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*, an agent in the guest OS kernel that efficiently captures and restores the state of a device driver. On the source machine, the shadow driver 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 machine. We implement shadow driver migration for Linux network drivers running on the Xen hypervisor. Shadow driver migration requires a migration downtime similar to the driver initialization time, short enough to avoid disrupting active TCP connections. We find that the performance overhead, compared to direct hardware access, is negligible and is much lower than using a virtual NIC.

1 Introduction

Virtualized systems are increasingly being used across deployed production servers, hosting providers, computational grids and data centers [3, 7, 25]. Virtual machine migration, such as VMware VMotion [13] and Xen and KVM Live Migration [15, 4], is a powerful tool for managing hardware and reducing energy consumption. 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 [23], in a driver virtual machine [6, 12, 18], or in the host operating system [14, 19]. 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. This overhead can be significant up to a factor of five for network transmit and receive [11, 17]. 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]. In this mode of device access, device drivers in the guest operating system communicate directly with hardware devices without hypervisor mediation for better I/O throughput. However, direct device access, also called direct I/O or passthrough 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. The problem is further complicated for the case when different devices run at the source and destination, because the state at the source may be irrelevant or incompatible with the device at the destination.

In this paper, we leverage shadow drivers [20, 21] to migrate virtual machines that directly access devices. A shadow driver is a kernel agent that 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 implemented a prototype 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 is an effective at migration providing:

1. **Live Migration**: Low latency migration that does not disrupt TCP connections to the direct-access device being migrated.

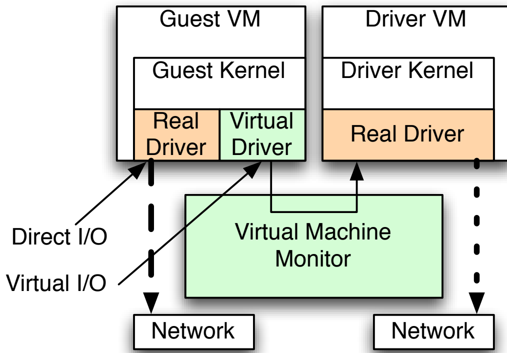

Figure 1: **I/O paths in a virtual machine.**

2. **Transparency**: Applications and services running in the guest VM do not need to perform any migration-specific tasks.

3. **Low overhead**: Shadow drivers have little impact on direct-I/O performance while providing live migration support for network direct-access devices.

4. **Moderate implementation efforts**: Our implementation does not require excessive or widespread changes to the kernel subsystem and requires no changes to the driver.

5. **No residual dependencies at the source**: We cleanly detach the device from the guest operating system at the source and divert all active connections to the destination so that it can be re-used by other guest operating systems post-migration immediately.

In the following section, we provide background on I/O virtualization. Following that, we describe the architecture of shadow driver migration, then the implementation of direct-I/O migration, followed by evaluation, related work and conclusions.

2 I/O Virtualization

Virtual machine monitors and hypervisors must provide guest virtual machines access to hardware devices to perform I/O. For fully virtualized and para-virtualized systems, only a virtual device is exposed to the guest operating system. In case of direct device access the real device is exposed to the guest operating system. These methods of device access have different tradeoffs.

2.1 Overview

With *virtual I/O*, operations in a guest VM are intercepted by the VMM and carried out by either the hypervisor [23], a host operating system [14, 19], or a driver executing in a privileged virtual machine [6, 12, 18]. In the virtual I/O model, the onus of supporting diversity of devices lies where the real drivers execute, i.e. with the host operating system or the hypervisor. The guest operating system is only presented a virtual device and it remains unaffected by the heterogeneity of the underlying hardware. The virtual interface also enables the hypervisor to provide device sharing between multiple guest operating systems.

This model easily supports migration because the hypervisor has access to the virtual device state exposed to the guest operating system. After migration, the hypervisor at the destination offers the same virtual device in the same virtual state. Thus, live migration for virtual devices is easily orchestrated by the hypervisor.

