CompSC: Live Migration with Pass-through Devices

Zhenhao Pan
Tsinghua University
Beijing, China
frankpzh@gmail.com

Yaozu Dong
Intel Asia-Pacific Research and Development Ltd
No. 880 Zi Xing Rd, Shanghai, China
eddie.dong@intel.com

Yu Chen Lei Zhang Zhijiao Zhang
Tsinghua University
Beijing, China
{chyyuu,sosilent.lzh,acer.zhang}@gmail.com

Abstract

Live migration is one of the most important features of virtualization technology. With regard to recent virtualization techniques, performance of network I/O is critical. Current network I/O virtualization (e.g. Para-virtualized I/O, VMDq) has a significant performance gap with native network I/O. Pass-through network devices have near native performance, however, they have thus far prevented live migration. No existing methods solve the problem of live migration with pass-through devices perfectly.

In this paper, we propose CompSC: a solution of hardware state migration that will enable the live migration support of pass-through devices. We go on to apply CompSC to SR-IOV network interface controllers. We discuss the attributes of different hardware states in pass-through devices and migrate them with corresponding techniques. Our experiments show that CompSC enables live migration on an Intel 82599 VF with a throughput 282.66% higher than para-virtualized devices. In addition, service downtime during live migration is 42.9% less than para-virtualized devices.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Communications Management—Network communication; D.4.5 [*Operating Systems*]: Reliability—Backup procedures

General Terms Design, Performance

Keywords Virtualization, Live migration, Pass-through device, SR-IOV

1. Introduction

Recently, virtualized systems have been experiencing dramatic growth. In data centers and cloud computing environments, for example, virtualization technology largely reduces hardware and resource costs [9, 18]. The virtual machine live migration technique [14] is considered as one of the most important features of virtualization [13]. It not only significantly increases the manageability of virtualized systems, but also enables several important virtualization usage models like fault tolerance [15, 28].

In such an environment, not only is the performance of CPU and memory virtualization a crucial component, but the performance of network I/O virtualization is also of upmost importance. In recent works, CPU and memory virtualization have been discussed in depth [8], and the performance with recent techniques is near native one [11, 31]. However, I/O device virtualization remains a great challenge, especially on network devices. Emulated network I/O [30] provides a complete emulation of the existing network interface controllers (NIC), and connects physical NICs together via a virtual bridge. Para-virtualized (PV) network I/O [3], which employs an optimized interface, has increased performance compared to emulated network I/O. Although recent efforts [23, 26, 29] largely improve the performance of PV network I/O, there still exists a significant performance gap with native network I/O [25, 26, 29] in addition to the burden of higher CPU overheads. Using pass-through I/O [7, 24], A.K.A direct I/O, a virtualized system can directly access physical devices without interceptions from the hypervisor, and thus is capable of providing near native performance. Single Root I/O Virtualization (SR-IOV) [16] is a modern specification proposed by PCI-SIG. This specification shows how PCIe devices can share a single root I/O device with several virtual machines. With the help of SR-IOV, one hardware device is able to provide a number of PCIe virtual functions to the hypervisor. By assigning these functions to virtual machines as pass-through devices, the performance and scalability of I/O virtualization can be considerably increased.

Nevertheless, pass-through I/O has limitations on virtual machine live migration. In pass-through I/O, the physical device is totally controlled by the virtual machine, and its internal states are not accessible by the hypervisor. If the internal states of the physical device are not migrated during a live migration, the device will stop working and the driver will likely crash. Several efforts have been made on enabling pass-through network I/O migration, such as the bonding driver solution [32] and Network Plug-in Architecture (NPA/NPIA) [2], but these methods all bypass the problem of hardware state migration by switching to either a PV device or an emulated device during the migration, and have similar short comings.

In this paper, we propose CompSC to address the problem of how to efficiently migrate the state of a hardware device and enable virtual machine live migration with pass-through NICs. We directly face and solve the challenge of hardware state migration. With an analysis on different kinds of hardware states, CompSC applies methods of state replay and self-emulation on pass-through devices. With few code changes on the device driver, CompSC achieves our objective of enabling live migration support for pass-through devices with minimal impact on run time performance as well as a minute time cost for hardware state migration.

We implemented CompSC on the Xen [11] platform, with the Intel 82576 SR-IOV NIC [19] and the Intel 82599 SR-IOV NIC [20]. In the evaluation section, we measured the performance impact of our method using micro benchmarks, scp, Netperf and SPECweb 2009 [4]. Our results show that CompSC enabled live

VEE'12, March 3–4, 2012, London, England, UK.

migration support on an Intel 82599 VF with a throughput 282.66% higher than PV network devices and 89.57% higher than VMDq. We also measured the service downtime during live migration; our solution had a 42.9% shorter downtime when compared to the PV network device.

Implementing and deploying CompSC is easy. A line count on code changes required by our implementation shows that CompSC needs little coding effort. After we implemented CompSC support on an Intel 82576 NIC, adding CompSC support on an Intel 82599 NIC only required two days time. Thus, we are convinced that CompSC can easily be deployed on a wide range of different NICs.

The remainder of this paper is structured as follows: Section 2 introduces the related work. Section 3 discusses the methods of migrating hardware states with different attributes. Section 4 describes the design of CompSC and the structure of the system. Section 5 describes how we implement CompSC on the Xen platform with an Intel 82576 NIC. Section 6 shows experimental results. Section 7 discusses the usage model of CompSC and we conclude in Section 8.

2. Related work

2.1 Migration

In the early years of distributed system research, process migration [27] was a hot topic. Process migration has many advantages, including processing power management, resource locality and fault resilience. However, process migration has suffered disadvantages with regards to implementation complexity and inflexibility. With the development of virtualization, the possibility of the live migration [14] of the whole operating system (OS) is now realizable and has in fact become a typical solution. In the study of Chen et al. [13], migration is considered as one of the most important features of virtualization. There are also studies that show further benefits of the live migration of virtual machines, including fault tolerance [15, 28] and trusted computing [12].

Live migration of virtual machines takes the advantage of the narrow and identical interface provided by the hypervisor. In the study by Christopher et al. [14], the process of live migration is divided into six stages:

1. Pre-Migration stage
2. Reservation stage
3. Iterative Pre-copy stage
4. Stop-and-copy stage
5. Commitment stage
6. Activation stage

Before the stop-and-copy stage, the virtual machine is running on the source host in the usual way. After the activation stage, the virtual machine runs on the destination host. The downtime (i.e. the time when the virtual machine is out of service) of the process consists of stop-and-copy and commitment stage. Downtime is one of the most important measurements of live migration.

There have been efforts on migration of the whole OS without virtualization as well. In a study of Michael et al. [22], issues and the solutions thereof for migration by OSs are discussed. Since OSs can be treated as a drivers of the whole machine, some of the issues raised by [22] are similar to ours.

2.2 SR-IOV

SR-IOV [16] is a new specification defined by PCI-SIG. The purpose of SR-IOV is to provide multiple PCI interfaces of one device in order to fit the usage model of directly-assigned/pass-through devices and provide increased performance. An SR-IOV device consists of one PF (physical function) and several VFs (virtual functions). The typical usage of an SR-IOV NIC on a virtual machine consists of using VFs as pass-through devices of virtual machines; the PF is used as a device of device domain or privileged domain, not only for networking, but also for VF management. On a PCI bus, a VF looks identical to an independent PCI device. Also, in virtual machines, pass-through VFs are equivalent to typical PCI NICs.

In today's cloud computing solutions, SR-IOV has been used in several NICs. In this paper, we use Intel 82576 and Intel 82599 NICs in our experiments, each of which support SR-IOV.

2.3 Similar works and technologies

There are several efforts on the topic of live migration with pass-through devices. In a study by Edwin et al. [32], the Linux Ethernet Bonding Driver [1] is used. In [32], a PV network device is used as the backup device of a pass-through device. Before the start of a live migration, the pass-through device is hot unplugged using an ACPI event. In this way, there is no need to worry about migrating the pass-through device. This method does not require any code changes on the virtual machine guest kernel, but has several disadvantages:

1. It only works with Linux guests.
2. It requires an additional PV network device. The physical device must be connected to the same Ethernet switch with the pass-through device. This may lead to additional hardware cost and resources costs.
3. The hot unplug event introduces another service downtime in our test. (Section 6.5)
4. After live migration, the driver clears every statistic register in the pass-through device, rendering the statistic function inaccurate or disabled.

