SRVM: Hypervisor Support for Live Migration with Passthrough SR-IOV Network Devices

Xin Xu

VMware Inc
xinxu@vmware.com

Bhavesht Davda

VMware Inc
bhavesht@vmware.com

Abstract

Single-Root I/O Virtualization (SR-IOV) is a specification that allows a single PCI Express (PCIe) device (physical function or PF) to be used as multiple PCIe devices (virtual functions or VF). In a virtualization system, each VF can be directly assigned to a virtual machine (VM) in passthrough mode to significantly improve the network performance. However, VF passthrough mode is not compatible with live migration, which is an essential capability that enables many advanced virtualization features such as high availability and resource provisioning.

To solve this problem, we design SRVM which provides hypervisor support to ensure the VF device can be correctly used by the migrated VM and the applications. SRVM is implemented in the hypervisor without modification in guest operating systems or guest VM drivers. Our experimental results show that SRVM can effectively migrate all memory state, and there is no data loss or corruption in applications after live migration. SRVM does not increase VM downtime. It only costs limited resources (an extra CPU core), and there is no significant runtime overhead in VM network performance. In fact, since the VF can continue to be used during the pre-copy phase, it offers network throughput which is 9.6 times and network latency which is 98% lower compared to other solutions that switch to para-virtualization mode during live migration.

Keywords Virtualization; Live Migration; Passthrough; Virtual Network; SR-IOV

1. Introduction

Networking I/O performance is a crucial aspect when moving applications from bare-metal systems to virtualized en-

vironments, especially for those applications that are not traditionally hosted in virtualization systems. For example, in the High-Performance Computing domain, applications rely on technologies such as InfiniBand [1] and RDMA [2] to achieve sub-microsecond message passing latencies and multiple-Gbps throughput between systems. Similarly, telecommunications operators have specialized packet processing appliances that deal with packet rates in the millions of packets/second with low latency and jitter. Providing high-performance network for those applications in virtualization systems still remains a challenging task.

SR-IOV has been utilized in some commercial products to improve network performance [3, 4]. Figure 1 shows an architecture comparison between para-virtualization mode and SR-IOV in passthrough mode. Note that in this paper we assume a hypervisor contains both the virtual machine monitor as well as the control domain which provides drivers and management tools (as in VMware ESXi).

Figure 1: Virtual Networking Architectures

Currently, para-virtualization mode is widely used for network I/O in virtual environments. As shown in Figure 1 (a), para-virtualized network I/O utilizes a back-end driver (also known as device emulation) in the hypervisor, between the guest VM and the physical device, to manage device sharing. This extra layer adds performance overhead compared to bare-metal machines.

In contrast, SR-IOV allows *self virtualization*, meaning that a single physical function (PF) can appear as multiple virtual functions (VF) in the hypervisor. Each VF can be directly assigned to a VM in passthrough mode as shown in Figure 1 (b). In passthrough mode, the hypervisor does only minimum management work to provision VFs for guest

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

VEE '16, April 2–3, 2016, Atlanta, Georgia, USA..

Copyright is held by the owner/author(s).

ACM 978-1-4503-3947-6/16/04.

<http://dx.doi.org/10.1145/2892242.2892256>

VMs. The VF driver in the guest VM directly configures the VF for packet processing. This configuration greatly reduces I/O overhead compared to para-virtualization.

Figure 2 shows a performance comparison between para-virtualization mode and SR-IOV passthrough-mode. The numbers are normalized with respect to the results from the native machine. The performance is measured in terms of throughput and response time. The experiments are conducted using Intel Niantic 10GbE Ethernet Controller with VMware ESXi. Figure 2 (a) shows L3 packet forwarding rate using DPDK library for various packet sizes. SR-IOV passthrough mode closely matches the native packet forwarding rate. In contrast, the para-virtualization mode (VMware VMXNET3 driver) can only achieve 16.2% to 49.7% of native throughput. Figure 2 (b) shows the minimum, average and maximum latencies. The average latency of SR-IOV passthrough mode is only 13% higher than the native machine. In contrast, the latency of the para-virtualization mode is 107.7% higher than the native machine. The results demonstrate that SR-IOV passthrough mode significantly outperforms para-virtualization mode. Figure 2 (b) shows the SR-IOV normalized maximum latency (66.2%) is lower than the SR-IOV normalized average latency (113%), which is correct but counter-intuitive. This is because of normalization with respect to the results from the native machine. The SR-IOV absolute maximum latency is higher than the SR-IOV absolute average latency.

However, SR-IOV is not a perfect solution. A well-known limitation is that VMs with passthrough devices are not compatible with live migration, which seamlessly moves a running VM from one host machine to another. This is because the hypervisor cannot easily save and restore the passthrough device state as it does for virtual devices in the para-virtualization mode due to two factors. First, saving hardware state is difficult since the hypervisor does not know the entire state of the passthrough device. Most of the device state in data path, such as ring registers, is directly managed by the VF driver in the guest VM. And, second, restoring hardware state is difficult because the hypervisor cannot simply over-write the hardware state as it does to the software state. Some hardware state cannot be programmed by the hypervisor, and programming certain hardware state may trigger side-effects. For example, writing any value to the transmit ring tail register will cause packet transmission.

Another important reason why SR-IOV is not compatible with live migration is that the hypervisor cannot easily track the VM memory modified by the passthrough devices. Therefore, if the VF is receiving packets during live migration, the received data may not be migrated to the destination host, and the application will suffer from data loss.

Previous research addresses this issue mainly using two approaches: migrating hardware state [5, 6]) or migrating VMs without passthrough devices [7, 8]. The approach of migrating hardware state tries to do transitional checkpoint-

Figure 2: Performance Comparison of Virtualization Network

and-restore to migrate as much hardware state as possible. For hardware state that cannot be checkpointed or saved, this approach utilizes an emulation layer to emulate them. It also requires major modifications to the guest OS or drivers. It is therefore difficult to adopt.

