# Live Migration in Bare-metal Clouds

Takaaki Fukai, *Member, IEEE,* Takahiro Shinagawa, *Member, IEEE,* and Kazuhiko Kato, *Member, IEEE*

**Abstract**—Live migration allows a running operating system (OS) to be moved to another physical machine with negligible downtime. Unfortunately, live migration is not supported in bare-metal clouds, which lease physical machines rather than virtual machines to offer maximum hardware performance. Since bare-metal clouds have no virtualization software, implementing live migration is difficult. Previous studies have proposed OS-level live migration; however, to prevent user intervention and broaden OS choices, live migration should be OS-independent. In addition, the overhead of live migration mechanisms should be as low as possible. This paper introduces BLMVisor, a live migration scheme for bare-metal clouds. To achieve OS-independent and lightweight live migration, BLMVisor utilizes a very thin hypervisor that exposes physical hardware devices to the guest OS directly rather than virtualizing the devices. The hypervisor 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 the live migration. A performance evaluation confirmed that the OS performance with BLMVisor is comparable to that of a bare-metal machine.

**Index Terms**—Virtualization, operating system, live migration, bare-metal cloud

## 1 Introduction

LIVE migration is commonly employed in Infrastructure-as-a-Service (IaaS) clouds. Live migration provides greater management flexibility by allowing cloud vendors to move a running operating system (OS) to another physical machine with negligible downtime [1]. For example, cloud vendors stop physical machines to perform routine maintenance, e.g., to replace failed devices and update firmware [1]. Live migration allows cloud vendors to perform such maintenance without interrupting services. Similar to routine maintenance, to anticipate hardware faults, proactive fault tolerance [2], [3], [4] 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. [5] 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 graphics processing units (GPU), solid-state drives (SSD), and InfiniBand network devices. Recently, the performance of storage devices (e.g., NVM Express [6] and 3D Xpoint SSDs [7]) and network devices (10 GbE and 40 GbE) has improved significantly. In addition, some devices offer rich functions, such as multiple queues and Single Root I/O Virtualization (SR-IOV). 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 [8], [9], [10], [11], new OS architectures [9], and optimal applications for modern devices [12] 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 (VM). 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 [13], [14]. Bare-metal clouds are also favored by security-sensitive users who are concerned about information leakage among VMs in traditional multi-tenant clouds [15], [16]. Therefore, bare-metal clouds have become widely available and are supported by leading cloud vendors, such as IBM [17], Oracle [18], Internap [19], and Rackspace [20].

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 [21], Xen [1], and KVM [22] support live migration. However, bare-metal clouds do not have a virtualization layer, and therefore, machine states exist in physical hardware. Thus, saving and restoring hardware states using a conventional software-based approach is difficult.

Previous studies have shown that live migration can be implemented in the OS layer [23], [24], [25]. However, in bare-metal clouds, live migration schemes should be independent of the OS for the following reasons. First, cloud users should be considered separately from cloud operators. If a live migration scheme depends on an OS, the live migration operations must be performed by the

- *T. Fukai and K. Kato are with the Department of Computer Science, Graduate School of SIE, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan.*
  *E-mail: fukai@osss.cs.tsukuba.ac.jp, kato@cs.tsukuba.ac.jp*
- *T. Shinagawa is with the Information Technology Center, The University of Tokyo, 2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-8658, Japan.*
  *E-mail: shina@ecc.u-tokyo.ac.jp*

*Manuscript received 12 Jul. 2017; revised 23 Feb. 2018.*

user. However, it is impractical to expect users to anticipate unpredictable maintenance and perform live migration as required. Second, the range of supported OSs should be broadened. If a live migration scheme is OS-dependent, cloud users cannot freely select an OS. In addition, even if a live migration scheme supports major OSs, installing and maintaining the additional software for live migration is a daunting task. Furthermore, users may customize OSs to optimize the performance for high-end devices. Such customization may conflict with a migration scheme that only considers major OSs.

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 hardware-assisted 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 [1], [26]. However, the internal states of various physical devices, such as network interface cards (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.

Although this scheme is device specific, it is not overly restrictive. First, it is easier to implement a device state migration scheme than to write a device driver for a physical device. For example, implementation of the migration scheme for a Realtek RTL8169 NIC comprises only 1,176 lines of code (LOC). Most parts of the device states are readable and writable, and only a few device states are completely inaccessible. Such inaccessible states can be handled by the proposed techniques. Second, there is typically less device diversity in IaaS cloud servers than in client machines. Most IaaS vendors use a common set of hardware supported by hypervisor vendors. For example, VMware develops and maintains device drivers for the vSphere ESXi hypervisor. Therefore, maintaining device-specific software for a set of server hardware is a practical approach.

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.