Despite these benefits, virtual I/O reduces performance due to the overhead of mediation since the hypervisor must interpose on all I/O requests, adding additional traps. Shown in Figure 1 is the support needed for virtual I/O; a guest virtual driver in the guest operating system sends its requests to a driver domain, which runs the real driver. These requests are mediated by the hypervisor.

With hardware that provides safe device access from guest VMs [1, 2], it is possible to grant virtual machines direct access to hardware. Guest VMs execute device drivers that communicate with the device, bypassing the VMM. This method of I/O access, known as *direct I/O*, provides performance close to native, non-virtualized I/O [26]. Direct I/O is most suited to hardware devices that require high throughput, such as network devices.

Direct access to hardware can raise security issues, as the driver or the device may use DMA to access memory outside the guest virtual machine or raise too many interrupts. These problems are resolved by using an IOMMU that restricts DMA to physical memory allocated to the virtual machine and that virtualizes interrupts to ensure they are sent to the correct processor. Figure 1 shows direct I/O on the left side in orange, where the real drivers execute in the guest operating system and directly access the device.

Direct I/O, however, 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 direct-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. Furthermore, if the destination host runs a different device, the existing driver running in the migrating guest OS will not work.

To provide live migration support, one needs to capture the state of the driver either in the guest operating system or in the hypervisor and orchestrate migration to correctly reconfigure the device while maintaining the liveness of migration. Maintaining the liveness of migration requires that network connections are not reset either within the guest operating

system or outside and that applications should not fail due to network discontinuity.

2.2 I/O Virtualization in Xen

Device drivers in Xen run outside the hypervisor in a virtual machine. Drivers may execute either in the *dom0* management virtual machine or in a separate driver virtual machine, called a *driver domain*. With the OS kernel of this VM, a *backend driver* receives requests from Xen and passes them to an unmodified device driver, which communicates with the hardware.

Guest operating systems run in unprivileged domains, titled *domU*. Xen provides three options for device virtualization: (1) no virtualization, (2) full virtualization, and (3) and para-virtualization. With full virtualization and no virtualization, guest OSes run unmodified device drivers. For the fully virtualized devices, Xen traps access to virtual hardware registers and translates these requests into calls into the backend driver. While a fully virtualized device can be shared between multiple guests, a non-virtualized device cannot be shared but provides high performance direct I/O.

The most common method of device virtualization is para-virtualization, in which guests operating systems run a virtual device driver, the *frontend driver*, that calls into Xen to communicate with the backend driver. This is more efficient than simulating real a device, as it makes use of shared memory for passing data between virtual machines.

Xen relies on hypervisor mediation to perform migration of I/O devices. Xen 3.2 requires the source and destination to have access to the storage where the guest operating system to be migrated resides. Xen only migrates if the network devices are virtual and source and destination host machines are on the same layer-2 network subnet.

Xen migration consists of the following steps. First, Xen sends the virtual machine state information to the destination and checks for the availability of appropriate resources to run the virtual machine at the destination, such as enough memory. Once these resources are reserved, Xen starts an iterative pre-copy of the pages from the source to the destination. This includes initially sending all pages to the destination followed by sending only the modified pages since the previous send. When only a few pages remain to be copied, Xen suspends the virtual machine at the source, and copies all modified pages. At this point, all network traffic is redirected to the destination physical host by sending a reverse ARP network packet. The source now discards all information about the virtual machine and while the virtual machine is restarted at the destination. All virtual devices are made available to the virtual machine and it continues to run normally.

3 Architecture

The primary goals of migration with shadow drivers are:

1. **Low performance cost when not migrating**: because migration is a relatively rare event, a solution should not incur much overhead when there is no migration happening (the common case).
2. **Minimal downtime during migration**: This is to ensure that the TCP connections are not broken when a migration occurs and hence the migration is *live*.
3. **No assistance from code in the guest before migration**: This is to ensure that migration is not delayed due to execution of a significant amount of code pre-migration.

Shadow driver migration introduces an agent in the guest operating system to manage the migration of direct-access devices. Shadow drivers are class-based: one implementation is needed for each class of devices sharing an interface, such as network devices or sound devices. Shadow drivers interpose on this interface to monitor driver execution [20]. A shadow driver captures the state of a direct-I/O driver in a virtual machine before migration and then starts and configures the appropriate driver after migration with this state.

