LIVE MIGRATION IN BARE-METAL CLOUDS

Seminar Report

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF TECHNOLOGY

In

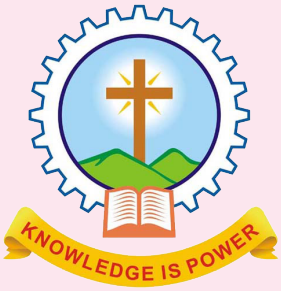
COMPUTER SCIENCE AND ENGINEERING

of

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

Submitted By

SARITHA K S



Department of Computer Science & Engineering
Mar Athanasius College Of Engineering Kothamangalam

LIVE MIGRATION IN BARE-METAL CLOUDS

Seminar Report

Submitted in partial fulfillment of the requirements for the award of degree of

BACHELOR OF TECHNOLOGY

In

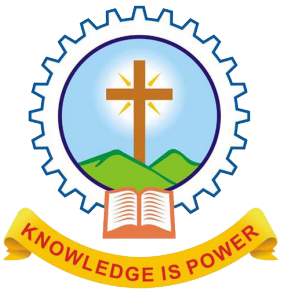
COMPUTER SCIENCE AND ENGINEERING

of

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

Submitted By

SARITHA K S



Department of Computer Science & Engineering
Mar Athanasius College Of Engineering Kothamangalam

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
MAR ATHANASIUS COLLEGE OF ENGINEERING
KOTHAMANGALAM

CERTIFICATE

This is to certify that the report entitled **Trajectory mining using uncertain sensor data** *submitted by* **Ms. SARITHA K S, Reg. No.LMAC15CS067** *towards partial fulfillment of the requirement for the award of Degree of Bachelor of Technology in Computer science and Engineering from APJ Abdul Kalam Technological University for December 2018 is a bonafide record of the seminar carried out by her under our supervision and guidance.*

.....	.....	.....
Prof. Joby George	**Prof. Neethu Subash**	**Dr. Surekha Mariam Varghese**
Faculty Guide	***Faculty Guide***	***Head Of Department***

Date:

Dept. Seal

ACKNOWLEDGEMENT

First and foremost, I sincerely thank the 'God Almighty' for his grace for the successful and timely completion of the seminar.

I express my sincere gratitude and thanks to Dr. Solly George, Principal and Dr. Surekha Mariam Varghese, Head Of the Department for providing the necessary facilities and their encouragement and support.

I owe special thanks to the staff-in-charge Prof. Joby george, Prof. Neethu Subash and Prof. Joby Anu Mathew for their corrections, suggestions and sincere efforts to co-ordinate the seminar under a tight schedule.

I express my sincere thanks to staff members in the Department of Computer Science and Engineering who have taken sincere efforts in helping me to conduct this seminar.

Finally, I would like to acknowledge the heartfelt efforts, comments, criticisms, co-operation and tremendous support given to me by my dear friends during the preparation of the seminar and also during the presentation without whose support this work would have been all the more difficult to accomplish.

ABSTRACT

Live migration allow a running operating system to be moved to another physical machine with negligible downtime. Live migration is commonly employed in Infrastructure-as-a-Clouds(IaaS). IaaS is inefficient due to overhead of live migration mechanism and OS dependency. Bare-metal cloud is a public cloud service where customer rends hardware resources from a remote service provider. To facilitate live migration in bare-metal clouds, an OS-independent set of techniques are introduced .BLMVisor is s live migration scheme for bare-metal clouds. BLMVisor exploits a very thin hypervisor to allow pass-through access to physical devices from the guest OS rather than virtualizing the devices. The BLMVisor captures, transfers, and reconstructs physical device states by monitoring access from the guest OS and controlling the physical devices with effective techniques. To minimize performance degradation, the hypervisor is mostly idle after completing live migration. BLMVisor is evaluated , compared to a commodity virtual machine monitor(VMM).

Contents

List of Figures

LIST OF ABBREVIATION

IaaS	Infrastructure-as-a-Service
SSD	Solid-State Drives
GPU	Graphics Processing Units
VM	Virtual Machines
VMM	Virtual Machine Monitor
APIC	Advanced Programmable Interrupt Controllers
VMCS	Virtual machine control structure
EOI	End-Of-Interrupt
ICW	Initial Control Words
NIC	Network Interface Card
PIC	Programmable Interrupt Controllers

Introduction

Live migration is commonly employed in Infrastructureas-a-Service clouds. Live migration provides greater management flexibility by allowing cloud vendors to move a running operating system to another physical machine with negligible downtime. For example, cloud vendors stop physical machines to perform routine maintenance,e.g., to replace failed devices and update firmware Live migration[1] allows cloud vendors to perform such maintenance without interrupting services. Similar to routine maintenance, to anticipate hardware faults, proactive fault tolerance , monitors various indicators, such as temperature and cooling fan states, and deals with faults proactively by replacing devices that are about to fail. Jiang et al. demonstrated that using live migration for dynamic replacement of instances optimizes data center efficiency. In addition to load balancing, live migration provides convenient functions for IaaS clouds.

Increasingly, cloud users with heavy workloads require high-end devices, such as GPU , SSD and InfiniBand network devices. However, the advantages of these high-end devices are limited by common software stacks, such as file systems and TCP/IP stacks. To remove these limitations, user-mode drivers, new OS architectures, and optimal applications for modern devices have been proposed. Unfortunately, typically, such approaches cannot be exploited in virtualized environments. For example, device virtualization conceals native device functions from the guest OS, and interrupt virtualization prevents the guest OS from using physical interrupt controllers. Therefore, exploiting such techniques in traditional IaaS clouds is difficult. To satisfy heavy workload requirements, bare-metal clouds have emerged as a new type of IaaS cloud. With bare-metal clouds, vendors lease physical rather than virtual machines. Bare-metal clouds are attractive for users who require guaranteed performance because they can avoid virtualization overhead and obtain maximum physical hardware performance. Thus, bare-metal

clouds are suitable for AI, big data, and high-performance computing, where virtualization overhead is not negligible,. Bare-metal clouds are also favored by security-sensitive users who are concerned about information leakage among VMs in traditional multi-tenant clouds. Therefore, bare-metal clouds have become widely available and are supported by leading cloud vendors, such as IBM.