- 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.
- The implementation of a hypervisor based on BitVisor [27] is described. The hypervisor can migrate an unmodified Linux OS running on machines with typical hardware, such as programmable interrupt controllers (PIC), advanced programmable interrupt controllers (APIC), programmable interval timers (PIT), and Realtek RTL8169 NICs.
- The proposed BLMVisor is evaluated compared to a commodity virtual machine monitor (VMM). The results demonstrate that BLMVisor incurs negligible overhead on network throughput and latency, and reduces overhead in Redis and MySQL benchmarks by 16.3% and 33.9%, respectively

Note that a preliminary version of this paper has been published previously [28]. Differing from the preliminary version, the implementation discussed in this paper supports multi-core CPUs. Supporting multi-core CPUs involves several new issues, such as determining which core should handle the NIC dedicated to the hypervisor, how dirty bits in multiple cores should be handled, how the hypervisor obtains control with sufficient frequency, and how to implement synchronization among cores. Note that these issues are related to the uniqueness of the BLMVisor architecture and are addressed and summarized in Section 4.6. In addition, a comprehensive evaluation was performed based on this implementation. Nearly all bare-metal clouds provide multi-core machines; thus, to demonstrate the practicality of BLMVisor, it should be evaluated on multi-core machines.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the BLMVisor design, and Section 4 explains the hypervisor implementation. Section 5 presents the evaluation, and Section 6 discusses potential applicability. Conclusions are presented in Section 7.

## 2 Related Work

Here, work related to the live migration of VMs, OSs, containers, and processes is reviewed.

### 2.1 VM migration

Live migration of VMs is supported by several major VMMs such as VMware's VMotion [21], XenSource's XenMotion [1], and KVM [22]. Live migration enables cloud vendors to manage multiple server machines effectively. Proactive maintenance [1] and proactive fault tolerance [2], [3] are crucial to maintain the health, reliability, and security of physical machines. Live migration of VMs 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 high-performance computing applications [14]. 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 [29], [30], [31]. 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 [29] 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 [30] introduced shadow drivers in the guest kernel that efficiently save and restore the states of device drivers. Pan et al. proposed CompSC [31], a scheme 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 [32] 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 [33] 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 [34] addressed the OS-dependency 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 [23], [24].

Hansen et al. [24] proposed two prototype systems for OS-level migration. 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 paging-via-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. [23] 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 [25]. 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 kernels.

### 2.3 Process migration

Process migration [35], [36], 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 [37] 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.

## 3 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

A 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

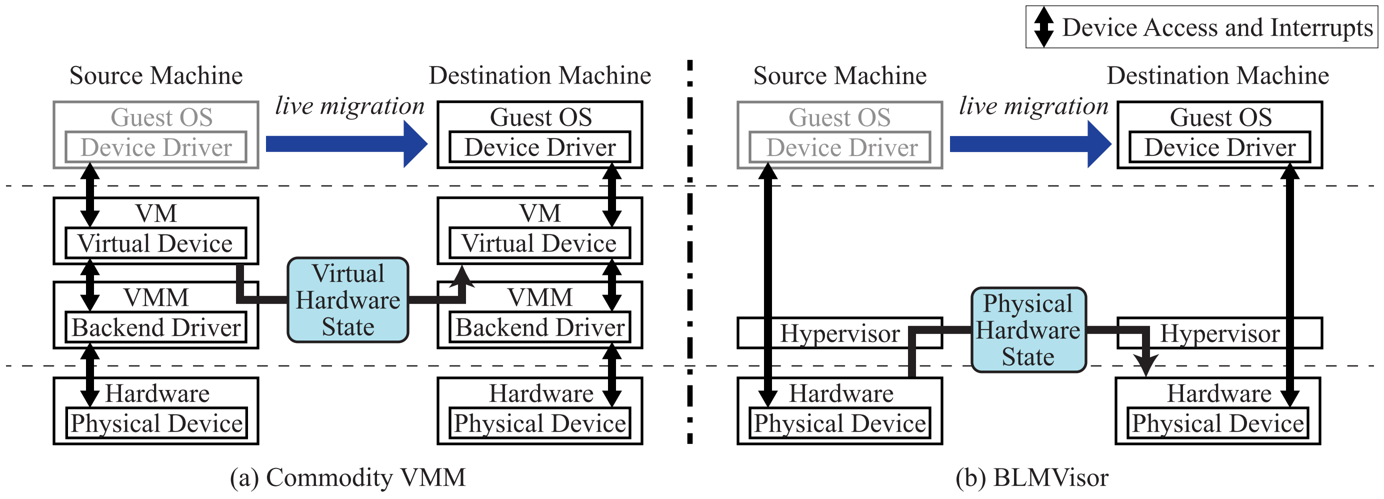

Fig. 1. Comparison of commodity VMM and BLMVisor architectures

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. 1. In the commodity VMM (Fig. 1-(a)), 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 (Fig. 1-(b)), 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.

The second goal, i.e., OS-independency, is achieved because the hypervisor saves and restores the physical hardware states without the support of the guest OS. Prior to live migration, the hypervisor on the source machine saves the physical hardware states with minimal operations. During live migration, the hypervisor transfers the physical hardware states to the hypervisor on the destination machine over a network. The destination hypervisor receives the states and restores them to the destination machine's physical hardware. After live migration, the hypervisor returns to a mostly idle state and waits for the next live migration.