The other approach detaches the passthrough device before live migration, so VMs are migrated without passthrough devices. This approach generally relies on bonding two network adapters, one para-virtualized network adapter and one passthrough adapter, using a special driver (e.g., Linux bonding driver [9] or Microsoft NetVSC [8]). Those bonding drivers are OS specific, e.g. Microsoft Hyper-V only supports live migration with passthrough devices in Windows guest VMs. Also, the passthrough device is not available during live migration, which may affect network throughput. For large sized VMs, which may take a long time to migrate (tens of minutes), this performance degradation should be avoided, if possible. Moreover, some applications (e.g., Network Function Virtualization or NFV) commonly utilize Data Plane Development Kits (DPDK) to achieve high performance networking. DPDK requires direct hardware access to a single network interface, and therefore it is not compatible with bonding driver.

In this paper, we describe the design and implementation of SRVM, which provides hypervisor support to enable live

migration for SR-IOV passthrough network devices. SRVM does not require modification in the guest driver or kernel, and allows the device to be used during live migration. The key observation is that it is not necessary to do traditional checkpoint and restore for passthrough devices. We only need to guarantee correct functioning of the device so that high-performance passthrough network activity can be maintained during and after live migration.

There are three technical challenges to design and implement SRVM. First, during live migration, the application that is receiving network packets may suffer from data loss, if received data are not properly handled. This is because received packets are placed in VM memory through direct memory access (DMA). DMA is directly issued from a VF device to guest VM memory, assisted by hardware IOMMU (e.g. Intel VT-d [10]). The hypervisor does not know which pages are modified by DMA, and therefore won't explicitly migrate modified VM memory.

Second, after live migration, the destination host must provision a new VF for the migrated VM. But the guest VF driver still has the original state. The hypervisor must make the new VF accessible to the guest VM and synchronize state between the guest VF driver and the new VF.

Third, SR-IOV can be used with user-level poll mode drivers (such as DPDK) to achieve high-throughput and low-latency networking. It is important that the proposed techniques work not only with traditional guest OS drivers (such as Intel IXGBEVF), but also with DPDK.

SRVM is designed to solve these problems. Specifically, we make three contributions in this paper:

- We develop a new technique to provide hypervisor support for tracking dirty memory pages. To the best of our knowledge, this is the first solution to dirty memory tracking without modification in the guest driver or OS kernel. It explicitly tracks and migrates the modified VM memory that may otherwise be lost after live migration.
- We also provide hypervisor support to provision VFs after live migration, which does not require modification in guest OS or driver. SRVM only saves a minimum set of the control path data (e.g., PCI configuration and MSI-X interrupt table), and restores it to a new VF in the destination host so that the guest VF driver can access the device and synchronize device state after live migration.
- We design and implement our techniques in both the Intel IXGBEVF driver and the DPDK user-level poll mode driver. To the best of our knowledge, this method is the first attempt to support user-level poll-mode driver.

We also conduct thorough evaluation to demonstrate the effectiveness and the performance overhead of our approach. This paper is organized as follows. Section 2 provides the background information on live migration and SR-IOV. Section 3 discusses existing work and explains the advantages of SRVM. Section 4 explains SRVM design details and imple-

mentation issues. Section 5 shows the evaluation results of SRVM in terms of effectiveness and performance overhead. Section 6 concludes the paper.

2. Background and Motivation

In this section, we explain the basics of live migration and SR-IOV device, and then explain the reason why live migration is not compatible with device passthrough mode.

2.1 Live Migration

Live migration is an important virtualization feature [11]. It allows a VM to be migrated from one machine to another while the VM is running. VM data to be migrated include CPU state, memory, and other virtual devices state such as virtual network devices. Live migration typically consists of the following steps [12].

Mark memory pages. In the beginning of live migration, the VM will be briefly paused. All memory pages of this VM will be marked as dirty, and will be migrated later.

Pre-copy. After all memory pages are marked, the pre-copy phase starts. Pre-copy iteratively scans the entire VM memory and migrate pages that have been marked as dirty. During the pre-copy phase, the VM is running. A page that has been migrated may be modified again (dirtied). The hypervisor needs to trap those modification and mark those pages as dirty, to make sure they are migrated. Tracking dirty pages can be done with hardware support (e.g., Intel EPT [13]) or without hardware support (e.g., shadow page table).

Pre-copy can have several iterations until only a small number of memory pages are left or the specified criteria are met (e.g., maximum time is reached). The pre-copy phase may be time-consuming (tens of minutes) if VM memory is large or the VM has memory-intensive workloads [12].

Checkpoint. After the pre-copy completes, the hypervisor will pause VM again and checkpoint the virtual device state and the virtual CPU state.

Switch-over. The checkpoint data and the remaining dirtied pages that are not copied by the pre-copy phase will be transferred to the destination.

Resume VM. After all data are transferred, the checkpoint data will be restored to the VM in the destination host. The VM is then resumed on the destination host.

2.2 SR-IOV Network Devices

SR-IOV network devices has been used in platforms requiring high performance network I/O [3]. A SR-IOV device can be used in full-function mode or in SR-IOV mode. When used in full-function mode, the device will be used as a single interface like other regular network devices. When used in SR-IOV mode, the hypervisor can configure the device to appear as multiple VFs in the host. For example, the Intel 82599 (codename Niantic) 10GB network controller supports a maximum of 63 VFs for each PF [14]. Kernel-level VF drivers (e.g., Intel IXGBEVF [15]) have already been in-

Figure 3: Packet Delivery Paths

tegrated into current OS so that the guest VM can easily use a VF in passthrough mode to achieve high performance.

Intel’s Data Plane Development Kit (DPDK) [16] is a user-level driver library that also supports SR-IOV VF passthrough mode. This software framework allows network applications to process packets in the user-level bypassing the kernel network stack. The network performance can be further improved by using DPDK with passthrough VF.

2.3 Live Migration with SR-IOV Passthrough VF

While SR-IOV provides high performance, the hypervisor does not have full control over the VFs in passthrough mode. The hypervisor only handles the control path, such as setting up PCI configuration, interrupt delivery, and memory mapped I/O (MMIO). The guest VF driver (Intel IXGBEVF or DPDK) directly manages the data path, such as setting up Tx/Rx rings and allocating data buffers. During packet transmission, the VF will directly access guest VM memory without assistance from the hypervisor. This causes a problem in the pre-copy phase, the checkpoint phase and the resume phase during live migration.