3.1 Shadow Driver Overview

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. Shadow drivers were originally designed to provide transparent recovery against driver failures. In the occurrence of driver failures, the device driver processes incoming requests and recovers the original driver.

Shadow drivers provide three critical features that are useful for migration of direct-access devices. First, shadow drivers continuously capture the state of the driver by intercepting calls between the kernel and the driver to record calls that configure the driver and to track all kernel objects in use by 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, 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. It then proceeds to correctly configure a new physical or virtual device. Third, shadow drivers proxy for the driver during recovery so that applications in the guest operating system do not observe a discontinuity when a migration occurs.

Figure 2 shows the use of shadow drivers before, during, and after migration. This approach relies on *para-virtualization* [24], as the code in the guest VM participates in virtualization.

3.2 Normal Operation

During normal operation between migrations, shadow driver *taps* intercept all function calls between the direct-I/O driver

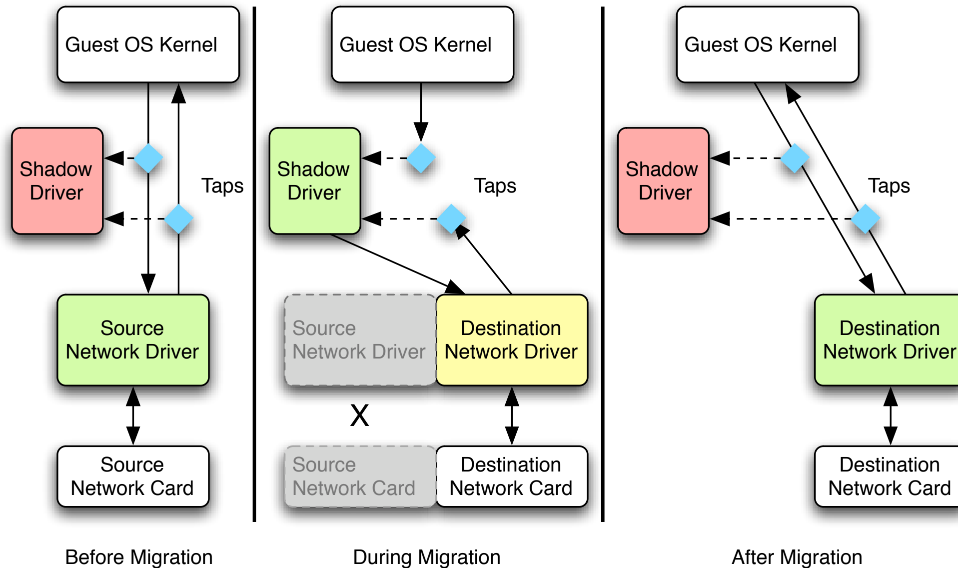

Figure 2: **Migration with shadow drivers.**

and the guest OS kernel. The shadow driver also tracks the return values of these function calls. By doing so, the driver can track all shared objects between the driver and the kernel like device objects, interrupt request lines etc. The shadow driver is also able to track state changes to the driver state like device configuration operations, ioctl requests etc. This mode of operations is referred to as the *passive mode* of operations where the shadow driver silently tracks device state.

3.3 Migration Support

When a migration is initiated, 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 happened. At this point, the guest OS notifies all shadow drivers that a migration has taken place.

Immediately after migration, shadow drivers transition into *active mode*, where they perform three functions. First, they proxy for the device driver, fielding requests from the kernel until a new driver has been started. This proxying ensures that the kernel and application software is unaware that the direct-access device has been replaced. Shadow drivers respond to requests for the device driver's service by returning cached data from a previous operation or by stalling the request until the migration completes.

Second, shadow drivers unload the existing direct-I/O driver. The shadow driver then walks the table of tracked objects in use by the driver and releases any objects not needed after migration. The shadow driver then proceeds to start the corresponding new driver for the destination host's direct-access device. When starting this driver, the shadow driver uses its log to configure the destination driver similar to the source driver, such as replaying configuration calls and injecting outstanding packets that may not have been sent. In addition, the shadow driver ensures that the information about the hardware change is propagated to external dependencies. For network devices, shadow drivers ensure 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 device is available for use.

