths in a virtual machine.

a privileged virtual machine [5, 7, 12]. The performance of virtual I/O is lower than native I/O on physical hardware because the hypervisor must interpose on all I/O requests, adding additional traps. Figure 1 shows an example system with two network cards, the left accessed via passthrough I/O and the right accessed via virtual I/O using a driver in a driver virtual machine.

With hardware that provide safe device access from guest VMs [1, 2], it is possible to grant virtual machines direct access to hardware. With this hardware support, the guest VMs execute device drivers that communicate with the device, bypassing the VMM. With this *passthrough I/O*, performance is close to native, non-virtualized I/O [19]. However, passthrough I/O prevents migration because the VMM has no information about the state of the device, which is controlled by the guest VM. The destination host may have a different passthrough-I/O device or may only use virtual I/O. Because the VMM does not have access to the state of the driver or device, it cannot correctly reconfigure the guest OS to access a different device at the destination host.

Recent work on migration of passthrough-I/O devices relies on the Linux PCI hotplug interface to remove the driver before migration and restart it after migration [21]. This approach maintains connectivity with clients by redirecting them to a virtual network with the Linux bonding driver. However, this approach only works for network devices (as the bonding driver only works for networks) and relies on an additional network interface with client connectivity to maintain service during migration. Our implementation does not require any additional interfaces with client access and can be applied to any class of devices. Intel has proposed additional hardware support for migration of passthrough-I/O devices, but relies on hardware support in the devices and modified guest device drivers to support migration [16]. In contrast, our approach supports migration with unmodified drivers in the guest virtual machine. Furthermore, both of these approaches require running migration code in the guest virtual machine at the source host, which may not be possible in all migration mechanisms.

3 Architecture

The primary goals of migration with shadow drivers are (1) low performance cost when not migrating, (2) minimal downtime during migration, and (3) no assistance from code in the guest virtual before migration. These ensure that migration is not delayed the device driver.

We leverage shadow drivers [14] to capture the state of a passthrough I/O driver in a virtual machine before migration and start a new driver after migration. A shadow driver is a kernel agent that monitors and stores the state of a driver by intercepting function calls between the driver and the kernel. While originally intended to provide transparent recovery from driver failures, shadow drivers provide two critical features for migration. First, shadow drivers continuously capture the state of the driver. After migration, shadow drivers can initialize a new driver and place it in the same state as the pre-migration driver. Second, shadow drivers can clean up and unload a driver without executing any driver code. Thus, for drivers without hotplug support, a shadow driver can unload the driver from the guest virtual machine without executing any driver code, which may malfunction when the device is not present. Figure 2 shows the use of shadow drivers before, during, and after migration. This approach relies on *para-virtualization* [18], as the code in the guest VM participates in virtualization.

During normal operation between migrations, shadow driver *taps* intercept all function calls between the passthrough-I/O driver and the guest OS kernel. In this *passive mode*, the shadow driver records operations that change the state of the driver, such as configuration operations, and tracks packets that are outstanding (*i.e.*, sent but not acknowledged by the device).

When migration occurs, the VMM suspends the virtual machine and copies its state from the source host to the destination host. At the destination, the VMM injects an upcall (or returns from a previous hypercall), notifying the guest OS that migration just occurred. At this point, the guest OS notifies all shadow drivers that a migration occurred.

Immediately after migration, shadow drivers transition into *active mode*, where they perform two functions. First, they proxy for the device driver, fielding requests from the kernel until a new driver has been started. This ensures that the kernel and application software is unaware that the network device has been replaced. Second, shadow drivers unload the existing passthrough-I/O driver and start a driver for the destination host's network device. When starting this driver, the shadow driver uses its log to configure the destination driver similar to the source driver, including injecting outstanding