In a similar work by Asim and Michael [21], a shadow driver is implemented to redirect network requests to a backup device during live migrations. Besides the flaws mentioned above, the method in [21] requires as many as 11K LOC (lines of code) changes on both the hypervisor and the guest kernel.

VMDq (Virtual Machine Device Queues) [5] is a technique proposed by Intel. The idea of VMDq is similar to SR-IOV, as both methods assign hardware resources to the virtual machine. In contrast to SR-IOV, however, VMDq also benefits from the PV network device. Unlike SR-IOV, which exposes a complete device interface to the virtual machine guest, VMDq only provides network queues to the virtual machine guest. With PV techniques like shared pages, VMDq avoids packet copying between the virtualized network queue and the physical network queue. VMDq provides faster performance than PV network devices and is still able to support live migration in a similar way. We elaborate the comparison of performance and downtime between VMDq and our solution in Section 6.5.

Network Plug-In Architecture (NPIA/NPA) [2] is an architecture raised by VMware and Intel that tries to solve the issues of pass-through device management and live migration. Instead of supporting all pass-through NICs, NPIA only focuses on SR-IOV [16] NICs. NPIA designs a shell/plug-in pair inside the kernel of the virtual machine. The shell provides a layer similar to a hardware abstraction layer, while the plug-in implements hardware communication under the shell. The plug-in can be plugged or unplugged during run time. To reduce the downtime during plug-in switches, an emulated network interface is used as a backup. By unplugging the plug-in, NPA can easily support live migration. Just like bonding driver solution, NPIA uses a software interface as backup device. Compared to the bonding driver solution, NPIA may need less

time switching the pass-through device to the backup. One major drawback is that NPIA also needs to completely rewrite the network drivers, which might prevent NPA from being widely employed.

From the perspective of ReNIC [17], hardware extensions are proposed to solve the issues of SR-IOV network device live migration. ReNIC meets similar difficulties with us, and solves them using hardware way.

3. Approaches of hardware state migration

The core problem with live migration support of pass-through devices is the migration of hardware states. The whole of the pass-through devices are assigned to virtual machines, rendering them inaccessible to the hypervisor. In this section, we propose methods of solving this problem.

3.1 I/O registers migration

I/O registers are the main interface between hardware and software. Almost every visible state of a hardware device is exposed by various kinds of I/O registers. In modern PCI architectures, two kinds of I/O registers are used: Programmed I/O (PIO) and Memory-mapped I/O (MMIO). Reading/writing operations of PIO and MMIO are atomic, and the virtual machine will not be suspended during an I/O reading or I/O writing.

I/O registers are classified into different kinds according to the method of access. One of the most common kinds is read-write registers. If access to a read-write register does not lead to side effects, then the register can be simply migrated by the hypervisor. Other kinds of registers, such as read-only and read-clear registers, cannot be simply migrated by the hypervisor, however.

The access of certain registers may result in side effects. For example, modifying a NIC's TDT (Transmit descriptor tail) register may trigger packet transmission. Without the full knowledge of these registers, access of them by the hypervisor may cause unexpected behavior or device failure.

3.2 State replay

Hardware specifications describe every detail about the interface between the device and driver, and hardware behavior. Given knowledge of the past communications on the interface, the current state of the hardware can easily be deduced. It is assumed that the driver knows the past communications on the hardware-software interface as well as the hardware specification. In most cases, the driver is able to drive the destination hardware from an uninitialized state into some specified state by replaying a given set of past communications.

The idea of state replay consists of two stages: a recording stage, where driver must record every operation of the hardware on the source machine; and a replaying stage, where the driver reads past operations from a list, and commits them to the destination machine one by one.

In regards to state replay, driver complexity may be a problem. Because recording every past communication requires so much effort, driving the destination device may also need a significant number of code changes. Fortunately, with the knowledge of devices, many communications can be optimized. For example, the device driver may write a register many times. If the writing operation of the register brings no side effects, one does not need to record each operation. Instead, one can record only the last one, because it is only the last one that is valid in the hardware.

Another efficient optimization technique is to define operation sets(opset). Some drivers' implementations may consist of several device operations. Instead of recording every step of the drivers' work, the devices' operations are packed into operation sets. Figure 1 illustrates this optimization. In the figure, four operations `op1`, `op2`, `op3` and `op4` are packed into one opset `opset1`. With the assumption that a live migration will not happen inside operation sets, three states are safely omitted: `A`, `B` and `C`.

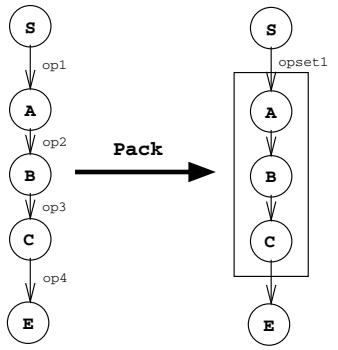

Figure 1. Packing device operations into an operation set

The opset optimization works especially well on NICs. With well-designed operation sets, the number of internal states of NICs can be largely reduced. In the case of VFs on the Intel 82576 NIC, i.e. the one used in our evaluation, all initializing and sending/receiving operations are packed into operation sets. The remaining states include only {uninitialized, up, and down} together with a collection of setting registers. In this kind of set up, only the latest operations on each setting register and whether or not the interface is up need to be tracked. In addition, the code for driving the destination hardware into the state of source hardware is significantly simplified by invoking existing initializing codes. In Section 6.6, we list the size of hardware states and past operations to be migrated for the Intel 82576 and Intel 82599 NICs.

Avoiding live migration inside an operation set needs a synchronizing method between the device driver and the hypervisor. A common question is whether or not this affects performance. The answer depends on the granularity of operation sets. If the driver makes an operation set that lasts for an extended period of time (e.g. several seconds), one can imagine that live migration may take a long time. Also problematic would be if the driver makes an operation set that can be invoked millions of times per second. With a set of well-defined operation sets, negative impacts on performance can be minimized. In Section 6.4, we prove that the performance deterioration in our implementation is negligible.

3.3 Self-emulation

Statistic registers of type read-only and read-clear commonly cannot be migrated through the software/hardware interface. The register that counts dropped packets in the NIC is an example. The only way to alter the register is to try to drop a packet. This is difficult, because to drop a packet would need cooperation with the external network. All existing solutions [2, 21, 32] do not cover this register. Instead, they perform device initialization after live migration, reset all statistic registers, and cause the statistic functions to become inaccurate or disabled.

Statistic registers often have mathematical attributes, e.g. monotonicity. After a live migration, one statistic register may have an incorrect value; the difference between its value and the correct value should be a constant. For example, let the count of dropped packets be 5 before live migration. After live migration, the same register on the destination hardware will be initialized to 0. After that, the value of register will always be smaller than the correct value by 5. If the value on the destination hardware is 2, the correct value will be 7. In the case of a read-clear register, the relationship is similar, with one notable difference: only the first access to a read-clear register will get an incorrect value after live migration.

With such a clear logic, the classic trap-and-emulation is chosen. In self-emulation, every access to a read-only or read-clear statistic register is intercepted by a self-emulation layer. In the layer, the correct value is calculated and returned to the caller. The self-emulation layer can be placed in any component on the access path of the register (e.g. the driver, the hypervisor). Figure 2 shows an example where the self-emulation layer is in the hypervisor.

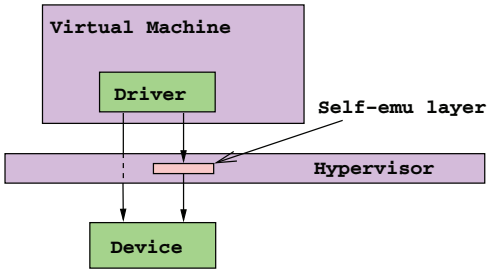

Figure 2. An example structure of self-emulation

3.4 Summary

I/O register migration is easy to perform, but the number of hardware states that support it are quite limited. State replay covers almost every hardware state, but demands extra code efforts in the driver. Statistic registers are hard to migrate, but can be covered by self-emulation. One practical approach for migration is to use the three of them in combination: use state replay for most hardware states, and use I/O register migration and self-emulation when possible.