3.4 Summary

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 device 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. Third, shadow drivers are class drivers and have only one implementation per device class type and are independent of device drivers. As a result, shadow drivers can provide migration support for an entire class of drivers with a single implementation effort.

4 Implementation

We implemented shadow driver migration for network devices using Linux 2.6.18.8 as a para-virtualized guest operating system within the Xen 3.2 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. The bulk of our changes reside in the guest OS, which we modified to enable shadow driver support.

4.1 Xen

We made minimal changes to the Xen hypervisor. These include changes to remove restrictions for direct-access device migration and changes for addition and removal of direct-access device information during migration. We modified Xen 3.2 to remove restrictions that prohibit migrating virtual machines using direct 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 direct-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. Unloading the device breaks the existing TCP connections and also delays the migration since a new device is to be registered post-migration. When the device is detached from the virtual PCI bus, the shadow driver in the guest operating system prevents the network driver from being unloaded and instead disables the device. This occurs just before migration, which reduces the network downtime during migration as compared to detaching before copying data. As a result the virtual machine configuration sent to the destination during the pre-copy stage includes the direct-access device details. We modified the migration code at the destination to remove references to the source direct-access devices by removing configuration information about all devices attached by the virtual PCI bus before restarting the virtual machine.

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 direct-I/O network device is available.

4.2 Shadow Drivers

We ported the existing shadow driver implementation [20] done on Linux 2.4.18 to the Linux 2.6.18.8 kernel. The original shadow driver implementation was designed to work with a fault isolation system. Since, fault isolation is not required for migration, we removed dependencies of the shadow driver code on the isolation system. We also do not create separate protection domains for driver execution, reducing performance overheads caused by switching domains.

The remaining shadow driver code provides object tracking, to enable recovery; taps, to control communication between the driver and the kernel; and a log to store the state of the direct-I/O driver.

4.2.1 Passive Mode

During passive mode, the shadow driver tracks kernel objects in use by the direct-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. This is done by generating a small trampoline function on the fly that sets a per-thread variable and jumps to a common wrapper. 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 driver 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 driver 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. The shadow garbage collects past updates to the same variable and only contains the latest configuration information. As a result, the size of the log remains roughly constant throughout the lifetime of the running driver.

4.2.2 Migration

When the host operating system initiates migration, 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, the shadow driver disables the net-device as previously described and also unmaps the driver's mapped I/O memory. This is done because Xen maps the device I/O memory directly with the physical memory of the host operating system, which prevents migration.

After the complete state of the virtual machine (VM configuration, dirty pages etc.) has been copied to the destination host, the host operating system in Xen does not actually suspend the guest for migration. It instead sends an event to the guest OS, which suspends itself via a hypercall. We invoke the shadow driver mechanism after this hypercall returns successfully at the destination (an error indicates migration failed) to ensure that the shadow driver mechanism is the first thing executed after migration. This method

is safe in the presence of multiple CPUs because the guest kernel shuts down all CPUs except `cpu 0` before migration. After migration, the kernel runs without preemption, which ensures that shadow driver code is executed before any other code.

The guest OS migration code invokes the shadow driver to perform recovery on the device driver. 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 frees anything not needed to restart the driver by issuing the same kernel function calls that the driver would use to deallocate these objects. For example for objects allocated by `kmalloc`, a corresponding `kfree` is issued and so on.

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. We describe what happens if the device is different in the next section.

As the driver reinitializes, it invokes the kernel to register and to acquire resources. The shadow driver interposes on these requests and re-attaches 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 and ensures that applications directly accessing the net device, such as packet filters, do not observe a discontinuity.

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. After migration, the shadow driver loads the new driver module into memory. It then proceeds with the shadow driver replugging mechanism [21], which allows replacing the driver during recovery, to correctly start the new driver.

One issue that may arise if the source and destination devices support different features, such as checksum offload. We rely on the replugging support in shadow drivers [21] 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 packets that are in flight during migration and may depend on features not available at the destination, the shadow driver discards the packets, trusting higher-level protocols to recover from dropped packets.