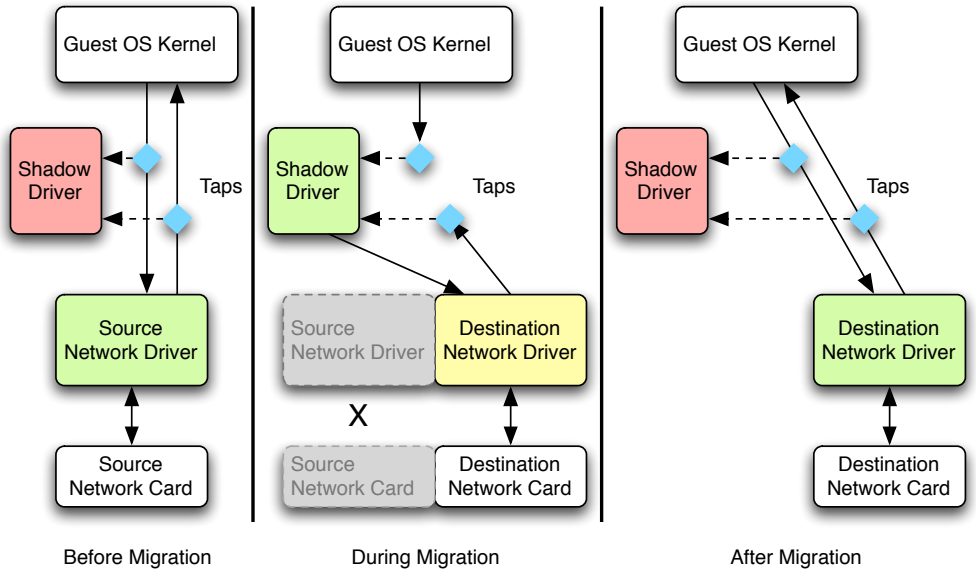

Figure 2: Migration with shadow drivers.

packets that may not have been sent. In addition, the shadow is responsible for ensuring that local switches are aware that the MAC address for the virtual machine has changed or migrated. Once this is complete, the shadow driver transitions back to passive mode, and the network is available for use.

Using shadow drivers for migration provides several benefits over detaching the device before migration and re-attaching it after migration. First, the shadow driver is the agent in the guest VM that configures the driver after migration, ensuring that it is configured properly to its original state and has network connectivity. Second, shadow drivers can reduce the downtime of migration, because we need not detach the driver prior to migration, allowing that to occur at the destination instead.

4 Implementation

We have prototyped shadow driver migration for network devices using Linux as a guest operating system within the Xen hypervisor. Our implementation consists of two bodies of code: changes to the Xen hypervisor to enable migration, and the shadow driver implementation within the guest OS. We made no changes to device drivers.

4.1 Xen

We modified Xen 3.2 to remove restrictions that prohibit migrating virtual machines using passthrough I/O. Because migration was not previously supported in this configuration, there were several places in Xen where code had to be modified to enable this. For example, Xen refused to migrate virtual machines that map device-I/O memory. Furthermore, after migration Xen does not automatically connect passthrough-I/O devices. As a result, migration would fail or the guest would not have network access after migration.

We currently address these problems in Xen by modifying the migration scripts. In dom0, the management domain, we modified migration code to detach the guest virtual machine in domU from the virtual PCI bus prior to migration. Detaching allows Xen to migrate the domain, but as a side effect can cause the guest OS to unload the driver. The shadow driver therefore prevents the network driver from being unloaded and instead disables the device.

We modified the migration code to unmap I/O memory at the source after copying the virtual machine's memory but just prior to suspending the VM. This occurs just before migration, which reduces the network downtime during migration as compared to detaching before copying data. However, the virtual machine state sent to the destination includes the passthrough device. We modified the migration code at the destination to remove references to the source device from the VM configuration before restarting the VM.

We also modified the migration code to re-attach the virtual PCI bus after migration and to create a virtual event channel to notify the migrated virtual machine, after it resumes, that a passthrough-I/O network device is available.

4.2 Shadow Drivers

We ported the existing shadow driver implementation to the Linux 2.6.18.8 kernel and removed code not necessary for migration. The remaining shadow driver code provides object tracking, to enable recovery; taps, to con-

trol communication between the driver and the kernel; and a log to store the state of the passthrough-I/O driver.