We classify the states of the Intel 82576 VF as follows: configurations of rings such as RDBA (Receive Descriptor Base Address), TXDCTL (Transmit Descriptor Control) are migrated by I/O register migration; interrupt related registers and settings inside the Advanced Context Descriptor are migrated using state replay; and all statistic registers are covered by self-emulation. Using the prescribed methods in this way, the live migration of network devices in our experiment runs smoothly.

4. Design of CompSC

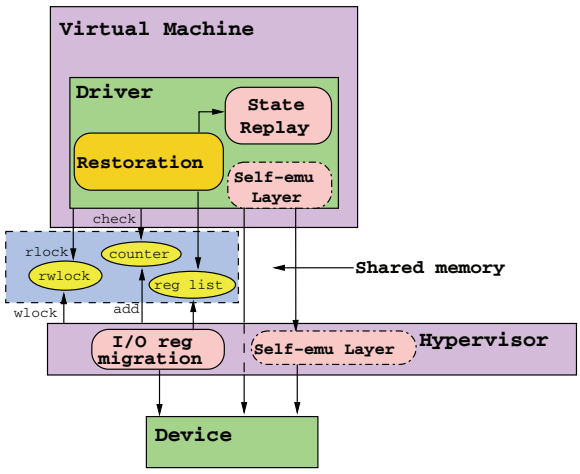

Figure 3. CompSC architecture

The architecture of CompSC is presented in Figure 3. The driver in the virtual machine is responsible for state replay and the hypervisor covers I/O register migration. A piece of shared memory between the hypervisor and the virtual machine is used for synchronization. Two self-emulation layers are provided in the driver and in the hypervisor.

Among the six stages of live migration [14], CompSC works inside the stop-and-copy stage and the activation stage. The usage of CompSC is intelligible: collecting the hardware states of the pass-through device at the stop-and-copy stage, and restoring them to the destination hardware at the activation stage. In addition, while collection is completed by different components (e.g. the hypervisor, the device driver, self-emulation layer), restoration is finished by the device driver only.

4.1 Synchronization

From the perspective of the device driver, live migration happens in a flash. After one context switch, the hardware suddenly turns into an uninitialized state. If there is anything that can indicate a live migration, it must be checked before every hardware access. If we use the state replay method and define several operation sets, the driver will never detect the disturbance of a live migration.

CompSC creates a shared memory area between the hypervisor and the virtual machine. An rwlock and a version counter are preserved in the memory area. The rwlock indicates the status of migration, and the counter records the number of live migrations that have occurred. When the stop-and-copy stage starts, the hypervisor tries to hold the write lock. In the activation stage, the hypervisor increases the version counter and releases the write lock. Conversely, the driver acquires the read lock before every hardware access. Once the lock is held, the driver checks the version counter to figure out whether a live migration has just occurred. If so, the restoration of the device driver will be invoked. In this way, the hardware is never accessed in an uninitialized state.

The logical meaning of the rwlock is as an indicator of who took over the hardware device. The device driver locks the read lock whenever it wants to access the hardware. After accessing is finished and the device state is taken over by the hypervisor for live migration, the driver unlocks the read lock. The hypervisor acquires the write lock before it touches the hardware device, after which the hardware device is taken over by the hypervisor.

We show that the cost of rwlock is relatively low. Intuitively, the lock will not be contended with as all the lock operations in the driver are read lock. The only costs during run time are the memory accesses and a little bit of cache pollution. In Section 6.4, we provide an evaluation on the costs of the rwlock.

4.2 Hardware state migration

CompSC performs the I/O register migration in a straightforward way. The hypervisor scans the list of registers on the network device and saves them into the shared memory area mentioned in Section 4.1. After a live migration, the driver inside the virtual machine is held responsible for restoration. Making as few least code changes as possible is one of CompSC's driving factors. In the design of CompSC, we try to prevent the hypervisor from having any device-specific knowledge. The hypervisor does not know the list of registers; it gets this list from the shared memory area, put there by the driver during the boot process.

State replay is completed by the device driver. The operation sets and hardware operations are protected by rwlock. Every time before the driver releases the read lock, it stores enough information of past operations or operation sets to achieve a successful restoration. In the restoration procedure, the device drives the destination hardware into the same state using the saved information.

The self-emulation layer can be put into the hypervisor or the device driver. A self-emulation layer in the hypervisor will trap all accesses to the emulated registers and return the correct value. A self-emulation layer in the driver will process the fetched value

correct after the access as described in Section 3.3. The former needs only the list of emulated registers and leads to fewer code changes in the driver, but at the expense of degraded performance due to I/O interception. The latter gains less overhead, but produces much more code changes. CompSC provides both methods, and the driver is free to choose either. A detailed discussion of the overhead of I/O interception is described Section 6.2.

4.3 SR-IOV NIC Support

On an SR-IOV NIC, migration becomes slightly different. The PF in an SR-IOV NIC provides management interfaces with the VFs. In our environment (Intel 82576 and Intel 82599), the PF holds a subset of VF states such as MAC addresses. In this paper, we call them VF-in-PF states (the VF part of PF states). Some of VF-in-PF states can be accessed by the VF driver through the PF-VF mailbox [19] and can be migrated using state replay, but the remaining can only be accessed through PF registers by the PF driver. In order to cover all hardware states, CompSC also uses the state replay method on the PF driver. The PF driver records all hardware operations of the specified VF before migration and commits them to the destination machine later.

5. Implementation

We used Xen [11] as the base of our implementation on the 64-bit x86 architecture. For NICs, we used the Intel 82576 (an SR-IOV 1Gbps NIC), and the Intel 82599 (an SR-IOV 10Gbps NIC). The PF drivers and the VF drivers of the Intel 82576 and Intel 82599 were changed in our implementation, detailed in Section 5.1. Section 5.3 presents the self-emulation layer.

Xen provides functions in the hypervisor to access foreign guest domains' memory page, which allow for easy implementation of shared pages between the hypervisor and the device driver. Details are offered in Section 5.2.

5.1 Driver changes

In our experiment, CompSC is executed on Intel 82576 and Intel 82599 NICs, with corresponding VF drivers IGBVF and IXGBEVF, respectively. As mentioned in Section 4.1, the read lock of the rwlock is used to protect the hardware operations and operation sets we defined. As soon as the lock is acquired, the driver checks the migration counter and invokes a restoration procedure if a migration is detected.

Formally, we pack `igbvf_up` and `igbvf_down` in the igbvf driver, and `ixgbe_up` and `ixgbevf_down` in the ixgbevf driver as operation sets. All hardware operations and operation sets are protected by the read lock. Because most device states have a copy in the driver, the state replay needs few code changes. The restoration procedure conducts the following tasks: device initialization, saved register writing, and the restoration of all states using state replay.

5.2 Shared page and synchronization

Shared pages are allocated by the NIC driver. The driver allocates several continuous pages which are structured to contain three pieces of information:

- The rwlock and the version counter;
- The list of registers that should be saved in the migration;
- The list of counter registers that need the help of the self-emulation layer in the hypervisor.

After initialization, the GFN (guest frame number) of the first page is sent to the hypervisor. In our implementation, this number is sent by PF-VF communication. For non-SR-IOV NICs, this number can be sent by a high level communication using the TCP/IP protocol.

When a live migration starts, memory pages are transferred until the stop-and-copy stage [14], until the virtual machine is to be suspended. Right before suspending, the write lock of the rwlock is acquired by the hypervisor. In this way, the hypervisor seizes the control of the device hardware. After the virtual machine is suspended, the hypervisor accesses the shared pages, and saves all registers listed in them. The remaining part of live migration transpires on the backup machine. Before the hypervisor tries to resume the virtual machine, saved values of read-only and read-clear counter registers are sent to the self-emulation layer in the hypervisor.