Unfortunately, bare-metal clouds do not support live migration. Live migration requires the ability to copy the complete state of a source machine to a destination machine over a network. In conventional IaaS clouds, the machine being migrated is a VM and its states are stored in memory. Therefore, supporting live migration is easy. In fact, many existing virtualization software, such as VMware vSphere , Xen, and KVM support live migration.This paper proposes BLMVisor, a live migration scheme for bare-metal clouds. BLMVisor utilizes a very thin hypervisor that directly exposes physical hardware to the guest OS rather than virtualizing hardware devices. The guest OS almost completely controls the physical hardware with little virtualization overhead, thereby maximizing hardware performance. During live migration, the hypervisor carefully monitors and controls guest OS access to physical devices based on device specifications and captures, transfers, and reconstructs the physical device states from the source to the destination machines. After live migration is completed, the hypervisor does not interpose on access to devices from the guest OS, thereby eliminating virtualization overhead as much as possible.

BLMVisor's primary challenge is handling physical device states. The CPU and memory states are relatively easy to handle because modern CPUs have hardwareassisted virtualization support that allows software to save and restore internal processor states to and from memory. Therefore, these states can be migrated using existing live migration techniques. However, the internal states of various physical devices, such as NIC, timer devices, and interrupt controllers, cannot be accessed by software. Therefore, the hypervisor cannot directly save or restore such internal states. To address this problem, BLMVisor employs a set of techniques that capture and reconstruct the internal physical device states indirectly based on device specifications.

BLMVisor assumes that the source and destination machines have the same hardware specifications,i.e., the same CPU model, the same PCI devices in the same slots, and the same

types of other internal devices. In addition, the destination machine must have at least as much memory as the source machine. In bare-metal clouds, many cluster nodes commonly have identical hardware specifications and configurations. Therefore, preparing a spare machine for live migration is reasonable and practical. BLMVisor also assumes that the source and destination machines have dedicated NICs for the hypervisor to perform live migration. The primary contributions of this paper are as follows. 1) To facilitate live migration in bare-metal clouds, an OS-independent set of techniques to migrate the internal states of physical devices without incurring high overhead is introduced. 2)The implementation of a hypervisor based on BitVisor is described. The hypervisor can migrate an unmodified Linux OS running on machines with typical hardware, such as PIC ,APIC programmable interval timers, and Realtek RTL8169 NICs. 3) The proposed BLMVisor is evaluated compared to a commodity VMM[2]. The results demonstrate that BLMVisor incurs negligible overhead on network throughput and latency, and reduces overhead in Redis and MySQL benchmarks by 16.3

Related Works

2.1 VM migration

Live migration of VMs is supported by several major VMMs such as VMware's VMotion , XenSource's XenMotion, and KVM. Live migration enables cloud vendors to manage multiple server machines effectively. Proactive maintenance and proactive fault tolerance are crucial to maintain the health, reliability, and security of physical machines. Live migration of VMs[3] is supported in VMMs because all VM states, including those of virtual CPUs, virtual memory, and virtual devices, are stored in memory and are accessible from the VMM. Unfortunately, virtualization overhead is inevitable in commodity VMMs. Despite efforts to reduce such overhead, such VMMs incur non-negligible overhead in resource-intensive and highperformance computing applications. Consequently, bare-metal clouds have emerged as an attractive platform for such applications. Several studies have proposed live migration schemes for VMs with direct-access devices. When a guest OS can directly access physical devices, as in PCI pass-through, virtualization overhead is reduced significantly. However, migrating the physical hardware states of direct-access devices remains challenging. Nomad addresses problems related to location-dependent resources and packet drops during migration by modifying user-level libraries and device drivers in the guest OSs. Kadav and Swift introduced shadow drivers in the guest kernel that efficiently save and restore the states of device drivers. Pan et al. proposed CompSC , a scheme[4]to migrate device states by changing the device drivers in the guest OS and incorporating an emulation layer in the Xen hypervisor. Although these approaches have effectively solved the problems associated with migrating the physical hardware states of direct-access devices, they cannot avoid guest OS dependency. As mentioned previously, OS dependency is undesirable in live migration schemes

Several studies have addressed the OS dependency problem in live migration with direct-access devices. ReNIC is a hardware extension scheme for SR-IOV NICs that allows the VMM to import and export device states for live migration and checkpointing. This scheme does not require modification of the guest OS; however, it does require hardware modifications. Vagabond switches the VM network interface between SR-IOV-based and virtualized interfaces. Although the SR-IOV interface is dedicated to the VM, it is still virtualized to switch interfaces transparently from the guest OS. SRVM addressed the OSdependency problem by supporting a live migration scheme with SR-IOV devices. SR-IOV is a PCI device function that duplicates the device interface using the device itself. Typically, each duplicated interface is assigned to a VM, similar to PCI pass-through, and incurs negligible performance overhead. SRVM migrates physical device states using dirty memory tracking and SR-IOV VF checkpointing without guest OS support. However, SRVM only duplicates SR-IOV devices and does not dedicate all hardware to the guest OS because core devices, such as interrupt controllers and timers, must be virtualized so that a commodity VMM can coexist with the guest OS. In addition, SR-IOV is not supported in all devices and requires additional device drivers (VF drivers) in the guest OS.

2.2 Operating system migration

Live OS migration can be implemented by the OS without the support of virtualization systems. Hansen et al. proposed two prototype systems for OS-level migration[5]. The first system uses the L4 microkernel and an adapted version of Linux that runs as an L4 task (L4Linux). This system implements the pre-copy migration of L4Linux's OS memory image using a pagingvia-IPC mechanism and recursive L4 address spaces. The second system implements self-migration via a modified XenoLinux (a Linux Xen port). Note that these systems require significant modifications to the original Linux OS; therefore, they rely heavily on the guest OS. In addition, they only migrate memory states, i.e., physical device states are not migrated. To address the problems associated with migrating physical device states, Kozuch et al. exploited the suspend and resume features implemented in many modern OSs. They also migrated device and driver states in uniform device descriptors for each class of devices using

the export and import routines implemented in the drivers. OS-level migration eliminates virtualization overhead at runtime; however, it requires OS modification, which is impractical for bare-metal clouds. OS-level containers can also support live migration. Containers create a set of processes that are isolated from other processes and run on top of a single kernel instance. Containers are isolated and largely independent; thus, checkpoints and migration are possible. Containers also incur little virtualization overhead. However, the implementation heavily depends on the underlying kernel, which limits the choice of kernel

2.3 Process migration

Process migration, which allows the migration of processes from source to destination OSs with the cooperation of both OSs, was an active research area in the 1980s. However, process migration suffers from dependencies on the source OS, i.e., so-called residual dependencies, whereby the migrated processes require resources that are only available to the source OS. To address this, Zap introduced a process domain that provides a process group with a private namespace. Although this concept provides the members of the process group with the same virtualized view of the system, it requires modification of host OSs.