4.2.1 Passive Mode

During passive mode, the shadow driver tracks kernel objects in use by the passthrough-I/O driver in a hash table. We implement taps with *wrappers* around all functions in the kernel/driver interface by binding the driver to wrappers at load time and by replacing function pointers with pointers to wrappers at run time. The wrappers invoke the shadow driver after executing the wrapped kernel or driver function.

The shadow driver records information to unload the driver after migration. After each call from the kernel into the driver and each return from the kernel back into the driver, the shadow records the address and type of any kernel data structures passed to the driver and deletes from the table data structures released to the kernel. For example, the shadow records the addresses of `sk_buffs` containing packets when a network driver's `hard_start_xmit` function is invoked. When the driver releases the packet with `dev_kfree_skb_irq`, the shadow driver deletes the `sk_buff` from the hash table. Similarly, the shadow records the addresses and types of all kernel objects allocated by the driver, such as device objects, timers, or I/O resources.

The shadow driver also maintains a small in-memory log of configuration operations, such as calls to set multicast addresses, MAC addresses, or the MTU. This log enables the shadow to restore these configuration settings after migration.

When a migration is initiated, the shadow driver continues to capture the state of the device until the domain is suspended. All recovery of the driver is performed at the destination host. At the source, just before the guest suspends itself, we stop the netdevice and unmap the driver's mapped I/O memory.

4.2.2 Migration

The shadow driver mechanism is invoked after migration to bring up the network driver. After the complete state of the virtual machine has been copied to the destination host, Xen returns from the `suspend` hypercall into the guest OS. We modified the Linux kernel to invoke the shadow driver recovery mechanism just after `suspend` returns successfully (an error indicates that migration failed). The shadow driver then proceeds to (1) unload the existing driver, (2) initialize a new driver, and (3) transfer state to the new driver.

After the guest virtual machine restarts, the shadow driver unloads the old driver using the table of tracked objects. This step is essential because after migration the original device is no longer attached, and the driver may not function properly. Instead, the shadow driver walks its table of kernel objects used by the driver and releases them itself, issuing the same kernel function calls the driver would use.

The shadow then proceeds to initialize the new driver. If the device at the migration destination is the same as the one at the source, the shadow driver restarts the existing driver that is already in memory. The shadow stores a copy of the driver's data section from when it was first loaded to avoid fetching it from disk during migration. If the device is different, necessitating a different driver, the shadow driver pre-loads the new driver module into memory. It then proceeds with the shadow driver replugging mechanism [15], which allows replacing the driver during recovery, to correctly start the new driver.

As the driver reinitializes, it invokes the kernel to register and to acquire resources. The shadow driver interposes on these requests and re-attach the new driver to kernel resources used by the old driver. For example, the new device driver re-uses the existing `net_device` structure, causing it to be connected to the existing network stack. This reduces the time spent reconfiguring the network after migration.

Finally, the shadow driver restores the driver to its state pre-migration by re-applying any associated configuration changes. We borrow code from Xen to make the network reachable by sending an unsolicited ARP reply message from the destination network interface as soon as it is up. In addition, the shadow invokes configuration functions, such as `set_multicast_list` to set multicast addresses, and retransmits any packets that were issued to the device but not acknowledged as sent.

At this stage, the shadow driver reverts to passive mode, and allows the guest OS to execute normally.

4.3 Migration between different devices

Shadow drivers also support live migration between heterogeneous NICs. No additional changes in the guest OS are required. However, the shadow driver must be informed that the different device at the guest is the target for migration, so it correctly transfers the state of the device from the source.

One issue that may arise if that source and destination devices may support different features, such checksum offload. We rely on the replugging support in shadow drivers [15] to smooth the differences. In most cases, including checksum offload, the Linux network stack checks on every packet whether the device supports the feature, so that features may be enabled and disabled safely.