To better explain this problem, we compare the packet delivery paths of para-virtualization, SR-IOV passthrough with Intel IXGBEVF, and SR-IOV passthrough with DPDK user-level library, as shown in Figure 3. In para-virtualization mode, the back-end driver and hypervisor configure the packet delivery path of the physical device. Specifically, the ring buffers and packet buffers that are used to send and receive packets through the physical network interface are allocated by the hypervisor (through the back-end driver). The hypervisor knows exactly when and where these packets are allocated or freed. Taking receiving packets as an example, the physical device writes all received packets to the packet buffers in the hypervisor. Then, the packet data is demultiplexed by a virtual switch in the hypervisor to determine which guest VM they should be delivered to. The back-end driver then delivers the packets to the guest VM.

The front-end driver in the guest VM thus only interacts with the back-end driver, rather than the actual physical device.

This configuration gives the hypervisor full control over the guest VM network. During the live migration pre-copy phase, the hypervisor can know which packets have been received, and can explicitly migrate them to ensure the destination host receives the up-to-date data. The hypervisor can also save and restore guest VM network configuration for since they are essentially state captured in the VM memory.

This is not the case in SR-IOV passthrough mode. Figure 3 (b) shows the architecture when Intel IXGBEVF is used in the guest VM. The packet processing path is moved up from the hypervisor to the guest VM. The guest driver directly configures the Tx/Rx rings and allocates the memory buffers within its own guest VM memory regions. The device can directly access these guest memory buffers through DMA. The only hypervisor intervention required is for interrupt delivery, assuming the VF is configured to deliver interrupts for I/O completion and other events. If the VF is used in poll mode as with DPDK, the hypervisor cannot even know when the packet is sent or received. Figure 3 (c) shows the architecture when a user-level poll mode driver (e.g. DPDK) is used. The packet delivery path is further moved up into the user-level. The packet buffers are managed and allocated in the user level memory regions within the guest VM.

In SR-IOV passthrough mode, the guest VF driver directly configures the VF for sending and receiving packets. For example, the guest VF driver should allocate memory buffers for packets and should configure the VF to access these buffers. DMA can only access memory through machine memory addresses, while the guest VF driver only provides guest physical addresses. Translation between guest physical addresses and machine addresses is done through a hardware IOMMU (e.g. Intel VT-d). The hypervisor only does a one-time configuration of IOMMU for VMs.

This SR-IOV passthrough mode (for both SR-IOV with Intel IXGBEVF driver and SR-IOV with DPDK driver)

causes two problems for live migration. The first problem is that in the pre-copy phase, the hypervisor is not able to track the pages that have been modified by DMA, because the address translation for DMA is done by hardware IOMMU not the hypervisor. If a page has already been migrated and is then modified by DMA, it will not be migrated again. As a result, the application may end up using incorrect data after live migration.

Another issue comes from restoring passthrough VF state in the checkpoint phase. In general, virtual device state can be easily saved and restored because it is essentially software state. But a passthrough VF is a hardware device that is not entirely configured by the hypervisor. Some hardware state (e.g., stats registers) are read-only, and they simply cannot be programmed by the hypervisor. Moreover, writing to some registers may cause side effects. For example, writing any value to the tail register of the transmit ring buffer will trigger packet transmission on the device. Therefore, restoring VF state is not as simple as restoring virtual device state.

Because of these two problems, current hypervisors either disable live migration for passthrough devices [17–19] or prevent using passthrough devices during live migration [8].

3. Related Work

Efforts have been made in both research and industrial communities to resolve this issue. Existing solutions are typically based on one of three techniques: 1) NIC bonding; 2) Shadow driver; 3) VF driver support. NIC bonding migrates VMs without the passthrough device. The shadow driver approach and the VF driver support approach migrate VMs by migrating the hardware state. A comparison of these techniques with SRVM is shown in Table 1. We discuss each technique and solutions below.

NIC bonding is a technique that aggregates multiple network interfaces into a single interface through a special driver such as Linux ethernet bonding driver [9]. A VF can be bonded with another virtual network interface that is compatible with live migration. Before live migration starts, the VF can be detached from the VM and traffic will go through the virtual network interface. In this way, the network connectivity is preserved during live migration. The VF is not available during live migration, so there is no dirty memory page issue. A Linux bonding driver is used with a hotplug mechanism to provide live migration capability for Linux VMs with SR-IOV passthrough devices [7]. Microsoft Hyper-V also uses a similar technique in which removal of the VF interface is done by a Windows plug-in, the Microsoft Network Virtual Service Client (NetVSC). However, current bonding drivers are designed for OS kernel drivers. For user-level driver libraries like DPDK, which is not managed by the OS kernel, it is not clear how to utilize bonding drivers for live migration. DPDK is likely used in NFV workload because of its high performance. It is critical to provide support for DPDK. Also, in this approach the high-

performance data path of SR-IOV is not available during the pre-copy phase. For large-sized VMs, which may take tens of minutes to migrate, the network performance degradation is significant.

Research work has been proposed to enable live migration of SR-IOV passthrough device using a shadow driver [5]. As shadow driver is an extra layer in the guest OS between the kernel and driver, that monitors the driver state and operations. After live migration, the shadow driver can restore the driver state on the destination host and can emulate the state that cannot be migrated. Marking dirty pages is not explicitly discussed in the paper. While this approach supports live migration between hosts with different types of passthrough device, it requires major modifications in the guest OS, which makes it difficult to utilize.

Guest VF driver support has been proposed to enable live migration to track dirty memory pages and to handle VF migration [6, 20]. Both use a similar technique to track dirty page, by explicitly writing dummy data into each received page so that the page modification can be captured by the hypervisor. The advantage of this approach is that it can be easily implemented in the guest VF driver. But this involves memory access and memory address translation in each received packet, which may significantly affect network performance. Many network applications that do not even access the packet data, such as bridging, may suffer performance degradation because of this approach. This is desirable since the most important advantage of SR-IOV passthrough is high performance. Also, this approach may cause security concerns since the guest VF driver explicitly modifies every received network packet. To handle VF migration, CompSC tries to migrate the entire VF state, which requires an emulation layer for the state that cannot be migrated (e.g. stats and ring registers) [6]; whereas in another work [20], they only migrate partial state by suspending and resuming the VF from the VF driver.