The first time the driver acquires the read lock, the device restoration procedure is invoked. The driver does necessary initializations on the device and restores the state using information collected by the state replay and I/O register migration. When all of this is accomplished, device migration has successfully been achieved.

5.3 Self-emulation layer

Xen hypervisor provides functions for trapping memory accesses, and the self-emulation layer in the hypervisor is based on them. Every time the self-emulation layer receives a request to commit self-emulation on a list of registers, it places a mark on the page table of the register. All further access to these registers will be trapped and emulated. The emulation does the real MMIO and the layer returns the calculated value to the virtual machine. The granularity of this trapping mechanism in our implementation is one page. On 64-bit x86 architecture, this translates to 4 KB. It should be noted that this may lead to unnecessary trappings and performance impacts; we elaborate on this is in Section 6.4.

5.4 Pages dirtied by DMA

The process of live migration is highly dependent on dirty page tracking. Dirty page tracking is implemented with the help of page tables in the newest version of Xen. However, memory access by DMA cannot be tracked by page tables. Intel VT-d technology [7] provides I/O page tables, but it still cannot be used to track dirty pages.

Hardware cannot automatically mark a page as dirty after DMA memory access, but marking the page manually is effortless. All that is required is a memory write. In a typical NIC, hardware accesses descriptor rings and buffers by invoking DMA. After the hardware writes to anyone of them, an interrupt is sent to the driver in the virtual machine guest kernel. Because the driver knows all changes on the descriptor rings and buffers, it simply performs a series of dummy writes (read a byte and write it back) to mark the pages as dirty.

This method misses a few packets that have already been processed by the hardware but have yet to be processed by the driver. This may lead to packet duplication or missing. Fortunately, the amount of such packets is small enough that connections of reliable protocols like TCP connections will not be affected. Section 6.3 presents the details of these duplicated or missed packets.

5.5 Descriptor ring

During our implementation, we came across an issue with both Intel 82576 VF and Intel 82599 VF. The head registers of descriptor rings (either RX or TX) are read-only. Their values are owned by hardware, and writing any value except for 0 is not allowed (writing 0 is an initialization). Consequently, head registers should be restored using state replay. However, committing state replay on this register is not that easy. The only way of increasing head registers is trying to send/receive a packet. By putting dummy descriptors in the rings, altering head registers does not need cooperation with external network, but it costs thousands of MMIO writings.

One method of solving this is resetting everything in the rings. By freeing buffers in the rings and resetting the rings to be empty, the driver will work well with the device. But this method needs tens or hundreds of memory allocations and freeings. The time cost associated with this method may be a problem, especially when the device has a large ring.

Another idea to handle the head registers is shifting. Instead of restoring the value of head registers, we shifts the ring itself. During the restoration procedure, the driver shifts the RX and TX rings, and makes sure the position of each original head is at index 0. After that, the driver needs only to write a 0 on the head registers to make the rings work. In addition to this, the driver must save the offsets between the original rings and the shifted rings. Every time the head/tail registers or rings are accessed by the driver, the offsets are used to make sure the access is completed correctly.

In CompSC, we use the method of shifting. Shifting introduces additional operations to access to indices/rings, and thus consumes more CPU time in the driver. Section 6.4 measures this performance impact.

5.6 Implementation complexity

The CompSC needs modifications in the network driver. Among the common concerns about the practicality of deployment, the complexity of device code changes is the most critical. In Table 1, we depict the number of line code changes in our implementation on different components. The synchronization mechanism is common to every network driver capable of live migration. The number of common code changes is just 153 lines. In the IGBVF driver, only 344 lines of codes are added or modified, and in the IXGBEVF driver only 303 lines are added or modified. Even the CompSC architecture itself has a small number of code changes. 808 lines of code changes were committed in either the Xen hypervisor or Xen tools.

We claim that one can easily patch an existing device driver into a CompSC supported one. During our implementation, we first completed the CompSC support on Intel 82576 NIC and related experiments. Further efforts to add the CompSC support on Intel 82599 NIC only cost us two days. As a result, CompSC is easy and practical to deploy.

Table 1. Lines of code changes in the implementation

	Line of code
Xen hypervisor	362
Xen tools	446
VF driver(common)	153
IGBVF driver	344
IGB driver	215
IXGBEVF driver	303
IXGBE driver	233

6. Evaluation

In this section, we present the results of our experimental data that compare a system equipped with our implementation of CompSC to the original system (without CompSC); a system with PV network device; a system with the bonding driver solution; and finally a system using the VMDq technique. We first present a micro benchmark to measure the performance impact of the self-emulation layer in the hypervisor. In Section 6.3 we show our measurements of the number of duplicated or missed packet due to the DMA dirty page. With scp, Netperf and SPECweb2009 benchmarks, Section 6.4 presents a comparison of the run time performance between several situations including the original environment and our implementation. Section 6.5 illustrates the migration process using a timeline figure comparing CompSC, a PV network device, the VMDq technique, and the bonding driver solution. Finally, Section 6.6 lists the size of hardware states to migrate.

6.1 Benchmarks and environment

Our target application is virtualized web servers. As a result, in our evaluation, we focus on the throughput and the overall performance as web servers. We use the Netperf benchmark, perform file transferring using scp to measure the throughput of virtual machines, and use SPECweb2009 to evaluate web server performance.

The evaluation uses the following environment: two equivalent servers equipped with Intel Core i5 670 CPU (3.47 GHz, 4 cores), 4 GB memory, 1 TB hard disk, an Intel 82576 and an Intel 82599 NIC; one client machine for the SPECweb2009 client, running an Intel Core i3 540 CPU (3.07 GHz, 4 cores), 4 GB memory, 500 GB hard disk, one Intel 82578DC NIC and one Intel 82598 NIC. These three machines are connected using a 1 Gb Ethernet switch and a 10 Gb Ethernet switch. The virtual machine uses 4 virtual CPUs, 3 GB memory, and one VF of Intel 82576 NIC or Intel 82599 NIC, and is virtualized in HVM (Hardware-assisted Virtual Machine). The virtual machine also uses a PV network device in the tests with PV device.

6.2 Micro benchmark for self-emulation

In Section 3.2 we presented the idea of self-emulation, and discovered that the self-emulation approach has a trade-off between accuracy and performance. In this section, we measured the performance loss due to self-emulation. In our test, we accessed one of the counter registers 10,000 times. Using TSC register, we measured the total cost of CPU cycles and calculate its average. We ran our test in both the direct-access and intercepted scenarios. Table 2 contains the results.

Table 2. Micro benchmark for MMIO cost

MMIO direct	MMIO intercept
3911 cycles	11860 cycles

The results of our self-emulation test show that MMIO with interception needs an additional 7,949 cycles for `VMEnter/VMExit` and context switches. For low access frequencies, this overhead is negligible, but for high access frequencies, the overhead may become problematic. Next, we measure the access frequency of statistic registers on different workloads.

Table 3. Access rate of statistic registers

	Time	RX bytes	TX bytes	MMIO
Netperf	60.02 s	54.60 G	1.19 G	4.50/s
SPECweb	8015 s	8.55 G	294.68 G	4.50/s

Table 3 shows the access frequency of statistic registers. From these results, it can be seen that the frequency of statistic register access was a constant: 4.5 accesses, no matter which task was been executed, and no matter which of either RX or TX was heavier. A subsequent code check on the Linux kernel uncovered this behavior. The IGBVF driver used a watchdog with a 0.5 Hz frequency to observe the statistic registers, and the access frequency is expected to be a constant. At such a low frequency, the overhead of self-emulation is roughly 10.30 μs/s. With the consideration of cache and TLB, the overhead may be slightly more, but it can still be considered negligible.

6.3 Duplicated and missed packet due to unmarked dirty page

In Section 5.4, we presented the idea of marking pages dirtied by DMA, and claimed that the solution may cause packet loss and packet duplication. In this section, we measured the number of duplicated and missed packets under different workloads. A busy CPU leads to longer time in suspension, and a busy NIC increases the number of packets received/transmitted during migration. A straight-forward prediction is that the number of duplicated and missed packets may become larger while both the CPU and NIC are busy. In our measurements, the workload of scp and SPECweb were used, and the scenario when there is no workload is also considered.