For features that affects kernel data structures and cannot be simply enabled or disabled, the shadow replug mechanism provides two options: if the feature was not present at the source device, it is disabled at the destination device; the shadow driver masks out bits notifying

Using Intel Pro/1000 gigabit NIC

I/O Access Type	Throughput	CPU Utilization
Virtualized I/O	698 Mbits/s	14%
Passthrough I/O	773 Mbits/s	3%
Passthrough I/O with shadow driver	769 Mbits/s	4%

Using NVIDIA MCP55 Pro gigabit NIC

I/O Access Type	Throughput	CPU Utilization
Virtualized I/O	706 Mbits/s	18%
Passthrough I/O	941 Mbits/s	8%
Passthrough I/O with shadow driver	938 Mbits/s	9%

Table 1: TCP streaming performance with netperf for each driver configuration using two different network cards. Each test is the average of five runs.

the kernel of the feature's existence. In the reverse case, when a feature present at the source is missing at the destination, the shadow replug mechanism will fail the recovery. While this should be rare in a managed environment, where all devices are known in advance, shadow drivers support masking features when loading drivers to ensure that only least-common-denominator features are used.

We have successful tested migration between different devices using the Intel Pro/1000 gigabit NIC to an NVIDIA MCP55 Pro gigabit NIC. In addition, the same mechanism supports migration to a virtual driver, so a VM using passthrough-I/O on one host can be migrated to second host with a virtual device.

5 Evaluation

In this section we evaluate our implementation of shadow drivers for its overheads and migration latency on Xen. The following subsections describe the tests carried out for evaluating the overheads of logging due to passive monitoring and the latency of migration introduced due to device migration support.

We performed the tests on machines with a single 2.2GHz AMD Opteron processor in 32-bit mode, 1GB memory, an Intel Pro/1000 gigabit Ethernet NIC (e1000 driver) and an NVIDIA MCP55 Pro gigabit NIC (forcedeth driver). We do most experiments with the Intel NIC configured for passthrough I/O. We use the Xen 3.2 unstable distribution with the linux-2.6.18-xen kernel in para-virtualized mode. We do not use hardware protection against DMA in the guest VM, as recent work shows its cost [19].

5.1 Overhead of shadow logging

Because migration is a rare event, preparing for migration should cause only minimal overhead. In this section, we present measurements of the performance cost of shadow driver taps and logging in passive mode. The cost of shadow drivers is the time spent monitoring driver state during passive mode. We measure the cost of shadow drivers compared to (1) fully virtualized network access, where the device driver runs in Xen's driver domain (dom0), and (2) passthrough I/O access without shadow drivers, where the driver runs in guest domain (domU) and migration is not possible.

We measure performance using netperf [4], and report the bandwidth and CPU utilization in the guest for TCP traffic. The results in Table 1 show throughput and CPU utilization using different I/O access methods for both network cards. Passthrough I/O with shadow drivers for migration has throughput within 1% of passthrough without shadow drivers, and 10-30% better than virtualized I/O. CPU utilization with shadow drivers was one percentage point higher than normal passthrough, and 40-70% lower than virtualized I/O. Based on these results, we conclude that shadow drivers incur a negligible performance overhead.

5.2 Latency of migration

One strength of live migration is the minimal downtime due to migration, often measured in tenths of seconds [3]. This is possible because drivers for all devices at the destination are already running in Dom0 before migration. With passthrough I/O, though, the shadow driver at the destination must unload the previously executing driver and load the new driver before connectivity is restored. As a result, the latency of migration is now affected by the speed with which drivers initialize.

In this section, we compare the duration of network outage using shadow driver migration against Xen's native live migration. We generate a workload against a virtual machine and monitor the network traffic using WireShark [20]. We measure the duration of connectivity loss during migration. We also measured the occurrence of important events during the migration process using timing information generated by `printk` calls.

Based on the monitoring experiments, we observe that the packets from a third host are dropped for 3 to 4 seconds while migration occurs, short enough to allow TCP connections to survive. In contrast, Xen with virtual devices can migrate in less than a second.