In general, providing guest VF driver support is less complicated to design than hypervisor support because the VF driver directly controls the data path. But it is not easy to adopt because users are required to replace original drivers with the modified version. We believe that hypervisor support for live migration of SR-IOV passthrough devices is necessary because it is easier for users to leverage this feature. Our solution requires no modification to the guest OS or drivers. It does not require any emulation layer after live migration, and allows the VF device to be used during the live migration pre-copy phase. The detailed design of SRVM is explained in the following section.

4. SRVM Design

Our goal is to provide hypervisor support for live migration of SR-IOV passthrough devices instead of modifying the guest OS or VF drivers. SRVM mainly consists of two components: the dirty memory tracking module and the SR-IOV

Table 1. Related Works Comparison with SRVM

	No Guest OS Modification	VF is Available during Live Migration	No Need to Access Packet Data	DPDK Support
NIC Bonding [7, 8]	Yes	No	Yes	No
Shadow Driver [5]	No	Yes	Yes	No*
VF Driver Support [6, 20]	No	Yes	No	No*
Hypervisor Support (SRVM)	Yes	Yes	Yes	Yes

* The approach may be made to work with DPDK driver library with major modifications, but currently there is no such support

VF checkpoint module. The architecture of our approach is illustrated in Figure 4.

Figure 4: SRVM Design Framework

The dirty memory tracking module is enabled at the beginning of the pre-copy phase. It leverages the SR-IOV PF driver in the hypervisor to inspect the VF state, to obtain the addresses of received packets. Then, it sends these addresses to the live migration module for migration later on. The dirty memory tracking module ensures that all received data can be migrated, and it stops when the pre-copy phase is over.

Next, the SR-IOV VF checkpoint module starts to save the control path state of the VF (e.g., MMIO and the MSI-X interrupt vector table). At the destination host, this module provisions a new VF for the migration VM and restores the saved state to the control path of the newly provisioned VF. This module ensures that the guest VF driver can access the new VF in the destination host as if it were accessing the original VF. In the following, we discuss the design and major implementation issues for each module in detail.

4.1 Dirty Memory Tracking

Lifetime of Memory Tracking Module. The memory tracking module is only needed when live migration is started. It can be enabled by the live migration module at the beginning of the pre-copy phase and disabled after the pre-copy is done. This implementation means that there is no overhead during normal execution of the VM.

Tracking dirty memory cannot be easily done in the hypervisor because the hypervisor does not configure the data path of the passthrough VF. But the VF hardware state is stored in the PF hardware device, so we can inspect the VF state through the PF driver instead of using the VF driver. The next step is to determine which state to inspect to obtain the dirty memory information.

Obtaining Ring Information. Network I/O devices generally use ring structures for packet reception and transmission. Fig. 5 shows how a ring is used for receiving packets for a VF device. A ring is created in guest VM memory when the VF driver initializes the VF. The ring contains a limited number of entries (e.g., 512), and each entry specifies a packet data address. The network device finds the packet data through the addresses in the ring, and reads (writes) data via DMA for transmission (reception). To track dirty memory pages, we need to obtain the ring location and inspect the entries in the ring. A key insight is that only the receive ring is required because memory pages are only dirtied when VM is receiving packets (but not sending packets).

Figure 5: VF Receiving Ring

The ring information such as the ring base address and length is stored in several specific registers in the VF device. The PF driver can access those registers [14] and our dirty memory tracking module interacts with the PF driver to obtain this information. Note that this step should be done at the beginning of the pre-copy and it only needs to be done once during the entire live migration, because the ring information will not change until the VF device is reconfigured. If the VF is reset, we need to obtain the ring information again.

The resetting event will be handled by PF driver, and therefore the PF driver and the hypervisor can notify the module to refresh the ring information.

Introspecting Ring Memory. Once the receiving ring information is obtained, we need to examine the ring to find newly updated descriptors, which contain the packet addresses of received packets. We only need to obtain the packet addresses for migration, and we do not need to access the actual packet data. Therefore, our module only inspects a small piece of guest memory (approximately 2KB).

Packets can be received in two modes: interrupt mode or poll mode. The design of inspecting ring memory may vary depending on the receive mode. Interrupt mode is usually the default mode of the VF driver (e.g. Intel IXGBEVF). Packet arrival is notified by a hardware interrupt from VF. This interrupt will be delivered to the guest VM by the hypervisor. Therefore, we can use this interrupt delivery as the signal to activate the memory tracking module to inspect the ring. When there is no interrupt delivery, the dirty memory tracking module can be put to sleep to minimize resource utilization.

In poll mode, the VF driver will constantly examine the VF device to see if there are any new packet, and no receiving interrupt will be generated. This mode is used for high-performance and low-latency applications (e.g., DPDK user-level library). Interrupt-based inspection will not work with poll mode drivers, since packet arrival is not notified by an interrupt. To work with poll mode drivers, our module can also work in poll mode. That is, our module can constantly examine the ring for new packets instead of waiting for an interrupt. This approach will increase resource usage, but requiring this extra resource usage is reasonable because poll mode drivers like DPDK also sacrifice similar overheads by polling to achieve high-performance and low-latency networking. Also, the overhead is only incurred by the memory track module during live migration. For these two reasons, we use poll mode dirty page tracking in the current implementation.

Lockless Introspection. Ring memory is accessed by both the VF driver and our dirty memory tracking module and the VF driver is not aware that there is another module constantly reading the ring memory. A lock between the hypervisor and the VF driver might be used to prevent any potential race conditions. But a locking mechanism requires modifications in the VF guest driver. It also slows down packet processing and may even cause packet drops. Therefore, using a lock is not feasible, and we choose to use lockless inspection instead.