Table 4. Duplicated and missed packet counts during live migration, using Intel 82576

	Dup	Miss
No workload	0	0
scp	0	0
SPECweb	0	3

The results in Table 4 show that our method worked perfectly both scenarios when there was no workload and also in scp; no packet loss or duplication occurred in either case. On the SPECweb workload, only 3 packets were lost, however, these abnormal behaviors did not break the TCP connection, and thus the service was kept alive during the migration.

6.4 Performance with workloads

In this section, the run time performance of CompSC is measured and compared to a bare system (without CompSC). CompSC adds a synchronization mechanism between the hypervisor and the driver, the performance impact of which was a vital concern to the design strategy of our solution. Our method of handling descriptor rings as described in Section 5.5 also has performance impact at run time. The self-emulation layer in the hypervisor also has performance overhead. Although in the test outlined in Section 6.2 concluded the measurable overhead is small, we still consider this factor in more detail this section. In Section 5.3 we described how the self-emulation layer in the hypervisor may perform unnecessary interceptions; because the layer is optional and only enabled after migration, we measured both cases with and without the layer enabled.

The first test ran a benchmark of Netperf and an scp workload with a CD image file `specweb2009.iso` of size 491.72 MB. In this test we measure the throughputs of the workload in three situations: original IGBVF driver (VF orig), IGBVF driver with CompSC (VF+comp), and IGBVF driver with CompSC and with the self-emulation layer enabled (VF+comp+int). Figure 4 illustrates the results. In the figure, we see that the throughput of three test cases were almost the same in the two different workloads. The CPU utilization in the figure shows that the VF+comp and VF+comp+int scenarios consume almost the same amount of CPU resources as the VF orig case. The only thing notable in the figure is that the throughput of scp on VF+comp+int was slightly less than that on VF orig and VF+comp. On the Netperf benchmark, the network was the bottleneck of the whole system while on the scp workload, it is the CPU that was the bottleneck. A CPU utilization near 100 percent shows a CPU bottleneck of a single-threaded workload. When the self-emulation layer in the hypervisor was enabled, more CPU resources get consumed and thus this scenario had a slightly lower performance compared to others.

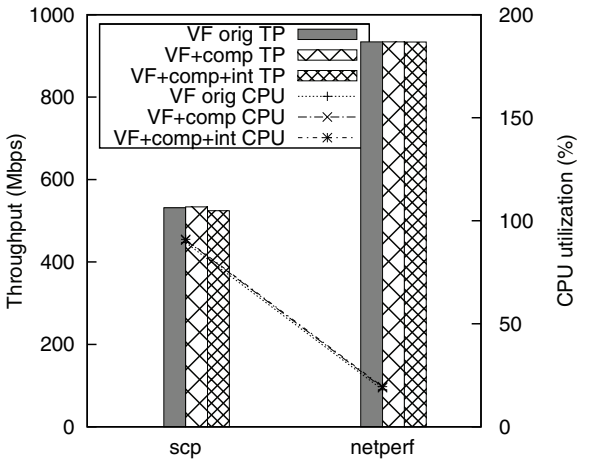

Figure 4. Throughput and CPU utilization by scp and Netperf on Intel 82576

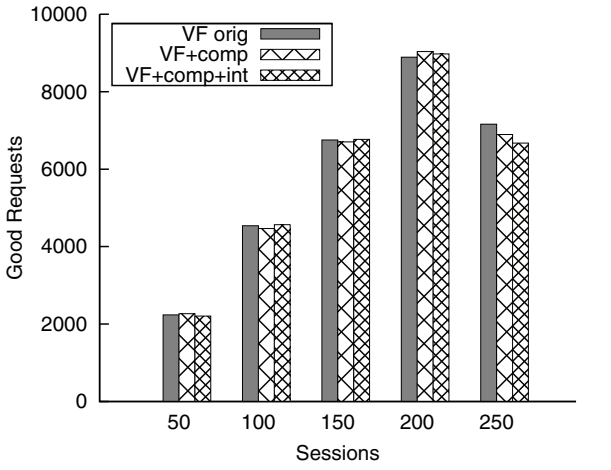

Figure 5. Good requests by SPECweb 2009 on Intel 82576

SPECweb 2009 is our real-world benchmark. In our evaluation, we configured and ran SPECweb 2009 with different pressures on the server in the virtual machine. We invoked the test with five different configurations: with 50, 100, 150, 200, and 250 concurrent sessions respectively. Tests with these configurations were run under three cases: using the original IGBVF driver (VF orig), the IGBVF driver with CompSC (VF+comp), and the IGBVF driver with CompSC and with the self-emulation layer enabled (VF+comp+int).

SPECweb 2009 classifies the requests based on response time into three types: good ones, tolerable ones, and failed ones. The good ones are requests which have a quick response, while the tolerable ones have a long but tolerable response. Failed ones have an intolerable response time or no response at all. In our test, we collected the number of good requests and present them in Figure 5.

The number of good requests increased in a linear fashion with the number of sessions, until we met a bottleneck at 250 sessions. To understand this bottleneck clearly, we also represent the average response time of requests in Figure 6. The average response times were comparable when the number of sessions was less than 250. On the test with 250 sessions, the response time grew by almost

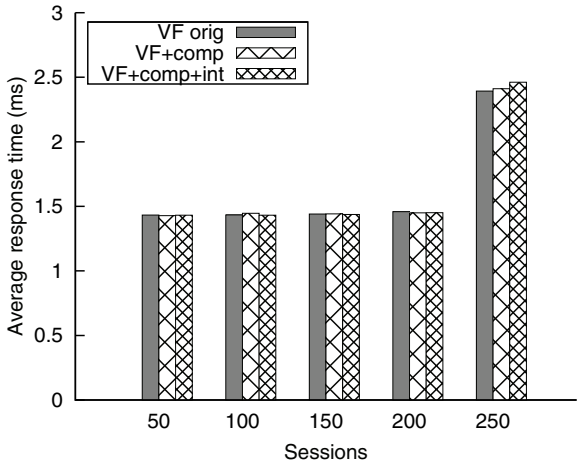

Figure 6. Average response time by SPECweb 2009 on Intel 82576

2/3 compared to previous sessions, which clearly indicates that the server was in a heavy workload.

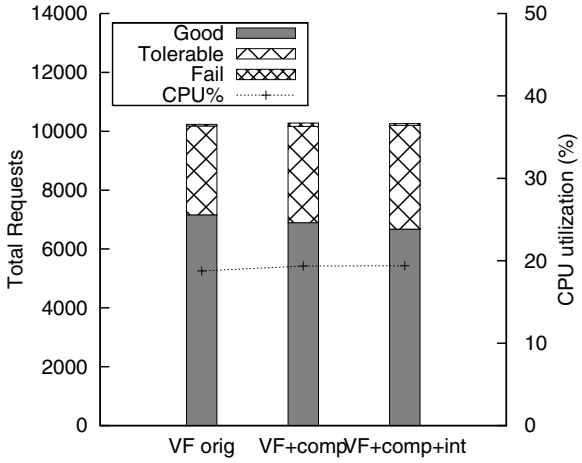

Figure 7. Performance and CPU utilization by SPECweb 2009 with 250 sessions on Intel 82576

Before reaching the bottleneck, no obvious differences present themselves in the three scenarios depicted in Figure 5 and Figure 6. This convinces us that the performance impact of our method under light workloads can be ignored. When the test approaches 250 sessions, VF+comp generated 3.74% fewer good requests than VF orig, and VF+comp+int generated 6.80% fewer good requests (in Figure 5). In regards to the measurement of average response time, VF+comp had a 0.75% higher response time and VF+comp+int had 2.88% higher response time when compared to VF orig (in Figure 6). To figure out why this is the case, we collected detailed performance data and CPU utilization results with the 250 sessions case in Figure 7.