Design

The basic concept of live migration in BLMVisor is the same as that in commodity VMMs, i.e., transferring the entire machine state from the source machine to the destination machine over a network. However, as the target machines are physical rather than VMs, a new method to access the physical machine states is required. This section describes BLMVisor's architecture, discusses the essential physical device states to be migrated, and presents techniques for migrating unreadable and unwritable device states.

3.1 Overall architecture

Live migration scheme for bare-metal clouds should achieve sufficient performance because users require maximum hardware performance. The live migration scheme should also be OS-independent because user intervention in live migration should be avoided, the scope of supported OSs should be increased, and the installation of additional device drivers should be prevented. In BLMVisor, the performance goal is achieved using a thin hypervisor that limits the number of running guest OSs to one. This design greatly reduces virtualization overhead compared to commodity VMMs. A comparison of commodity VMM and BLMVisor architectures is shown in Fig. 3.1. In the commodity VMM , the guest OS runs on a VM and accesses virtual devices created by the VMM. Access to a virtual device is converted to an abstract interface of the backend driver in the VMM, and the backend driver issues an actual physical device access. A hardware interrupt from the physical device is first trapped by the backend driver. Then, the virtual device generates a virtual interrupt to notify the device driver of the guest OS. These multiple indirect accesses and interface conversions incur virtualization overhead. In contrast, in BLMVisor , the hypervisor allows the guest OS to manage physical devices di-

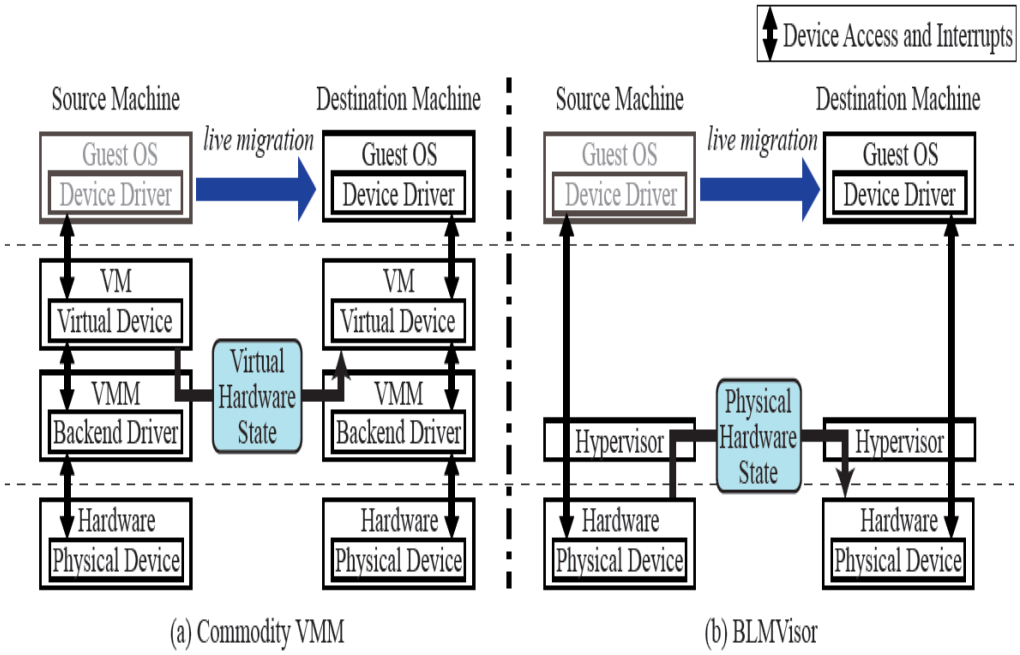


Figure 3.1: Comparison of commodity VMM and BLMVisor architectures

rectly via their native interfaces. In addition, the hypervisor does not interpose on hardware interrupts (not even timer interrupts). Instead, interrupts are delivered directly to and handled by the guest OS. This architecture eliminates device virtualization and VM scheduling, thereby providing execution environments that are equivalent to those of bare-metal instances.

This design greatly reduces virtualization overhead compared to commodity VMMs. A comparison of commodity VMM and BLMVisor architectures is shown. In the commodity VMM, the guest OS runs on a VM and accesses virtual devices created by the VMM. Access to a virtual device is converted to an abstract interface of the backend driver in the VMM, and the backend driver issues an actual physical device access. A hardware interrupt from the physical device is first trapped by the backend driver. Then, the virtual device generates a virtual interrupt to notify the device driver of the guest OS. These multiple indirect accesses and interface conversions incur virtualization overhead. In contrast, in BLMVisor, the hypervisor allows the guest OS to manage physical devices directly via their native interfaces. In addition, the hypervisor does not interpose on hardware interrupts (not even timer interrupts). Instead, interrupts are delivered directly to and handled by the guest OS. This architecture eliminates

device virtualization and VM scheduling, thereby providing execution environments that are equivalent to those of bare-metal instances.

In commodity VMMs, the hardware states are effectively the software states of the VM; therefore, the VMM can easily save and restore all hardware states. In contrast, in BLMVisor, the hardware states to be saved and restored are those of actual physical devices. For CPUs and memory, saving and restoring hardware states is not difficult because the hypervisor can employ hardware-assisted virtualization functions to save and restore CPU states, and it can simply read and write the entire memory data. However, physical device states are problematic because they can be unreadable or unwritable by software. To address this problem, the hypervisor inspects and controls the physical devices based on their specifications. The hypervisor captures the unreadable states of physical devices by monitoring access to the devices and their behaviors. Then, it reconstructs the unwritable states of physical devices by triggering a state transition that is associated with device processing. Although reconstructing all physical device states is difficult, reconstructing only the essential device states is sufficient to achieve live migration. The essential device states are described in the following.

3.2 Essential device states

In bare-metal live migration, two groups of essential physical device states must be migrated. The first is the configuration states. OSs typically modify these states to change device behaviors. For example, an OS may configure a NIC device to use a 1,000-Mbps link rather than a 100-Mbps link. The configuration states must be transferred to reproduce the same configuration on the destination machine. The second group is the processing states, which are updated by the devices themselves. For example, after a reset, a device updates its status to indicate its readiness.