In fact, our module only does read access. As long as our module can process ring memory faster than VF packet processing, using a lock is not necessary. In fact, the inspection can be done very fast because it essentially just reads ring memory to obtain an address and then changes the corresponding bit in a bitmap. In Section 5.4, we evaluate the time

spent in our module and VF packet processing to show that we can ensure our module processes faster than the packet arrival rate. Based on our experiments, SRVM performance is impacted by descheduling and interrupts. However, by assigning a dedicated core to SRVM and migrating interrupts out from the dedicated core during the pre-copy stage, this impact can be mitigated. Other sources of variance can be disabled by disabling power management and setting the host BIOS to maximum performance. We empirically found the above configurations can ensure SRVM processes data faster than the VF receives packets.

However, if the system is not properly configured, our module may be slowed down. Hardware support may be designed for such case to detect if SRVM misses any received packets. For example, a page fault behavior is described in PCI(e) address translation service specification [21]. PCI(e) devices and hardware IOMMU can leverage this page fault behavior to allow the hypervisor to track all the dirtied memory pages (by DMA) by raising a page fault. Current PCI(e) devices and hardware IOMMU do not support such page fault behavior, and future research and development effort is required. Note that even if this page fault behavior is supported by hardware in future, it alone cannot solve the dirty memory tracking problem while maintaining high performance. Because triggering a page fault for every received packet is likely too expensive, and the network throughput will be significantly affected. But IOMMU page fault may help us improve SRVM in other ways such as reducing required CPU resources for tracking dirty pages. With this hardware support, we may not need a dedicated CPU core for ring inspection. We will investigate this in our future work.

Also, two techniques can be employed to improve the processing speed of our module. One potential solution is deliberately slowing down the live migration VM if a high VF receiving rate is observed. This same approach has been used in VMware ESX when the memory dirty rate is too high during live migration [12]. Another potential solution is using multiple threads to process ring memory in parallel. Each thread can be assigned to a CPU core exclusively, which can further improve the processing performance. Since our evaluation shows that current single thread implementation can already process the ring fast enough, we use this single-thread implementation in the paper.

Network devices can be configured with multiple queues. SRVM currently uses one core per queue. Multiple queues would require multiple cores. SRVM can obtain VF configuration through PF driver to determine the required CPU cores. It is possible that SRVM cannot find an idle core for certain configurations (e.g. migrating too many VMs with VFs simultaneously). But such cases can be reliably detected and handled (e.g., alert users) by the hypervisor or management software, which manage live migration and its required CPU resources.

Introspect Ring Descriptor. A ring descriptor contains the address of packet data. Our module just needs to read the descriptor to identify the address. Our implementation is based on the Intel 82599 10GbB NIC card [14]. Its ring descriptor has two different formats: advanced descriptor format and legacy descriptor format. By examining the VF register, we can identify the currently used format. With the legacy format, the addresses of header and payload will be stored in the descriptor until the descriptor is recycled and over-written with a new descriptor by the VF driver. In this case, our module can directly read the descriptor to obtain the addresses.

The advanced descriptor format is different. The addresses are stored in the descriptor before the packet data is received. Once the packet data is received, the hardware will immediately over-write the descriptor with meta-data. In this mode, our module obtains the packet addresses of the allocated packet buffers that have not been written with new data through DMA, and marks them dirty. That is, our module is *pre-processing* the packet addresses before new data are written instead of *post-processing*. So our module does not rely the head and tail pointers for inspection. This is beneficial because accessing those device registers may slow down the inspection speed. In fact, this does not have any adverse impact on the correctness of the migrated VM or applications, because those allocated buffers eventually need to be migrated if they are filled with new received data in the future.

4.2 SR-IOV VF checkpoint

After live migration, the original passthrough VF is gone, and the guest VM has to use a newly provisioned VF on the destination host. However, the guest VM does not know about this change, and it will still try to access the VF with the original configuration. The access may fail without special handling in the hypervisor. Our SR-IOV VF checkpoint module must properly insert the new VF to solve two problems: 1) the control path of the new VF must be properly configured so that the VM can continue accessing this VF device through its original configuration; 2) synchronize the data path configuration state between the new VF and the guest VF driver so that the device can properly process packets.

Checkpoint the control path configuration. In general, PCI devices are first configured by the OS before they can be accessed by drivers. In the virtualization system, SR-IOV devices are configured by the hypervisor before they can be used by the guest VM in passthrough mode. The control path configuration mainly involves PCI(e) configurations (e.g., MMIO configuration) and interrupt delivery.

PCI(e) configuration is stored in a set of standard device registers in each PCI(e) device, no matter what type of PCI(e) devices they are. The OS or the hypervisor follows standard steps to configure the device [22], such as configuring memory mapped I/O (MMIO) and enabling the device.

After this configuration is done, the driver can access the device through MMIO. In our case, to allow the VF driver to access the new VF in the destination host, the MMIO configuration must be saved and restored.

In native machines, MMIO can be configured by the BIOS or OS. Mapped MMIO addresses will be stored in PCI base address registers (BARs). In current VMware ESXi, a virtual MMIO of SR-IOV passthrough VF is exposed to the guest VM. This virtual MMIO can be leveraged to checkpoint the control path configuration. The virtual MMIO design in the VMware ESXi is illustrated in Figure 6.

Figure 6: Virtual MMIO Mapping

The guest VM only sees a virtual MMIO, and the hypervisor manages the mapping between the physical MMIO of a VF and a virtual MMIO. After live migration, the guest VM will try to use the same virtual MMIO to access the VF. But the hypervisor on the new host will associate a new virtual MMIO with the newly provisioned VF. The consequence is that the guest VF driver cannot access the VF anymore, even though a new VF has been provisioned.

To allow the guest VM to access this new VF in the destination, the hypervisor needs to save and restore the original virtual MMIO. The virtual MMIO configuration can be obtained from the VF device through the VF's BARs. The design of the virtual MMIO greatly improves the flexibility of provisioning VFs. For example, the VF ID in the destination host and the source host can be different. If the VF ID in the source host has been taken by other VMs, we can provision any other available VFs for the migrated VM in the destination VMs.