The total requests handled by the server in the three scenarios were on the same horizontal line in Figure 7. The reason why VF+comp and VF+comp+int have fewer good requests is due to longer response time, in which case some of the requests were classified into tolerable requests. In other words, both the VF+comp and VF+comp+int cases had the same service capability, but had slightly longer response times. In the meantime, VF+comp and VF+comp+int consumed 0.59% and 0.64% more CPU than VF orig, respectively; this impact can also be considered as very small.

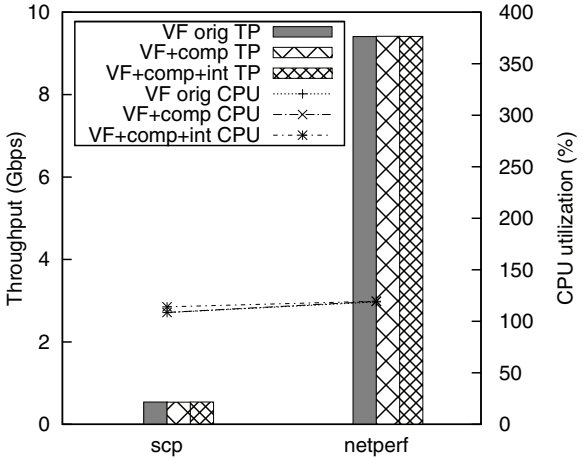

Figure 8. Throughput and CPU utilization by scp and Netperf on Intel 82599

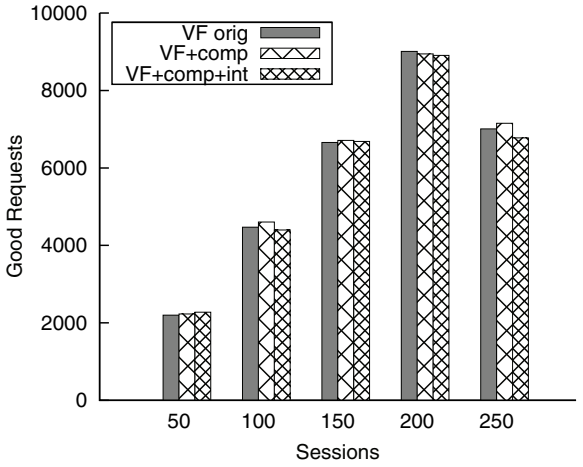

Figure 9. Good requests by SPECweb 2009 on Intel 82599

In order to evaluate the performance impact of CompSC on more challenging scenarios, we ran the Netperf tests, scp tests and SPECweb 2009 tests on an Intel 82599 VF with CompSC. As Figure 8 shows, the Intel 82599 VF was capable of more than 9.4 Gbps throughput on the Netperf tests, and CompSC had no detectable impact on throughput. In the scp tests, the Intel 82599 VF produced almost the same throughput as the Intel 82576 VF because the CPU was the bottleneck in scp tests. In the SPECweb2009 tests, the performance of the Intel 82599 VF was also comparable to the performance of the Intel 82576 VF. We can clearly see the bottleneck was reached at 250 sessions, and CompSC slightly degraded response time in a similar fashion to the Intel 82576 VF.

6.5 Service down time

In this section, we graphically illustrate the whole process of live migration. We treated the server as live if it had a positive throughput. To fulfill the throughput, we ran the Netperf benchmark during our test. The throughput on the Netperf client machine was recorded as data. In order to shorten migration time (mostly decided by the amount of memory), we modified the configuration of

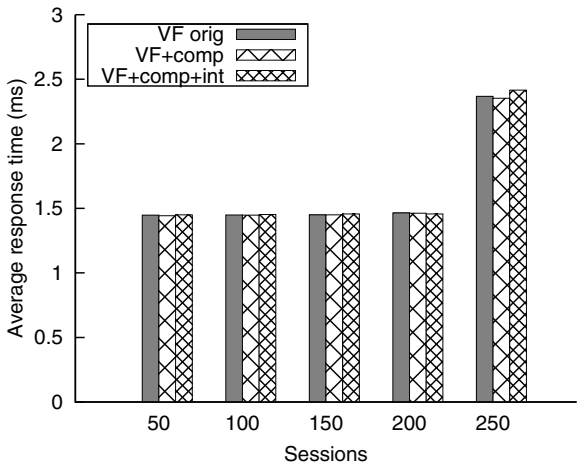

Figure 10. Average response time by SPECweb 2009 on Intel 82599

the virtual machine. In this test, the virtual machine was equipped with 1 GB of memory.

Figure 11 presents the throughput and CPU utilization during a live migration when using CompSC on an Intel 82576 VF, and Figure 12 presents the results with the PV device using an Intel 82576 PF as physical device. In the two figures, we first notice that the service downtime of CompSC was about 0.9s while the downtime of the PV device was about 1.4s. CompSC had a 35.7% shorter and better service downtime. It can also be seen that in PV device test, service was down shortly before the 1.4s downtime (On about 20.6s) and CPU utilization reached as high as 327%. The reason for this behavior is the suspension process of PV-on-HVM (Para-virtualization on Hardware-assisted Virtual Machine). The suspension on PV-on-HVM demanded the cooperation of drivers in the virtual machine. This cooperation consumed many CPU resources and caused a small period of service down. If we focus on CPU utilization, we notice that the CPU% lines on both figures have the same shape, and the line on Figure 12 is higher than the line on Figure 11. This fits our expectation. The pass-through device consumed less CPU resources than the PV device, which is the precise usage of pass-through devices.

We also have a test with regards to the bonding driver solution. Due to limitations of current Xen implementations, we only have a test for the bonding driver on a VF from the Intel 82576 and an emulated E1000 device as backup. Figure 13 shows the results of this test. The bonding driver solution had an extra service down at about 3s. Because the switching of the bonding driver took several milliseconds and caused packets to be lost. The shape of CPU utilization line is similar to that of the CompSC and PV device, but the throughput was much less. The performance of the emulated device was not as good as either the PV device or the pass-through device. In the figure, it can also be seen that the service downtime of bonding driver solution was about 1.2s.

In order to assess the performance benefit of SR-IOV, we evaluated the migration process of an Intel 82599 VF. Figure 14 depicts the results of our test on an Intel 82599 VF with CompSC solution. The shape of the CPU line and throughput line are almost the same as in Figure 11. Sometimes the throughput collapsed for a little while (less than 0.2s), because Dom0 and the guest were sharing the physical CPU, and a throughput of 10 Gbps was very challenging for our environment. The test results of PV device are shown in Figure 15. We used the PF Intel 82599 PF as the physical device of the PV device, however, the PV device could only achieve about 2.5 Gbps throughput. The Intel 82599 VF with CompSC achieved as much as a 282.66% higher throughput than the PV device. The impact on throughput occurred when the CPU utilization was higher than 200% (16s to 22s). In terms of downtime, the result these tests are similar to that of the Intel 82576 situations. The downtime of CompSC on the Intel 82599 VF was about 0.8s, which was 42.9% less than the downtime of the PV device, i.e. about 1.4s.

The test results of VMDq are presented in Figure 16. While VMDq support in Xen is currently abandoned, we found VMDq support on earlier version of Xen. Thus, in our VMDq tests, we used Xen 3.0 and Linux 2.6.18.8 with a PV guest virtual machine. We used an Intel 82598 NIC as the physical device of VMDq, because the Intel 82599 NIC is not supported in Linux 2.6.18.8. The migration time and downtime in the test was shorter than CompSC and PV scenarios due to the PV guest. The PV guest had advantages on migration, since the kernel of PV guest is modified for virtualization. The core issue of VMDq relates to throughput, which was about 5 Gbps. Although VMDq had larger throughput than the PV scenario, the throughput of the Intel 82599 VF with CompSC was 89.57% higher than VMDq.

6.6 Size of total hardware states and past communications

In Section 3.2, we mentioned that state replay may record large amounts of past communications, and introduced several optimizations in response. In this section, we list the amount of hardware states and past communications needed in our implementation with the Intel 82576 and Intel 82599 NIC.

Table 5. Size of total hardware states in our implementation

	Size (bytes)
States in IGBVF driver	88
VF-in-PF states in IGB driver	848
States in IXGBEVF driver	104
VF-in-PF states in IXGBE driver	326

According to Table 5, the total number of hardware states to be transferred during the migration is less than 1 kilobyte in both the IGBVF and IXGBEVF drivers. In a typical network environment, the network throughput is at least 100 Mbps. Consequently, the transmission cost of hardware states can safely be ignored.