Both states must be transferred to enable the guest OS to run continuously after live migration. If a configuration state is unavailable, the guest OS will not recognize configuration changes and the driver may request unreasonable device operations after a configuration change. For example, assume a driver requests a transfer rate of 1000 Mbps for a device set to 100 Mbps. In this case, many packets will be dropped, and the connection will become

unstable. If a processing state is absent, the states managed by the driver will not match those managed by the device. For example, if the reset state is not transferred, the driver may attempt to use the device before the reset is complete. Note that not all states are essential to run the OS continuously. For example, statistical states (e.g., statistical values, such as the number of received packets) do not necessarily require migration because their values are generally not evaluated after live migration, as such values are specific to each source and destination machine. For applications that require migration of statistical values of the source machine, the hypervisor can migrate statistical states by virtualizing their access; however, this can incur overhead. Other non-essential states for live migration are interrupt states. Interrupts are tightly coupled with physical device states; thus, migrating them would be pointless or possibly even damaging. For example, a non-maskable interrupt issued by a failed physical device at the source must not be migrated to the physical destination device because that device has not failed.

Command register values also represent a non-essential state. Command registers act as device interfaces that receive commands, and their values are not directly related to the internal states of the device. For example, I/O APIC has an end-of-interrupt notification register, i.e., a register to notify the completion of interrupt handling in an interrupt handler. Its value does not have meaning; therefore, migrating its value is also meaningless. Another example is a register in a NIC to request the start of packet transmission. In this case, a timing problem in which device state migration begins just before or after issuing a command could occur. If the migration begins just before issuing the command, the command will be issued on the destination machine immediately after the migration is finished as if the machine had not changed. If migration begins just after issuing this command, the source device starts the operation and sets a status to indicate this transmission, which will then be migrated by the hypervisor. Therefore, the command does not result in the timing problem.

3.3 Capturing unreadable states

There are two types of unreadable states in physical devices, i.e., write-only register states and internal states. Software can write values to write-only registers; however, the written

values cannot be read. Such states are typically configuration states updated by the OS. Internal states, which are typically processing states updated by the device, cannot be accessed by software. The hypervisor cannot save such write-only registers or internal states; thus, these require a different approach. To capture write-only register states, the hypervisor constantly monitors the write I/Os to these registers. When the guest OS issues an I/O write request to an I/O address, the hypervisor intercepts the request, stores the written value in memory, and forwards it to the physical hardware. During live migration, the hypervisor obtains the last written value of each monitored I/O address from memory and sends it to the destination hypervisor. The I/O addresses to be monitored can be determined by manually inspecting the device specifications. Although this process is device specific, it only requires identification of the write-only register addresses. Therefore, it is much simpler than writing device drivers. Although intercepting I/Os may incur some overhead, such overhead will be negligible because there are very few write-only registers and they are accessed infrequently in modern computer architectures. An example of I/O addresses to be monitored is discussed in Section 4.4 To capture internal states, the hypervisor controls physical devices and inspects their behaviors during live migration. From these behaviors, the hypervisor can estimate the device states indirectly. These operations increase downtime negligibly.

3.4 Reconstructing unwritable states

The hypervisor reconstructs unwritable states indirectly by controlling the physical devices such that they cause internal state transitions to become the desired states. For example, to reconstruct the internal register of a NIC, the destination hypervisor sends dummy packets that change the register value. This approach depends on device specifications and controlling internal device states may appear non-trivial. Fortunately, such states are relatively rare, and it is theoretically possible to set internal device states to the desired states because the current states are the result of existing software control. The implementation of these techniques is feasible and relatively easy in real physical devices.

Implementation

This section describes the implementation of BLMVisor, which is based on BitVisor. BLMVisor uses the pre-copy method, which comprises pre-copy and stop-and-copy phases. In the pre-copy phase, the hypervisor transmits memory data from the source to the destination machine in background while the guest OS is running on the source machine. When the amount of remaining memory becomes sufficiently small, the stop-and-copy phase begins. In this phase, the source hypervisor stops the OS and transmits the remaining memory data, CPU states, and device states. After setting the transmitted states on the destination machine, the destination hypervisor resumes OS execution. The stop-and-copy phase incurs little downtime (typically a few seconds at most). Note that, in addition to the pre-copy method, the post-copy method can also be supported. The following sections describe the implementation to migrate CPU, memory, and storage states using a common algorithm, and describe the implementation for migration of physical device states. Then, issues related to multi-core systems and the implementation status are presented.

4.1 Migrating CPU states

The CPU is assumed to have a hardware-assisted virtualization function that can save and restore processor states from memory, such as Intel VT-x or AMD-V. For example, Intel VT-x supports a memory structure called the virtual machine control structues, which retains the guest and host processor states. VMCS contains the processor states required for migration, including internal register values that cannot be accessed by normal instructions. As the hypervisor can read the guest processor states on the source machine and write them on the destination machine using the VMCS, it can easily migrate CPU states. Generalpurpose

registers and most model-specific registers are not included in the VMCS; however, they are accessible via software. Although the current implementation only supports Intel CPUs, AMD CPUs can also be supported.

4.2 Migrating memory data

Memory data are migrated using the pre-copy scheme, in which the hypervisor first transfers all memory pages from the source to destination machine in background. The source hypervisor then detects pages that have been dirtied by the guest OS during transfer and re-transfers them to the destination hypervisor. This step iterates until the number of dirty pages becomes sufficiently small. The source hypervisor then stops execution of the guest OS and transfers the remaining dirty pages. It is assumed that a dedicated NIC is available for migration to prevent performance degradation during live migration by transferring large amounts of memory data across the network. The current implementation has two limitations relative to the conventional pre-copy method. The first is that the current implementation transfers all memory data; essential memory pages are not identified. Therefore, the amount of transferred memory can be greater than that in existing VMMs. According to our experiment, the amount of essential memory just after booting an OS was 491 MiB and that after running Redis was 3,581 MiB on a 4 GiB memory machine used in Section 5. Therefore, although there is room for saving up to 3.6 GiB of memory transfer, savings are not that much on busy machines. The second is that the pre-copy transmission speed and threshold are fixed. In the current implementation, the transmission speed is approximately 1 Gbps and the threshold is 64 MiB. Changing transmission speed dynamically could improve the guest OS performance during migration. These limitations are not architectural and can be eliminated by additional engineering efforts. Note that these limitations only affect performance during live migration and total migration time, i.e., they do not degrade performance during normal execution.