Interrupt delivery should also be provisioned in the control path. The hypervisor is required to configure interrupt delivery for the VF even if it is in passthrough mode. Specifically, our module saves and restores the interrupt vector table (e.g., MSI-X in the Intel 82599 10GB NIC). The hypervisor then configures the interrupt delivery path according to the restored interrupt vector table to ensure that interrupts can be delivered to the correct VM. In this way, the migrated VM will be able to access the new VF through the original VF driver state. Configuring interrupt delivery is a one-time setup only when the VM is first powered on or migrated. The hypervisor is not involved in the MMIO accesses from guest VMs. SRVM does not rely on or change the VF interrupt de-

livery path, so it can work with DPDK which does not use interrupt to notify packet arrival.

Note that our approach requires that the destination host has the same SR-IOV device. This is the same requirement for using the VF driver support approach [6, 20]. To automate the live migration, modifications can be made in the virtualization management software (such as VMware vCenter) to automatically select a pool of available destination hosts with the same SR-IOV device. We plan to implement this feature as part of our future work.

Reset VF. At this point, the VF device will not correctly function because the data paths are not synchronized. That is, the new device has a clean state, but the guest VM driver still carries the old state of the original VF. To synchronize the data path, we set the VF device to a reset state from the hypervisor. This can be detected by the VF driver, which will then clean the driver state, allowing the device to function correctly. Note that by its nature, DPDK applications need to initialize and configure VFs through DPDK. To correctly handle VF reset for live migration, the application should include the status checking and resetting functions from the DPDK library. These are common routines that a robust DPDK application should do and can be easily implemented with unmodified driver library.

Note that our implementation is based on VMware ESXi and the Intel 82599 10GB NIC, but the design of SRVM essentially relies on PCI-SIG standard mechanisms such as SR-IOV VF interfaces and MMIO registers mapped via BARs. The concept of invoking a device-specific callback for tracking dirty pages can be applied to different I/O devices from different vendors for different technologies including storage (e.g. SAS controllers), networking, InfiniBand, etc. SRVM only needs to define high-level APIs to obtain information for live migration, e.g., dirtied page number. We plan to further investigate supporting these devices for future work. Other hypervisors such as KVM can leverage SRVM because they use similar live migration techniques and SR-IOV PF/VF driver interfaces.

5. Evaluation

We conduct a series of experiments to evaluate SRVM in terms of both effectiveness and performance impact. Essentially, we would like to use experiments to answer these questions:

- Can SRVM successfully track dirty memory pages?
- What is the downtime of live migration when using SRVM?
- What is the performance overhead of SRVM?

5.1 Experimental Setup

The experiment setup is as shown in Figure 7. We use two Dell PowerEdge T320 servers as the migration source and destination hosts. Each host has 24GB memory, one six-

core Intel Xeon E5-2440 CPU, and one two-port Intel 82599 10GB NIC. VMs are configured with 4 VCPU, 8GB memory, one VMXNET3 network interface (for live migration and management) and one VF passthrough network interface for application traffic. Both VMs are running RedHat Enterprise Linux Workstation 7.0. The migration VM on the source is used for live migration, and the testing VM on the destination host is used to generate network traffic and to measure downtime. The migration VM and the testing VM are connected through passthrough VFs. If our approach is effective, the testing VM should be able to connect to the migration VM through VFs before, during and after live migration (with downtime). Our prototype is implemented based on VMware ESXi.

For each experiment, we test two drivers, the Intel IXGBEVF driver (shipped with the OS) and DPDK 2.1.0 [16]. The Intel IXGBEVF driver is configured in interrupt mode by default. We use DPDK Pktgen 2.9.5 [23] to generate high packet rates. We also develop simple packet sending/receiving applications for DPDK and IXGBEVF for testing. The applications can send packets with data in specified formats so that we can easily verify if there are any missing packets.

Figure 7: Experimental Setup

5.2 Packet Missing

Without SRVM, the packets that have been received during live migration may not be migrated to the destination. The consequence is that the application cannot know if its data are correct after live migration. To show the effectiveness of SRVM, we conduct experiments to compare those received packet buffers before and after live migration.

In the testing VM, we write a simple application to send packets from the test VM to the live migration VM. The packet is sent with a special format with an incrementing sequence number so that we can easily identify each packet.

For the DPDK driver, we allocate a large memory region containing 4KB buffers. This memory region is initialized with zero content and the DPDK application receives data directly into this memory buffer. The application does not update any data in the packet after the packet is received so

the hypervisor won't consider them as dirty pages without SRVM. We compare the memory content before and after live migration. If there are no missing packets, the content should be the same. Any missing packet will show zero content in its buffer after live migration, whereas the same buffer before live migration should have data with a correct sequence number. An example is shown in Figure 8.

Figure 8: Missing Packet

We use the same test application for Intel IXGBEVF. To examine the received packets, we made modification to the driver in the related receiving functions. We examine the packet buffers in the receiving ring and also set the buffer size to 4KB. We compare the data in these buffers before and after live migration. If there are packets missing, the data in the same packet buffer are different. Note that this driver modification is solely for testing purposes, so that we can clearly see the missing packets. The memory tracking module does not require modifications to the Intel IXGBEVF driver.

To collect enough packets to compare, we run a simple application in the live migration VM that continuously writes dummy data into random memory pages. Running this application will increase the pre-copy phase of live migration to about two minutes, which is reasonable for our experiment. We repeat this experiment ten times. We always see packets missing when SRVM is not enabled and no packet missing with SRVM. As an example, Table 2 shows the result from one run of our experiments.

Table 2. Missing Packets After Live Migration

	SRVM	No SRVM
Intel IXGBEVF	0	86
DPDK	0	4

When using the Intel IXGBEVF driver, there are 111 received packets in total during the pre-copy phase. There are 86 packets missing after live migration. For DPDK, there are 4 packet missing out of 120 received packets. Note that the application or IXGBEVF driver will not be able to know if there is a missing packet. This essentially causes silent data corruption in applications. With SRVM, there is no data loss. The results effectively show that SRVM can track dirty memory pages and migrate them correctly.