7. Discussion

In this paper, we focus on the pass-through NIC, but in the design of CompSC, we focus on every pass-through device. While CompSC can be used for pass-through devices other than those in this work, not all devices might perform as this paper describes. One aspect that needs to be considered is the number of hardware states, which varies among different devices. In our evaluation, the number of hardware states of NIC is small, but some devices have tremendously large state capacities, such as graphic cards with large video memory. In modern graphic cards, video memory larger than 256 MB is quite common. With such devices, the transmission costs for device state is quite large and can have a large impact on the service downtime or even be a bottleneck. One potential solution would be to shut down some features of graphic cards such as 3D rendering before migration to reduce the total amount of the hardware states. Another aspect to be acknowledged is the cost for state replay. Since state replay only commits on invisible states, devices with many invisible states may have higher costs for state replay. Actually, IGBVF/IXGBEVF are examples of this phenomenon. Because the ring head register is invisible, the state replay may cost hundreds of MMIO. In our implementation, we use a method of shifting to avoid this large cost. The cost for state replay depends

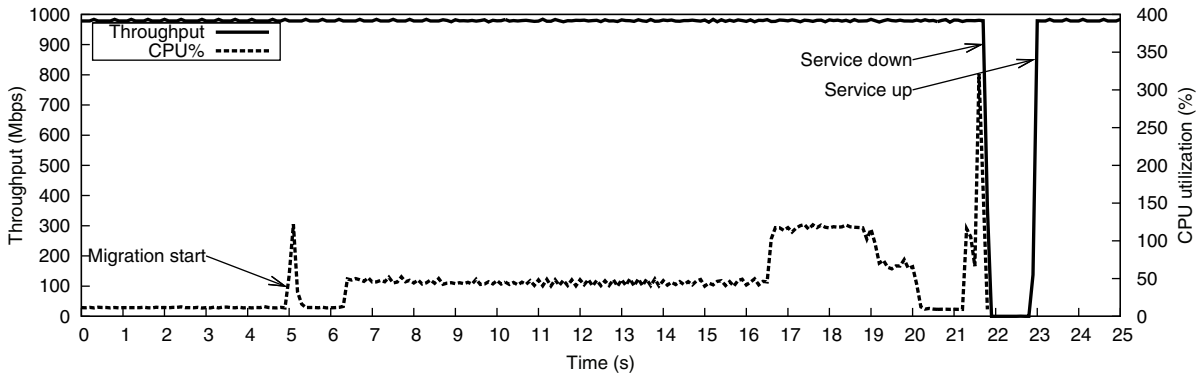

Figure 11. CompSC on Intel 82576: Throughput and CPU utilization during live migration

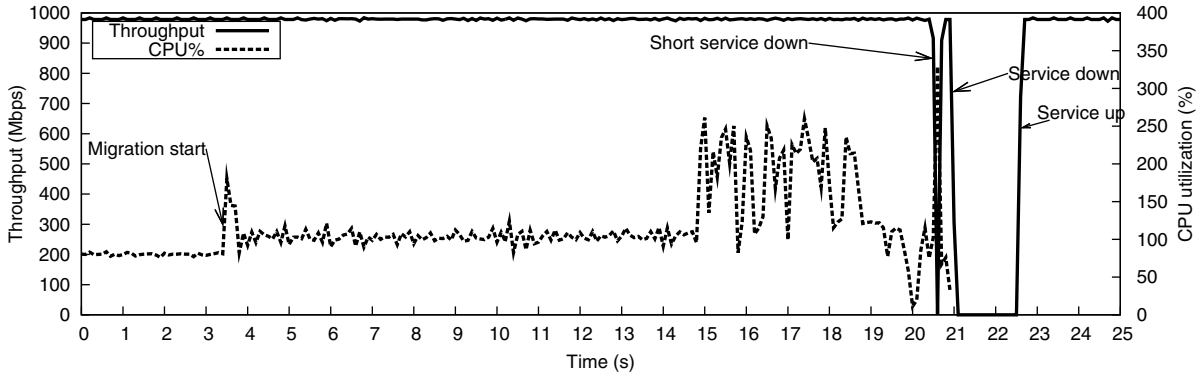

Figure 12. PV device on Intel 82576: Throughput and CPU utilization during live migration

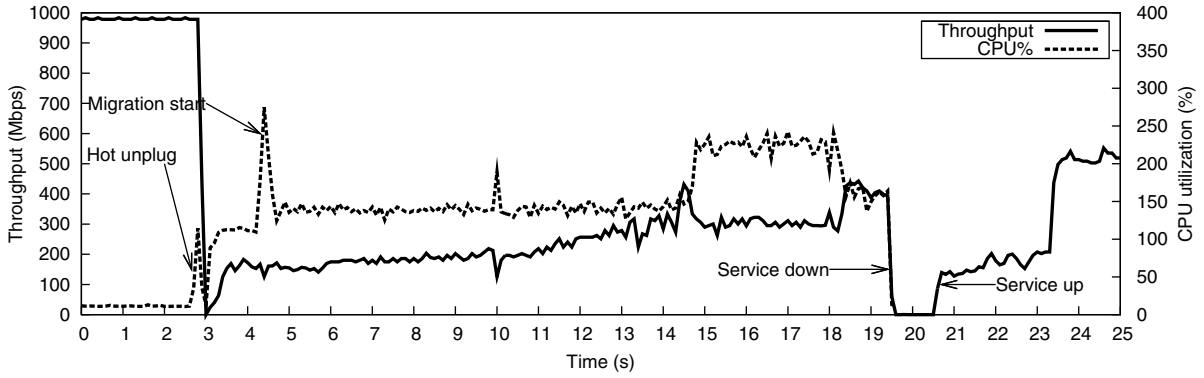

Figure 13. Bonding driver: Throughput and CPU utilization during live migration

on the hardware design of devices. Luckily, for most devices the cost for state replay is small because it is generally just the cost of device initialization.

CompSC can also be implemented on other hypervisors; no assumptions are made in this regard in the design of CompSC. The requirements for hypervisor of CompSC are: (1) Live migration support (2) Pass-through device support (3) Foreign page access. These features are common in today's hypervisors such as KVM [10] and VMware ESX [6]. Hopefully, the CompSC support of these hypervisors only need less than 1K LOC just like our implementation on Xen.

CompSC needs both driver changes and hypervisor changes. While this is somewhat of a limitation on deployment, CompSC does not need changes on virtual machine guest kernel, and the new driver is completely compatible with the original hypervisor and non-virtualized environments. In this respect, deployment is easy since one can safely use the new (CompSC) driver in old environments. Once the CompSC support of the hypervisor is settled, live migration is enabled. In terms of deployment, the bonding driver solution needs hypervisor changes, guest kernel changes, and a new guest driver. The convenience of the bonding driver solution is based on the fact that the Linux kernel already has the bonding

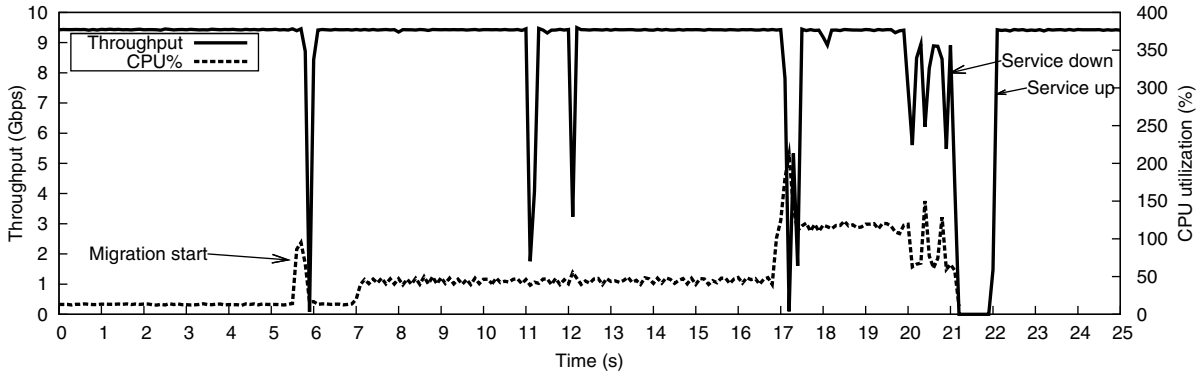

Figure 14. CompSC on Intel 82599: Throughput and CPU utilization during live migration