4.3 Migrating storage states

The current implementation assumes that the storage device is an iSCSI device. Therefore, storage content can be shared between the source and destination machines without being copied. During live migration, the guest OS can access the storage device continuously because the iSCSI server address does not change. This is a standard configuration in live migration systems to avoid long downtime. However, it is possible to support live migration of storage devices such as HDDs and SSDs. The device states of the host controller can be migrated using the method described in the following. For storage data, it is possible to apply a background storage copy technique.

4.4 Capturing physical device states

During live migration, the source hypervisor must capture the physical device states. To achieve this, the device registers are classified as readable, write-only, and internal registers. The values of readable registers are read by the hypervisor during live migration, those of write-only registers can be obtained by monitoring access to the registers in the hypervisor, and those of internal registers are captured and estimated by monitoring the device's behavior in the stop-and-copy phase.

In modern PC architectures, device registers are accessed via programmed I/O or memory-mapped I/O . The hypervisor can intercept PIO accesses using hardware-assisted virtualization technology. A CPU with virtualization technology manages a bitmap of PIO addresses that determines whether the hypervisor should intercept accesses to specified addresses. MMIO access is intercepted by the EPT. The hypervisor can configure the EPT such that access to specified MMIO pages causes a page fault (i.e., an EPT violation). An EPT violation transfers control from the guest OS to the hypervisor; thus, the hypervisor can intercept access to specified MMIO pages and interpose on the MMIO access.

4.4.1 Obtaining write-only register values

Registers in PITs are an example of write-only registers. PITs are used by OSs to generate periodical timer interrupts. The frequency-setting register is write-only; thus, the hypervisor obtains its value by monitoring PIO access to the register. For some device registers, the hypervisor must monitor a sequence of I/O accesses because the type of I/O access depends on the previous I/O accesses. PICs are an example of such devices. In PIC initialization, four values, referred to as the initial control words (ICW), are written to the PIC by software. The first ICW (ICW1) is written to port 0x20, and the remaining three ICWs (ICW2–ICW4) are written sequentially to port 0x21. Unfortunately, the ICWs are written to write-only registers; therefore, to obtain their values, the hypervisor must monitor the sequence of accesses to ports 0x20 and 0x21. Note that monitoring write I/O access during normal execution incurs some overhead; however, there are only a few registers to monitor. TABLE 1 lists the addresses (I/O ports) that should be monitored by the hypervisor. All addresses are PIO ports for legacy devices (i.e., no MMIO addresses). Legacy devices are assigned write-only registers because only a 64-KiB address space is available in x86 PIO and I/O ports are valuable resources. Consequently, these devices assign two different registers to a single port, i.e., read-only and write-only registers, to reduce the number of I/O ports for register access. On the other hand, MMIO address spaces are vast and no address is shared between different functions. Fortunately, legacy devices in recent commodity OSs are used only at boot time; therefore, constant monitoring of the I/O accesses in such devices is not required in modern machines.

4.4.2 Obtaining Internal register register values

An example of internal registers is those employed in the Realtek RTL8169 NIC. In this device, internal registers are related to packet reception and transmission operations. With the RTL8169, the NIC and OS send network packets to each other via memory buffers. When a packet is received from the network, the NIC initially stores the packet in a buffer, and then, the OS reads the packet in the buffer. When sending a packet, the NIC and OS perform reverse operations. Here, the buffers are organized into rings and are managed by descriptors

in memory. There are two ring buffers, i.e., reception (RX) and transmission (TX) ring buffers. Each entry of the descriptors contains an OWN bit and a pointer to a buffer. If the OWN bit is set, the entry is owned by the NIC and the corresponding buffer is used to send or receive a packet. If the OWN bit is cleared, the entry is owned by the software (device drivers), and the corresponding buffer can be used to read or write packet data. The NIC clears the OWN bit when it receives a packet and stores it in the RX buffer and when it sends a packet stored in the TX buffer. The OS will set the OWN bit when it finishes processing the packet in the RX buffer and when it stores a packet to be sent in the TX buffer. The head of the RX ring buffers is pointed to by the RX head pointer and the tail of the TX ring buffers is pointed to by the TX tail pointer. Note that both pointers are stored in the NIC's internal registers and cannot be accessed directly from software.

With the RTL8169 [6], the source hypervisor sends and receives dummy packets in cooperation with the destination hypervisor to obtain the internal RX head pointer and TX tail pointer values . In the figure 4.1, the hypervisor first stops the guest OS in the stop-and-copy phase, and then increases the number of descriptor entries by two and fills the descriptors with entries to send or receive dummy packets, except for the last one. In the middle , all entries except for the last one are "N," meaning that the OWN bits are set and owned by the NIC. Then, the hypervisor sends a request to the NIC to send and receive packets. The NIC processes the descriptor entries sequentially, starting from the entry pointed at by the internal pointer and stopping at the last entry . Here, the hypervisor can obtain the original internal pointer value by finding the boundary between the entries where the OWN bit is set and cleared.

4.5 Reconstructing physical device states

During live migration, the destination hypervisor must reconstruct the device states. Device registers are classified as writable or unwritable. Writable registers can simply be written by the hypervisor, and unwritable registers can be set by the hypervisor by carefully controlling the device. In the Realtek RTL8169, the internal RX head pointer and TX tail pointer are stored in unwritable registers. To control the values of these registers, the destination hypervisor again

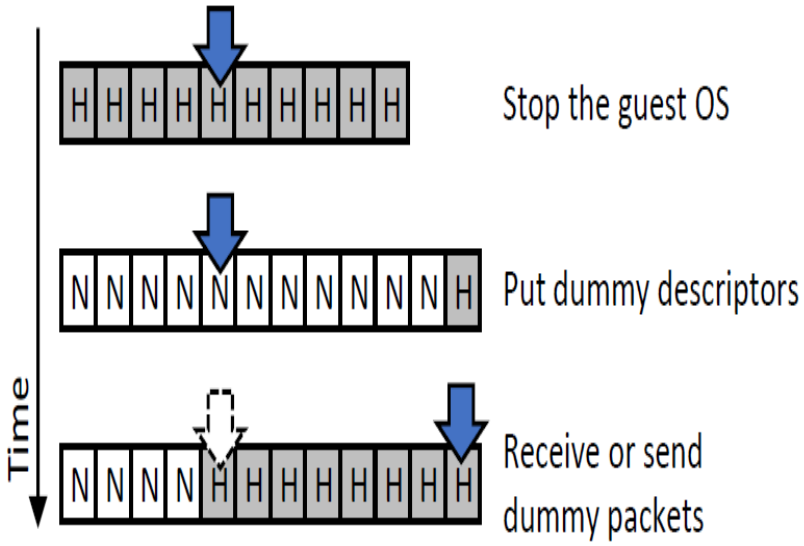


Figure 4.1: Obtaining internal pointer in Realtek RTL8169 by sending/receiving dummy packets. “N” and “H” entries belong to the NIC and host, respectively. The original internal pointer points to the fifth entry.

sends and receives the required number of dummy packets.