For features that affect kernel data structures and cannot simply be 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 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 successfully tested migration between different devices using the Intel Pro/1000 gigabit NIC to an NVIDIA MCP55 Pro gigabit NIC. In addition, the same mechanism can support migration to a virtual driver, so a VM using direct-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 also evaluate the implementation cost of shadow drivers as a kernel subsystem.

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 conducted most experiments with the Intel NIC configured for direct 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 [26].

5.1 Overhead of Shadow Logging

Migration is a rare event, so preparing for migration should cause only minimal overhead. In this section, we present

Network Device	I/O Access Type	Throughput	CPU Utilization
Intel Pro/1000 gigabit NIC	Virtualized I/O	698 Mbits/s	14%
	Direct I/O	773 Mbits/s	3%
	Direct I/O with shadow driver	769 Mbits/s	4%
NVIDIA MCP55 Pro gigabit NIC	Virtualized I/O	706 Mbits/s	18%
	Direct I/O	941 Mbits/s	8%
	Direct 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.**

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) direct I/O access without shadow drivers, where the driver runs in guest domain (domU) and migration is not possible.

We measure performance using netperf [5], 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 for transmit traffic. Direct I/O with shadow drivers for migration has throughput within 1% of direct I/O without shadow drivers, and 10-30% better than virtualized I/O. CPU utilization with shadow drivers was one percentage point higher than normal direct I/O, 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 [4]. This is possible because drivers for all devices at the destination are already running in Dom0 before migration and do not require additional configuration during migration. With direct 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 measure the duration of the network outage using shadow driver migration. We generate a workload consisting of ping operations and file transfer application against a virtual machine and monitor the network traffic using WireShark [27]. We measure the duration of connectivity loss during migration. We also measured the occurrence of different steps of migration 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 also performed experiments to perform file copy operations to the migrating virtual machine. We copied a 100MB file to the migrating machine from another host connected over a megabit network. The host operating systems meanwhile were connected over a gigabit network. We find that the file copy application scp, does not abort file transmission when the guest operating system where the file is being copied onto is migrated. The file transfer is stalled for less than 2 seconds starting from the point when the guest is suspended at the source and the file transfer resumes as soon as migration finishes and the local switches are informed about the changed MAC address.

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* actual migration begins, with the PCI unplug operation. Thus, network connectivity would be lost while the virtual machine is copied between hosts. This is required because Xen currently prevents migration while PCI devices are attached to a guest VM; we postpone this detachment to only just before the machine is suspended to minimize the migration downtime.

Second, we re-use the network device interface registered with the kernel. This helps us save time required to register a new network device with the kernel.

Third, we observe that the majority of the time, over two seconds, is spent waiting for the e1000 driver to come up after 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 [17]. However, our experience with other drivers suggests that for most devices the driver initialization latency is much shorter [20].

5.3 Complexity of Implementation

In this section, we show that shadow driver migration can be easily integrated in the kernel with minimal programming effort.

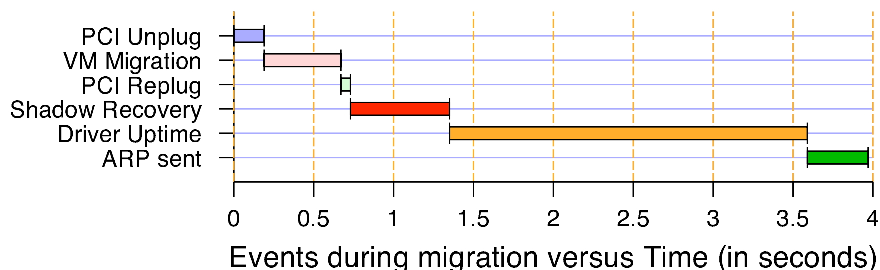

Figure 3: **Timing breakdown of events during migration.**

Our implementation of the kernel subsystem, as described before, consists of wrappers around the kernel driver interface, taps and changes to record/replay the driver state. We have implemented shadow drivers to support all class of drivers but have the wrappers and recovery support implemented only for network drivers. Our code consists of 18KLOC. Of this, approximately, 45% of the code consists of wrappers around the kernel-driver interface, half of which can be automatically generated via scripts.