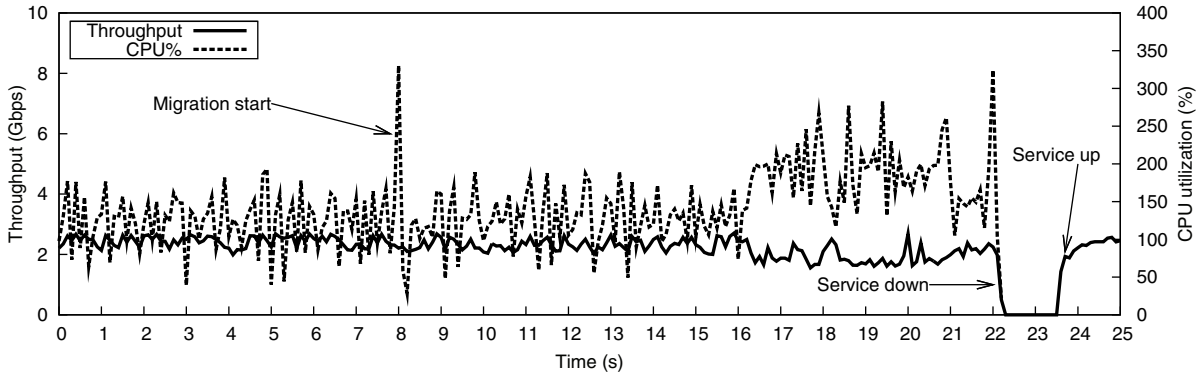

Figure 15. PV device on Intel 82599: Throughput and CPU utilization during live migration

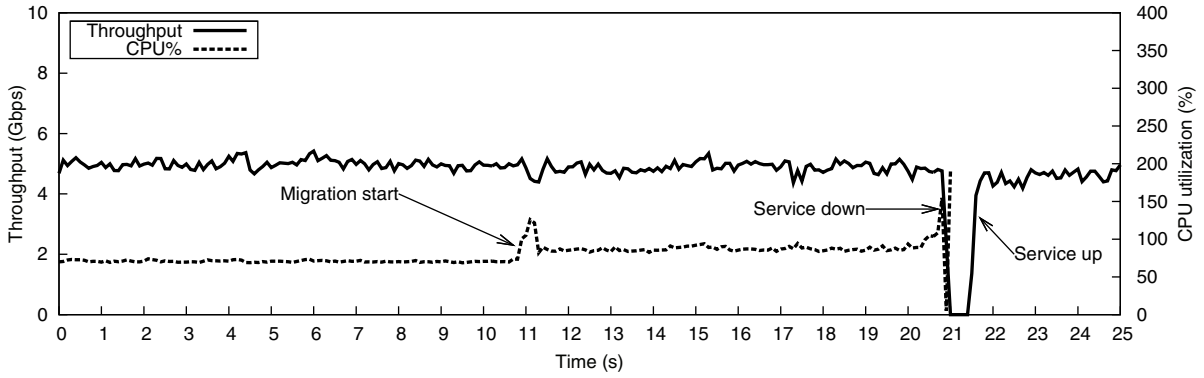

Figure 16. VMDq on Intel 82598: Throughput and CPU utilization during live migration

driver; the solution is hard to deploy on other OS such as Windows, however. NPIA needs hypervisor changes and a set of plug-in binaries. Compared to CompSC, every device NPIA supports has a brand new driver (plug-in binary). Furthermore, the new driver can only be used in NPIA environments. The VMDq solution is even worse: it needs hypervisor changes, guest kernel changes, and a pair of new drivers (A.K.A front-end driver and back-end driver). Overall, CompSC has deployment and usage advantages that outweigh these other solutions.

8. Conclusion

In this paper, we present CompSC: a solution of hardware state migration that achieves live migration support on pass-through network devices. With a synchronization mechanism between the device driver and the hypervisor, hardware is taken over by the hypervisor and hardware state saving is performed. Right after migration, the device driver restores the hardware state on the destination machine using knowledge of the device and saved states. Furthermore, a self-emulation layer inside the hypervisor is provided to ensure the accuracy of statistic registers.

With CompSC, the Intel 82599 VF enabled live migration support with a throughput 282.66% higher than PV network devices and 89.57% higher than VMDq. During live migration, the service downtime was 42.9% shorter than that of PV network devices. The performance impact of CompSC during run time is negligible. Lastly, CompSC needs minimal effort to implement and can easily be deployed on different NICs.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and suggestions. We also would like to thank Ian Pratt for his insight in the paper. This work has been supported by National Natural Science Foundation of China under Grant No. 61170050 and Grant No. 60973143.

References

[1] Kvm and linux source. http://git.kernel.org/.

[2] Network plugin architecture discussion on lkml. http://kerneltrap.org/mailarchive/linux-kernel/2010/5/4/4565952.

[3] Paravirtops. http://wiki.xen.org/xenwiki/XenParavirtOps.

[4] Spec web 2009. http://www.spec.org/web2009/.

[5] Virtual machine device queues. http://www.intel.com/content/www/us/en/network-adapters/gigabit-network-adapters/io-acceleration-technology-vmdq.html.

[6] Vmware esx. http://www.vmware.com/products/vsphere/esxi-and-esx/index.htm.

[7] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankran, I. Schoinas, R. Uhlig, B. Vembu, and J. Weigert. Intel virtualization technology for directed i/o. *Intel Technology Journal*, 10, Aug 2006. http://www.intel.com/technology/itj/2006/v10i3/.

[8] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 2–13, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, Feb. 2009.

[10] K. Avi. Kvm : The linux virtual machine monitor. *Proceedings of the Ottawa Linux Symposium*, 2007.

[11] 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. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003. ISSN 0163-5980.

[12] S. Berger, R. Cceres, K. A. Goldman, R. Perez, R. Sailer, and L. Doorn. vtpm: Virtualizing the trusted platform module. In *In USENIX Security*, pages 305–320, 2006.

[13] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.

[14] 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*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[15] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *In Proc. NSDI*, 2008.

[16] Y. Dong, Z. Yu, and G. Rose. Sr-iov networking in xen: architecture, design and implementation. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association.

[17] Y. Dong, Y. Chen, Z. Pan, J. Dai, and Y. Jiang. Renic: Architecture extension to sr-iov network for efficient high availability replication. In *In Proceedings of the 7th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC'12, 2012.

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

[19] *Intel 82576 Gigabit Ethernet Controller Datasheet*. Intel, . http://download.intel.com/design/network/datashts/82576_Datasheet.pdf.

[20] *Intel 82599 10 GbE Controller Datasheet*. Intel, . http://download.intel.com/design/network/datashts/82599_datasheet.pdf.

[21] A. Kadav and M. M. Swift. Live migration of direct-access devices. *SIGOPS Oper. Syst. Rev.*, 43:95–104, July 2009. ISSN 0163-5980.

[22] M. A. Kozuch, M. Kaminsky, and M. P. Ryan. Migration without virtualization. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[23] G. Liao, D. Guo, L. Bhuyan, and S. R. King. Software techniques to improve virtualized i/o performance on multi-core systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 161–170, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-346-4.

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

[25] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[26] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[27] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32:2000, 2000.

[28] A. B. Nagarajan and F. Mueller. Proactive fault tolerance for hpc with xen virtualization. In *In Proceedings of the 21st Annual International Conference on Supercomputing (ICS07*, pages 23–32. ACM Press, 2007.

[29] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[30] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association. ISBN 1-880446-09-X.

[31] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002. ISSN 0163-5980.

[32] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for linux vm. In *Ottawa Linux Symposium*, 2008.