In the figure 4,2, the destination hypervisor first sets the same number of dummy entries in the descriptor as the value of the internal pointer to be set. For each entry, the OWN bit is set, and the buffer is prepared for a dummy packet. the hypervisor attempts to set the internal pointer to five. Then, the hypervisor sends a request to the NIC to send and receive packets. The NIC will send and receive dummy packets until the internal pointer is incremented to the desired value . To avoid receiving unexpected packets, the NIC is configured to receive only unicast packets whose destination MAC address matches that of the NIC; thus, the NIC does not receive multicast or broadcast packets. After setting the internal pointer, the hypervisor restores the original descriptors and buffers . Finally, the hypervisors on the source and destination machines exchange the MAC addresses of their respective NICs.

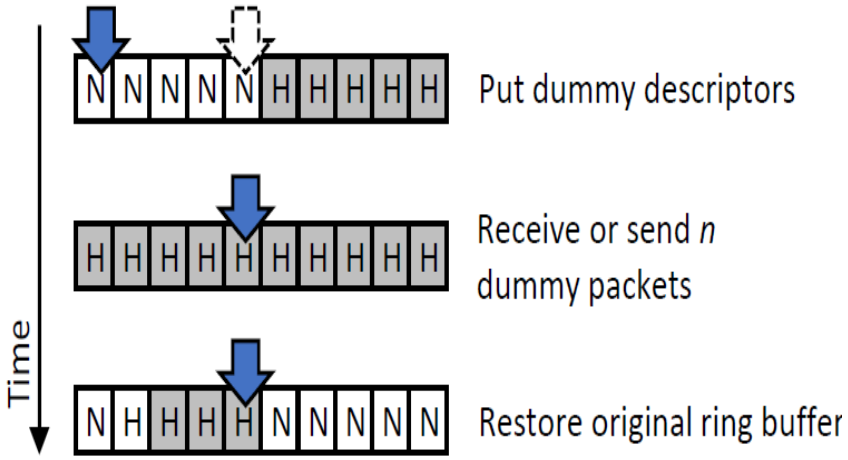


Figure 4.2: Setting an internal pointer in the Realtek RTL8169 by sending/receiving dummy packets. "N" and "H" entries belong to the NIC and the host, respectively.

4.6 Supporting multi-core system

There are several issues related to supporting multi-core systems. The first is which core should perform memory transfer in the pre-copy phase. A dedicated NIC is assigned to the hypervisor to transfer memory data. In multi-core systems, if all cores attempt to transfer memory data via this single NIC, NIC lock contention occurs; thus, cores are frequently blocked, and performance is reduced significantly. To avoid lock contention[7], only a single core is assigned to use the NIC. To reduce the performance impact on the guest OS, an idle core should be selected dynamically. However, due to implementation complexity, the current implementation exploits a fixed core for migration. This complexity arises because the architecture does not include a virtual CPU scheduler and physical CPUs are directly exposed to the guest OS.

The second issue related to handling dirty bits in multiple cores. In the pre-copy phase, the hypervisor must periodically find dirty pages by monitoring dirty bits in the entries of the EPT. Unfortunately, EPT entries of frequently accessed pages (including their dirty bits) are cached in the translation lookaside buffer (TLB) of each core. Therefore, a core may not be able to read a dirty bit in other cores. TLB shootdown (flushing all cores' TLBs) synchronizes

the EPT entries; however, frequent TLB shootdown degrades performance. Therefore, in the current implementation, each core records its own dirty bits in shared memory.

The third issue is how the hypervisor obtains control at sufficient frequency. In single-core systems, interrupts are controlled by a PIC. As described in Section 4.4.2, the hypervisor must intercept write I/O access to some PIC registers to capture the write-only states of the PIC. To intercept I/O access to a register, the hypervisor configures the CPU to cause a "VM exit" event upon access to the register's port address. A VM exit is an event where the CPU passes control from the guest OS to the hypervisor. One of the write-only states of the PIC is stored in a register that shares the port address with the EOI register. Therefore, the hypervisor must configure the CPU to cause a VM exit event when accessing this shared port address to intercept the write I/O access to the PIC register. Since the EOI register is accessed frequently, VM exits occur frequently. In contrast, interrupts in multi-core systems are controlled by the IOAPIC, local APIC, and MSI(-x) mechanisms. Since these mechanisms have no write-only states, the hypervisor does not need to intercept write I/O access to them. As a result, the number of VM exits is reduced significantly (Section 5.2.2). Note that VM exits are costly operations; thus, fewer VM exits are better in terms of performance.

4.7 Performance Evaluation

This section presents an evaluation of BLMVisor that measured performance compared to a bare-metal machine and KVM.

4.7.1 Setup

In this evaluation, two physical machines with the same specifications were used as the source and destination machines for live migration. Each machine had an Intel Core i7-4790K CPU (4.00 GHz, four cores), 4 GiB memory, and Realtek RTL8169 and Intel PRO/1000 NICs. The Realtek NIC was used by the guest OS and the Intel NIC was used by the hypervisor to transfer the machine states. The guest OS was Linux 3.4, unless specified, configured with a boot option to exclude the C-state and enter the idle state via polling rather than executing

hlt or mwait instructions. This configuration was designed to maximize system performance. Note that no Linux code was modified, and no additional software was installed. For some experiments, Windows Server 2016 was used as the guest OS. The evaluation was performed using a bare-metal machine ("Baremetal"), BLMVisor, and KVM configured to assign the same number of virtual cores as physical cores to a guest OS. The assignment of physical CPU cores to vCPUs was fixed. KVM was configured in two ways, i.e., with a PCI pass-through NIC ("KVM (pass)") and with a Virtio NIC ("KVM (virt)"). Note that KVM in the passthrough configuration cannot perform live migration. The client in the benchmarks and the iSCSI server each had an AMD Phenom II X6 1090T CPU (3.2 GHz), 8 GiB memory, and a Broadcom BCM57788 NIC. Crucial CT512MX100SSD1 SSDs[8]were used as storage devices by the guest OSs. All machines were connected via a gigabit Ethernet switch. Note that hardware interrupt distribution is not supported by these Intel machines; thus, for fair comparison, interrupt distribution among the VMs in KVM was disabled.