High packet Rate. We use a lockless design for tracking dirty memory pages. If the packet receiving rate is higher

than the dirty memory tracking rate, some data may get lost. We conducted experiments to measure the dirty memory tracking rate. Specifically, we measure the time spent scanning an entire ring buffer (with 512 entries) and to mark those pages. We can scan and mark one packet in 28ns on average, including the time to inspect the ring memory and marking dirty pages. Comparatively, the packet processing rate is much slower than this. When using DPDK with SR-IOV passthrough mode, even if applications can run near line rate at 14.4 million packets (64KB) per second (mpps), and this translates to 69ns for processing each packet. Therefore, our module can track the packet much faster than a 10Gb NIC card can process packets. Also, note that usually near line rate speeds can only be achieved when the application is doing L2 forwarding, which basically does minimal packet processing. If the application does extra processing, the packet rate will drop. Moreover, there are more optimization that can be done to further improve the dirty memory tracking rate. For example, we can easily use multiple threads (CPU cores) for the dirty memory tracking module to improve the speed, and we can also separate inspection and marking tasks to further reduce the time spent on inspection. Therefore, we consider lockless inspection to be feasible.

5.3 Downtime

Downtime is an important metric for live migration. During the switch-over phase of live migration, there is a brief downtime for the migrated VM and its applications. This time should be as short as possible. We consider two types of downtime: VM downtime and network downtime. VM downtime is the period when the VM is paused. Network downtime is the period when the network is not available. SRVM needs extra time to set up the VF in the destination host, so the network will not be immediately ready after the VM is running, which means network downtime will be longer than VM downtime. We conduct experiments to examine both downtimes caused by SRVM.

For the Intel IXGBEVF driver, we use *ping* to measure network device downtime. We use another test VM to ping the live migration VM at short interval (100ms) through the VFs. Downtime is calculated as the total number of dropped ping packets multiplied by the ping interval. For the DPDK driver, *ping* won't work because the device is controlled by the DPDK application. Instead, we use the sending and receiving applications in Section 5.2 to measure network downtime. The test VM sends packets with sequence numbers at 100ms intervals. In the migrated VM, the receiving DPDK application examines the sequence number of the received packet to find dropped packets.

VM downtime is calculated by multiplying the number of missing packets by the interval, and is reported by examining the live migration logs in the source and destination hosts. We synchronize the time between two hosts and examine the time when the VM is paused in the source host and when it is resumed in the destination host.

Inside the migration VM, we run the memory dirtying application as the workload. As a comparison, we also measure the downtime of migrating the VM without a passthrough VF (with the VMXNET3 virtual device). We repeat experiments for three times and report the average downtime. Figure 9 shows the results of downtime for various configurations. The network downtime shown in the figure already includes the VM downtime. Note that the migrated VM is running a memory intensive application, so the measured results in this experiments may be longer than common cases.

Figure 9: Comparison of VM and Network Downtime. For each configuration, VM downtime and network (Net) downtime are measured. For SR-IOV IXGBEVF, the downtime after optimization is shown as Net(OPT)

From the results, we can see our technique does not increase VM downtime. For example, migrating the VM with SR-IOV VF and DPDK driver results in 0.69s VM downtime. Comparatively, migrating the VM with VMXNET3 and DPDK driver results in 0.71s VM downtime. The VM downtimes are comparable. This is because SRVM may only affect VM downtime when it saves and restores VF configuration and we only save and restore minimum data.

But our technique did increase the network downtime. For the SR-IOV DPDK case, the downtime increases from 1.13s (VMXNET3) to 2.01s (SR-IOV). This is because it takes time for the driver to detect the device reset status and to reset the device (we set the interval of detecting device state to 0.1s). The extra downtime time (0.9s) involves device resetting and may have some other factors that we would like to investigate in the future. Note that during this extra time, VM is up and running.

For the SR-IOV IXGBEVF case, the downtime increases from 0.80s to 3.97s. We further examined the IXGBEVF driver and found that the extra downtime is due to the IXGBEVF watchdog timer that checks the device state with a 2s interval. Therefore, after VM is resumed the driver may not detect the reset event until the timer is fired. We change the timer interval from 2s to 0.01s, which allowed us to basically

remove this extra time and to understand the real downtime caused by resetting the device. After this simple optimization, the downtime was reduced to 2.3s. The network downtime overhead is about 1.5s. We plan to further investigate other sources of downtime and optimize it for future work. Note that our experiments show SRVM does not drop TCP connection. We believe TCP is resilient to transient changes to the underlying network like link state.

Previous approaches are evaluated in different hypervisors. Therefore, we cannot directly compare our downtime results with theirs. Also, the major difference of our approach is that it is completely designed and implemented in the hypervisor requiring no modifications to the guest OS and drivers. This is a very important design consideration for cloud service providers or infrastructure software providers, which greatly increases the portability of the approach.

5.4 Overhead

To understand the runtime performance overhead of SRVM, we conduct experiments to measure the network throughput and latency when network applications are running inside the migration VM. We also discuss the extra resource usage by our modules.

Network Throughput. To measure network throughput, we run DPDK in the VM. The application simply receives the packets that are sent from the testing VM through DPDK. Since SRVM only adds runtime overhead during the pre-copy phase, we measure the network throughput only in this phase. We compare the throughput in three different configurations.

- **DPDK SRIOV with SRVM:** DPDK uses the VF in passthrough mode with SRVM. This is the configuration we would like to test.
- **DPDK SR-IOV without SRVM:** DPDK uses the VF in passthrough mode without SRVM. Note that without SRVM, there might be missing packets and the VF won't be available after live migration. We just use this configuration to compare the throughput during the pre-copy phase.
- **DPDK VMXNET3:** DPDK uses the VMXNET3 driver that is backed by the PF in full function mode. Live migration is already supported in this configuration. The results are shown in Figure 10.

Comparing the results of DPDK SRIOV with or without SRVM, we see SRVM does not impact network throughput. This result is expected because SRVM runs on a separate CPU core without acquiring any lock from the VM.