We analyzed the causes for the expanded migration time, and show a time line in Figure 3. This figure shows the events between when network connectivity is lost and when it is restored. Several differences to Xen stand out. First, we currently disable network access *before* migration begins, with the PCI unplug operation. Thus, network connectivity is lost while the virtual machine is copied between hosts. This is required because Xen currently prevents migration while PCI devices are attached

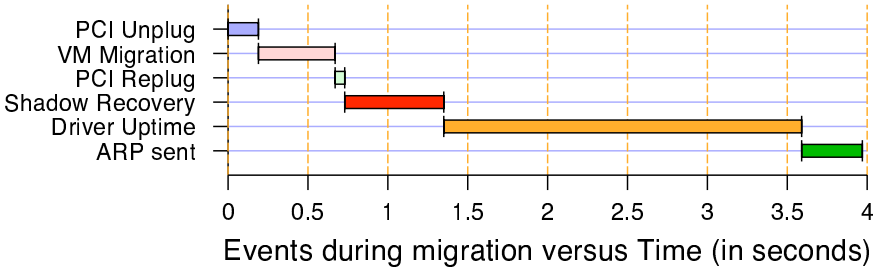

Figure 3: Timing breakdown of events during migration.

to a guest VM; with additional changes to Xen this time could be reduced.

Second, we observe that the majority of the time, over two seconds, is spent waiting for the e1000 driver to come up migration. This time can be reduced only by modifying the driver, for example to implement a fast-restart mode after migration. In addition, device support for per-VM queues may reduce the driver initialization cost within a VM [11]. However, our experience with other drivers suggests that for most devices the driver initialization latency is much shorter.

6 Conclusion

We propose using shadow drivers to migrate the state of passthrough I/O devices within a virtual machine. This design has low complexity and overhead, requires no driver and minimal guest OS modification, and provides low-latency migration. In addition, it supports migrating between passthrough-I/O devices and virtual-I/O devices seamlessly. While we implement this solution for Linux network drivers running over Xen, it can be readily ported to other devices, operating systems, and hypervisors.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) grant CCF-0621487, the UW-Madison Department of Computer Sciences, and donations from Sun Microsystems. Swift has a significant financial interest in Microsoft.

References

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankran, Ionnis Schoinas, Rich Uhlig, Balaji Vembu, and John Weigert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–191, August 2006.

[2] Advanced Micro Devices, Inc. AMD I/O virtualization technology (IOMMU) specification. `www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`, February 2007. Publication # 34434.

[3] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proc. of the 2nd Symp. on Networked Systems Design and Implementation*, May 2005.

[4] Information Networks Division. Netperf: A network performance benchmark. `http://www.netperf.org`.

[5] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, October 2004.

[6] Aravind Menonand, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the 2006 USENIX ATC*, pages 15–28, May 2006.

[7] Mike Neil. Hypervisor, virtualization stack, and device virtualization architectures. Technical Report WinHec 2006 Presentation VIR047, Microsoft Corporation, May 2006.

[8] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[9] Qumranet Inc. KVM: Kernel-based virtualization driver. `www.qumranet.com/wp/kvm_wp.pdf`, 2006.

[10] Quramnet Inc. KVM: Migrating a VM. http://kvm.qumranet.com/kvmwiki/Migration, 2008.

[11] Jose Renato Santos, Yoshio Turner, , G. John Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2008.

[12] Ashley Saulsbury. Your OS on the T1 hypervisor. www.opensparc.net/publications/presentations/your-os-on-the-t1-hypervisor.html, March 2006.

[13] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX ATC*, Boston, Massachusetts, June 2001.

[14] Michael Swift, Muthukaruppan Annamalau, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), November 2006.

[15] Michael M. Swift, Damien Martin-Guillerez, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Live update for device drivers. Technical Report CS-TR-2008-1634, March 2008.

[16] Sean Varley and Howie Xu. I/O pass-through methologies for mainstream virtualization usage. http://intel.wingateweb.com/US08/published/sessions/IOSS003/SF08_IOSS003_101r.pdf, August 2008.

[17] VMware Inc. I/O compatibility guide for ESX server 3.x. http://www.vmware.com/pdf/vi3_io_guide.pdf, June 2007.

[18] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 5th USENIX OSDI*, pages 195–209, Boston, MA, December 2002.

[19] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2008.

[20] WireShark: A network protocol analyzer. http://www.wireshark.org.

[21] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live migration with pass-through device for Linux VM. In *Proceedings of the Ottawa Linux Symposium*, pages 261–267, 2008.