4.7.2 Performance during normal execution

Performance during normal execution was measured, including network throughput, latency, number of VM exits, memory consumption, system benchmarks, and a database benchmark.

Network throughput and latency

First, network throughput and latency were measured using Netperf. A Netperf client ran on the AMD machine and a server ran on the Intel machine. The TCP and UDP throughput was measured while changing the packet size from 1 to 2048 bytes, and latency was measured as the round-trip time of a packet with a 1-byte payload. In each configuration throughput and latency were measured 10 times, and the average and standard deviation were calculated.

In the Fig. 4.3 ,it shows network throughput of TCP/UDP workloads. The measured values were normalized by the "Baremetal" value. The TCP inbound throughput did not differ significantly among the systems because multiple small packets were aggregated into a single

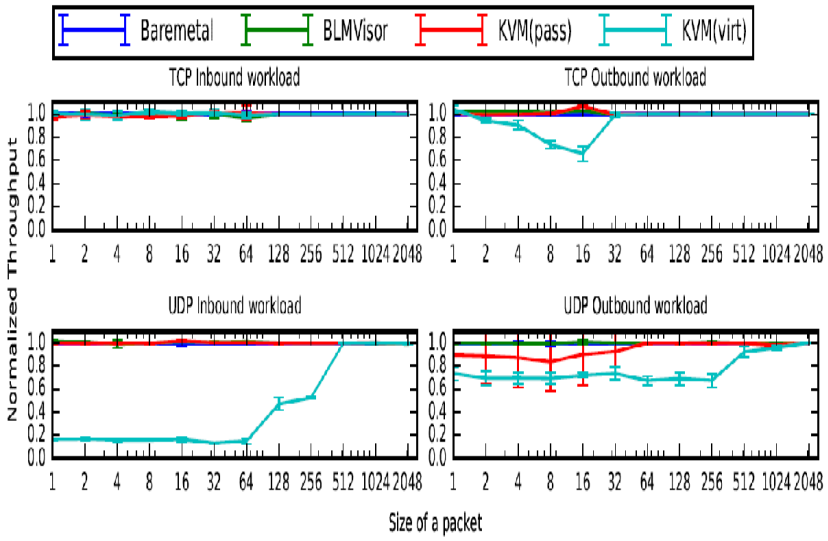


Figure 4.3: Network throughput of TCP/UDP inbound/outbound workload with 1 to 2048-byte packets (normalized to bare-metal)

packet in the Linux TCP stack using Nagle's algorithm. In the TCP outbound throughput, the "KVM (virt)" overhead increased as the packet size increased from 1 to 16 bytes. Here, when the packet size was 1 byte, multiple packets were merged into a single packet and the number of packets sent from the NIC was small. When the packet size increased to 16 bytes, packets could not be merged into a single packet and multiple packets were sent from the NIC. This caused overhead in packet processing and interrupt handling for multiple ACKs. When the packet size was 32 bytes, the TCP window size increased, and multiple packets were acknowledged by a single packet, thereby reducing the processing overhead of packet and interrupt handling.

Number of VM exits

The number of VM exits was counted to analyze the overhead of the virtualization layers. The Netperf latency workload was used in this analysis. For this measurement, BLMVisor was modified to have a counter incremented by the VM exit handler and a VMCALL handler to read the counter value. The number of VM exits in KVM was measured using the kvm stat command. In all configurations, the VM exits of all cores were counted.In addition to the number of VM exits, the CPU cycles per second spent in VM exits were counted

by reading the time stamp counter at each VM exit and VM entry, excluding cycles spent for those operations. Here, 12,790.5 cycles were spent in "EPT violation," 6,981.1 cycles were spent in "Exception on NMI," and 19,771.6 cycles were spent in total (1.24 µs per core). This is considered negligible overhead.

Memory consumption

Memory is an important resource in memory-intensive workloads. However, virtualization layers consume some memory for their own purposes. To evaluate this memory consumption, the amount of memory available to the guest OS was measured. In KVM, the host OS was configured without a swap to avoid over-committing, and a VM was configured to use as much memory as possible. Note that "KVM (pass)" and "KVM (virt)" had the same memory configuration (both are shown as "KVM"). Compared to "Baremetal," the total memory in "BLMVisor" and "KVM" was reduced by 131 MiB and 307 MiB, respectively.The figure 4.4 shows the redis throughput with various workloads. Thus, BLMVisor's simple architecture contributes to consuming less memory consumption, which will increase the performance of memory-intensive applications.

System benchmarks

CPU, memory, and file I/O performance was measured using Sysbench 1.0. These tests were run in four threads (number of CPU cores). Since the storage was iSCSI, file I/O was performed via the network. Here, performance was measured 10 times for each configuration and plotted as the average and standard deviation of the execution time.. The execution times were similar for all systems because there were few VM exits in the CPU-intensive workloads. The total memory written was 100 GiB. Here, "BLMVisor," "KVM (pass)," and "KVM (virt)" showed 2.21respectively. The overhead was caused by additional address translations from the guest to host physical addresses. With KVM, such address translations are indispensable for the VMM to coexist with VMs on a single machine. On the other hand. the BLMVisor's hypervisor does not necessarily require address translations because it uses identity mapping. The hypervisor only requires EPT to trace dirty pages in the pre-copy phase. Therefore, overhead

in BLMVisor can be mitigated.

Database benchmark

To evaluate the performance of real server applications, the throughput of Redis (version 3.0.0) and the execution time of an SQL workload on MySQL with a Sysbench OLTP test client were measured.The throughputs showed a similar trend in all workloads, i.e., as the number of threads increased, the difference in overhead between the systems increased. With 64 threads, “BLMVisor” incurred a 4.6–11.0incurred a 10.3–25.5a 41.0–49.5workload with 64 threads with all systems. This workload involved heavy memory access, which caused many TLB misses. Such TLB misses required EPT address translations, which degrades system performance.