Using SR-IOV VF passthrough mode allows high performance network, especially in a small packet size. VMXNET3 can only achieve about 1.1 to 1.2 mpps at various packet sizes. In contrast, using SR-IOV VF passthrough mode allows high throughput, e.g., with a 64-byte packet size the VF passthrough mode increases throughput by 9.6 times to

Figure 10: Network Performance During The Pre-Copy Phase

11.5 mpps. Note that this experiment is done during the pre-copy phase, so the throughput of VMXNET3 is affected by the pre-copy phase. The throughput of VMXNET3 during normal execution is higher than these numbers.

Network Latency. Another important network performance metric is latency. To measure latency, we run *ping* in the test VM with 1ms intervals. We compare round-trip time (RTT) in three configurations.

- **IXGBEVF with SRVM:** we use the Intel IXGBEVF driver in the migration VM with SRVM enabled. This is the configuration under test.
- **IXGBEVF without SRVM:** we use the Intel IXGBEVF driver without SRVM as comparison. Note that without SRVM, there might be packet loss and after live migration the VF device will not be available.
- **VMXNET3:** we use the VMXNET3 driver, which is backed by a PF in full-function mode.

The results are shown in Figure 10 (b). We find that the latencies when SRVM is enabled are between 0.09s and 0.11s, which is comparable to the latencies when SRVM is

disabled. The results show that SRVM does not affect VM network performance.

Compared with VMXNET3, using VF passthrough mode greatly reduces network latency (by 98% with a 64-byte packet size). This once again shows the advantage of enabling an SR-IOV VF for live migration.

Resource overhead. The memory tracking module is enabled only in the source host during live migration. Therefore, there is no runtime performance overhead for VMs before the live migration is started. However, during live migration the memory tracking module will use a separate CPU core to process and mark dirty pages. Therefore, CPU overhead is one core per live migration task. If multiple threads are used, the number of cores increases accordingly.

The memory tracking module does not copy any memory data from the ring buffer. Instead, it directly inspects the ring buffer. Therefore, it only requires very small memory space for its local memory, which can fit into a single page. We also need some space to save PCI configuration for live migration and this can also fit into one page. Note that SRVM only inspects a small piece of guest memory (e.g., 2 pages for a 512-entry ring) and so the memory access overhead such as address translation and cache misses is also small.

6. Conclusion

In this paper, we presented the design and implementation of SRVM that enables live migration with passthrough SR-IOV VFs. Our experiment results show that SRVM can effectively migrate all received packets to ensure application data integrity. Also, SRVM does not increase VM downtime and does not affect the network performance. By leveraging hypervisor support provided by SRVM, the virtualization system users can take advantage of both high performance networking (SR-IOV VF passthrough mode) and advanced virtualization features (live migration) with their legacy VM configurations without having to modify their guest OS or drivers. For our future work, we plan to develop automation support in cloud management software (e.g. VMware vCenter) to facilitate SRVM.

Acknowledgment

We would like to sincerely thank all those engineers who provided great technical suggestions and comments for our project: Guolin Yang, Radu Rugina, Gabriel Tarasuk-Levin, Sreekanth Setty, Wei Xu, Michael Ho, Doug Covelli, Venkata Subhash Reddy Peddamallu, Xunjia Lu, Haoqiang Zheng, Josh Simons and Chris Rossbach. We would also like to thank all reviewers for their insightful comments and suggestions.

References

- [1] "InfiniBand Trade Association," <http://www.infinibandta.org>.
- [2] "RDMA Consortium," <http://www.rdmaconsortium.org>.

- [3] “HIGH PERFORMANCE, OPEN STANDARD VIRTUALIZATION WITH NFV AND SDN,” http://www.windriver.com/whitepapers/ovp/ovp_whitepaper.pdf, [Online; accessed 12-Oct-2015].
- [4] “WIND RIVER TITANIUM SERVER,” http://www.windriver.com/without-compromise/resources/pdf/CGCS_Product_Overview.pdf, [Online; accessed 12-Oct-2015].
- [5] A. Kadav and M. M. Swift, “Live migration of direct-access devices,” *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 95–104, Jul. 2009.
- [6] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang, “Compsc: Live migration with pass-through devices,” in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE ’12. New York, NY, USA: ACM, 2012, pp. 109–120.
- [7] E. Zhai, G. D. Cummings, and Y. Dong, “Live migration with pass-through device for linux vm,” in *OLS08: The 2008 Ottawa Linux Symposium*, 2008, pp. 261–268.
- [8] “Microsoft Hyper-V,” <http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx>, [Online; accessed 20-Aug-2015].
- [9] “Linux ethernet bonding driver howto,” <https://www.kernel.org/doc/Documentation/networking/bonding.txt>, 2011, [Online; accessed 20-Aug-2015].
- [10] Intel, “Intel Virtualization Technology for Direct I/O,” [Oct-2014].
- [11] 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 conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.
- [12] VMware, “VMware vSphere vMotion Architecture, Performance and Best Practices in VMware vSphere 5,” [Technical White Paper].
- [13] Intel, “Intel 64 and IA-32 Architectures Software Developers Manual,” [May-2011].
- [14] —, “Intel 82599 10 GbE Controller Datasheet,” [Feb-2015 Revision 3.1 331520-002].
- [15] —, “Intel 82599 SR-IOV Driver Companion Guide,” [May-2010 Revision 1.00].
- [16] “Data Plane Development Kit,” <http://dpdk.org>, [Accessed on Nov 17 2015].
- [17] 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 *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03. New York, NY, USA: ACM, 2003.
- [18] “Kernel-based Virtual Machine,” <http://www.linux-kvm.org>, [Online; accessed 15-Oct-2015].
- [19] “VMware,” <http://www.vmware.com>, [Online; accessed 15-Oct-2015].
- [20] H. Z. Weidong Han, “Live Migration with SR-IOV Pass-through,” [Presentation on KVM Forum 2015, Aug 2015].
- [21] PCI-SIG, “Address Translation Services,” [Jan-26-2009 Revision 1.1].
- [22] —, “PCI Express Base Specification Revision 3.0,” [Nov-10-2010 Revision 3.0].
- [23] “Pktgen-dpdk: Traffic generator powered by DPDK,” <http://dpdk.org/browse/apps/pktgen-dpdk/refs/>, [Accessed on Nov 17 2015].