We argue that our changes are minor when considered in the context of kernel driver code. For example, the `e1000` driver code alone consists of 11KLOC and we add significant functionality for an entire class of drivers at a comparable implementation effort.

6 Related Work

Live virtual machine migration is supported by most commercial virtualization platforms. VMware Workstation, VMware Server, and Xen (2.0 onwards) support the hosted (or split) I/O model [10], while the hypervisor direct I/O model is supported by VMware ESX Server (for storage and network). Direct I/O is supported by Xen and recently by KVM. Live Migration is an important and often-used feature. None of the commercial VM migration techniques support migration for devices performing direct device access [10, 4]. Recently, Xen introduced a patch in Xen 3.3 to unplug direct-access devices, perform migration, and then re-plug a different device at the destination. This migration mechanism is not live and does not maintain any active connections.

There was been significant work in bridging the heterogeneity in migration. Past work investigates performing CPU, memory and I/O migrations across different virtual machine monitors [9]. The approach uses the common ABI for para-virtualization and emulation-based I/O devices.

Recent work on migration of direct-I/O devices relies on the Linux PCI hotplug interface to remove the driver before migration and divert all traffic to the virtual interface and divert it back to physical NIC after migration [28]. This approach maintains connectivity with clients by redirecting them to a virtual network with the Linux bonding driver. Because the bonding driver only supporting network devices, this approach is similarly limited. It also relies on an additional network interface with client connectivity to maintain service during migration. Our implementation does not require any additional or different [16] interfaces with client access and can be applied to any class of devices.

Intel has proposed an additional solution for migration of direct-I/O devices, but it relies on hardware support in the devices and modified guest device drivers to support migration [22]. In contrast, our approach supports migration with unmodified drivers in the guest virtual machine with less than 1% performance overhead. This is especially useful for devices that do not have drivers that can execute inside the hypervisor but still require migration, such as network cards with specialized performance optimizations. Furthermore, both Intel's approach and the hotplug approach require running migration code in the guest virtual machine at the source host, which may not be possible in all migration scenarios.

7 Conclusion

We describe using shadow drivers to migrate the state of direct-access I/O devices within a virtual machine. While we implement shadow driver migration for Linux network drivers running over Xen, it can be readily ported to other devices, operating systems, and hypervisors.

Our design has low complexity and overhead, requires no driver and minimal guest OS modification, and provides low-latency migration. In addition, it can support migrating between direct-I/O devices and virtual-I/O devices seamlessly. Migration time is fast, but not as fast as existing systems using virtualized I/O. A fast-restart mechanism in drivers, to load the driver rapidly after migration, could reduce the driver initialization time, the most significant delay during migration.

Shadow driver migration can also be used for providing additional functionality. Shadow driver migration can be

used to provide support for I/O devices in client space in an Internet Suspend/Resume [8] setup. Shadow driver support in virtual machines can also be used to perform online hot-swap of direct-access devices for fault tolerance. When these devices fail, the shadow driver can transparently failover to another same or different device without any downtime. This cannot be provided by the hypervisor because the hypervisor has no information on the state of the device.

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] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[4] 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.

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

[6] 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.

[7] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 125–134, New York, NY, USA, 2006. ACM.

[8] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, June 2002.

[9] Pengcheng Liu, Ziye Yang, Xiang Song, Yixun Zhou, Haibo Chen, and Binyu Zang. Heterogeneous live migration of virtual machines. In *International Workshop on Virtualization Technology (IWVT'08)*, 2008.

[10] Mallik Mahalingam. I/o architectures for virtualization. In *VMWORLD*, 2006.

[11] 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.

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

[13] 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.

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

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

[16] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 61–70, Washington, DC, USA, 2009.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] Xen Wiki. Xen virtual hosting server providers. http://wiki.xensource.com/xenwiki/VirtualPrivateServerProviders.

[26] 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.

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

[28] 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.