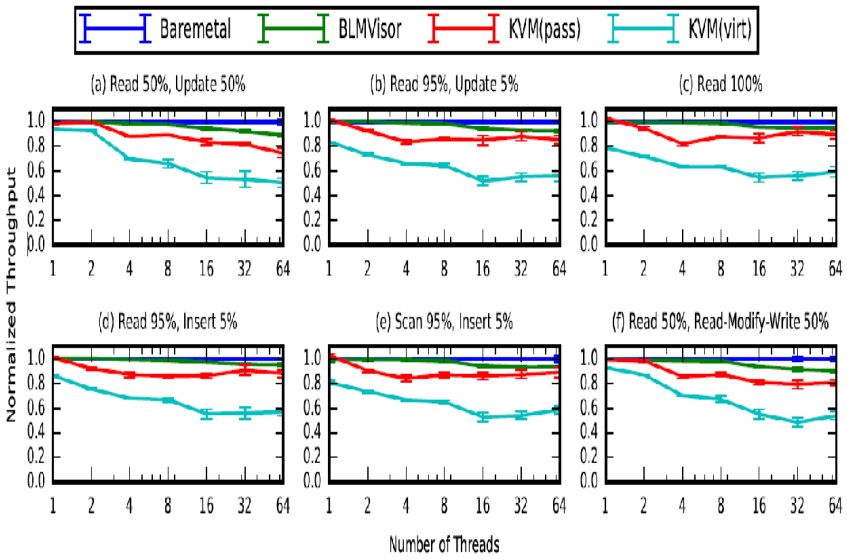


Figure 4.4: Redis throughput with various workloads. Horizontal and vertical axes denote the number of client threads and the throughput in number of operations per second (normalized by the bare-metal throughput), respectively.)

In Linux, “BLMVisor” increased execution time by 1.6in the worst case (eight threads). In contrast, “KVM (pass)” increased execution time by 2.1–44.7increased execution time by 9.0thread) and 63.6“BLMVisor” increased execution time by less than 8with 16 threads or

fewer, and increased execution time by 19.0execution time by 20.7–75.2execution time by 34.3–111.5threads and the worst case was 32 threads). This workload was I/O intensive because it stored the result of each transaction in storage. In addition, the storage in this experiment was iSCSI connected via the network. Consequently, there were many network I/Os in the workload. Therefore, the throughput with KVM was degraded significantly by I/O emulation and interrupt handling. In contrast, BLMVisor incurred much lower overhead with this workload.

4.7.3 Performance during live migration

Performance during live migration was measured, including network throughput, VM exit number, and downtime.

Network throughput

To demonstrate that BLMVisor can perform live migration on a bare-metal machine and verify its performance, network throughput during live migration of Linux and Windows were measured. The current implementation fixes the memory transfer speed at approximately 1 Gbps. Therefore, this measurement includes the maximum overhead of the pre-copy phase.The Netperf client measured the throughput at 100-ms intervals. Live migration began three seconds after initiating the Netperf benchmark. The precopy phase took approximately 36 and 44 seconds for Linux and Windows, respectively. The stop-and-copy phase was less than one second for both Linux and Windows. Thereafter, network throughput returned to the same level as before live migration began. This result confirms that BLMVisor had no critical performance impact on bare-metal instances during live migration. Next, multi-thread execution times were measured for all cores during normal execution and live migration. Here, the number of threads was four because the CPU has four cores. The execution times were 2.04 and 2.22 (8.8 seconds during normal execution and live migration, respectively. Although the performance of the pre-copy core was reduced significantly, the overall performance degradation was not excessive because the other cores mitigated the performance degradation.

Downtime

Netperf was used to measure downtime during live migration with BLMVisor. The average, maximum, and standard deviation of the downtime was 0.861, 1.15, and 0.104 seconds, respectively. The downtime was primarily due to the memory copy operation. In the current implementation, the pre-copy threshold is 64 MiB; thus, the hypervisor transmits 64 MiB of memory in the stop-and-copy phase. Here, with the 1-Gbps NIC, sending 64 MiB required 0.5 second. Note that downtime was also caused by transferring the CPU and device states, setting the states (including sending and receiving dummy packets), and updating the MAC table in the network switch. To analyze the downtime in detail, the number of retransmission packets during the stop-and-copy phase was counted using tshark (the CLI version of WireShark). In the stop-and-copy phase, 31 1,448-byte packets (44,888 bytes in total) were retransmitted. These retransmission packets consisted of two TCP retransmission packets, one fast retransmission packet, and 28 selective ACK response packets. Note that the retransmitted packets were divided into smaller packets. The results demonstrate that the number of retransmission packets is in a reasonable range.

Conclusion

This paper has presented BLMVisor, a live migration scheme for bare-metal clouds. BLMVisor exploits a very thin hypervisor to allow pass-through access to physical devices from the guest OS. To perform live migration, the hypervisor captures and reconstructs physical device states, including both unreadable and unwritable states. Unreadable states are captured indirectly by monitoring accesses to device registers and the behaviors of the given device. Unwritable states are reconstructed indirectly by carefully controlling the device. A prototype implementation based on BitVisor supports live migration of PIC and a Realtek RTL8169 NIC, in addition to the CPUs and memory. Performance was evaluated in a series of experiments that confirmed BLMVisor achieves performance that is comparable to that of a bare-metal machine. In future, the overhead of memory-intensive workloads will be reduced by dynamically starting and stopping the hypervisor.

REFERENCES

[1] 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 Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2005, pp. 273–286.

[2] [13] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High Performance VMM-bypass I/O in Virtual Machines," in Proceedings of the 2006 USENIX Annual Technical Conference, Jun. 2006, pp. 29–42.

[3]] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast Transparent Migration for Virtual Machines," in Proceedings of the USENIX Annual Technical Conference, 2005, pp. 25–25.

[4]] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy Live Migration of Virtual Machines," SIGOPS Operating Systems Review, vol. 43, no. 3, pp. 14–26, Jul. 2009.

[5] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda, "Nomad: Migrating OS-bypass Networks in Virtual Machines," in Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE 2007). ACM, 2007, pp. 158–168.

[6] NVM Express Revision 1.2.1, 2016. [Online]. Available: http://www.nvmexpress.org/specifications/

[7] "3D XPointTM: A Breakthrough in Non-Volatile Memory Technology." [Online]. Available: http://www. intel.com/content/www/us/en/architecture-and-technology/ intel-micron-3d-xpoint-webcast.html

[8] T. Merrifield and H. R. Taheri, "Performance Implications of Extended Page Tables on Virtualized x86 Processors," in Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016), 2016, pp. 25–35.