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 (see "Virtual Hardware State" in Fig. 1-(a)). In contrast, in BLMVisor, the hardware states to be saved and restored are those of actual physical devices (see "Physical Hardware State" in Fig. 1-(b)). 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 (Section 5.3.3).

### 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 (Section 4.5).

## 4 IMPLEMENTATION

This section describes the implementation of BLMVisor, which is based on BitVisor [27]. BLMVisor uses the pre-copy method [1], 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 [26] 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 structure (VMCS), 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. General-purpose 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 [1], 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 retransfers 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 [14].

### 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 (PIO) or memory-mapped I/O (MMIO). 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 extended page table (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.

TABLE 1
PIO ports monitored during normal execution

| Device | I/O port | State |
|---|---|---|
| PIC | 0x20 and 0x60 | ICW 1 |
| PIC | 0x21 and 0x61 | ICW 2–4 and other configurations |
| PIT | 0x40 | Timer interval |
| PIT | 0x43 | Mode of the timer |

#### 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 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

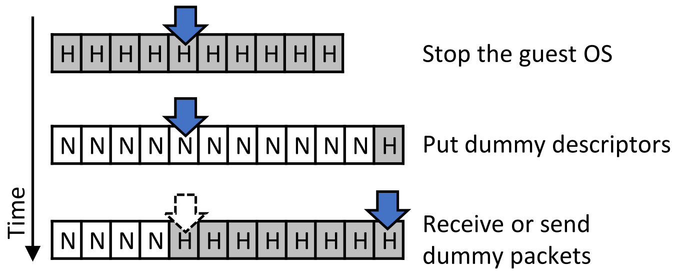

Fig. 2. Obtaining internal pointer in Realtek RTL8169 by sending/receiving dummy packets. "N" and "H" entries belong to the NIC and host, respectively. The original internal pointer points to the fifth entry.

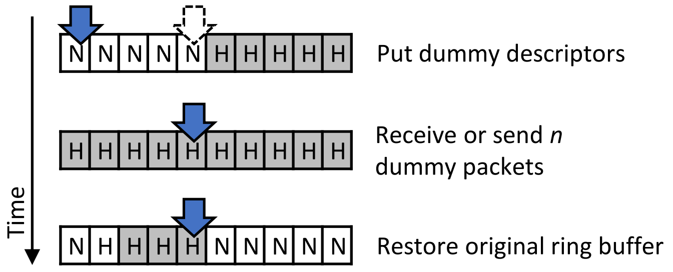

Fig. 3. Setting an internal pointer in the Realtek RTL8169 by sending/receiving dummy packets. "N" and "H" entries belong to the NIC and the host, respectively.

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.

One way to obtain internal pointer values is to constantly monitor communication between the OS and devices; however, this is expensive and should be avoided. Instead, the hypervisor actively controls the devices and inspects their behaviors in the stop-and-copy phase. This is a destructive inspection, i.e., internal states are destroyed by the hypervisor's operations. However, this is not a problem because the guest OS on the source machine no longer needs to run after the stop-and-copy phase. Just before this destructive operation, the hypervisor saves the descriptor and buffer data in memory; therefore, nearly all received and sending packets transferred by DMA are saved properly. Although some packets may be lost due to the timing problem, Ethernet is originally an unreliable network, and, in any case, there is a certain downtime in the stop-and-copy phase. Fortunately, lost packets will be retransmitted by the upper-layer protocol.

With the RTL8169, 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 (Fig. 2). 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 of Fig. 2, 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 (bottom of Fig. 2). 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 sends and receives the required number of dummy packets.

Fig. 3 shows this operation. Here, 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. As shown in the top of Fig. 3, 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 (middle of Fig. 3). 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 (bottom of Fig. 3). Finally, the hypervisors on the source and destination machines exchange the MAC addresses of their respective NICs.

Note that care must be taken relative to the order in which device states are reconstructed because reconstructing configuration states typically affects the processing states. For example, reconstructing the configuration states of a NIC may reset processing states. In contrast, reconstructing the processing states does not affect configuration states. Therefore, the hypervisor first reconstructs the configuration states and then reconstructs the processing states. The same sequence is employed for device initialization.

### 4.6 Supporting multi-core system

In our previous work [28], the implementation only supported single-core systems; however, nearly all server machines in bare-metal clouds are multi-core systems. Therefore, to evaluate BLMVisor in practical environments, it is essential to support multi-core systems.

There are several issues related to supporting multi-core systems. The first is which core should perform memory transfer in the pre-copy phase. As described in Section 4.2, 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, 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 end-of-interrupt (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.

However, fewer VM exits pose a problem. Since the hypervisor cannot obtain control frequently, it does not have sufficient opportunity to record dirty pages and transfer memory pages in the pre-copy phase. In commodity VMMs, the VMM manages the timer devices and timer interrupts occur periodically, thereby allowing the VMM to obtain control at sufficient frequency. However, in BLMVisor, timer devices are completely handled by the guest OS and their interrupts are delivered directly to the guest OS without hypervisor intervention. To handle this situation, the hypervisor uses the VT-x preemption timer, which forcibly generates VM exits at a preset frequency. In the current implementation, the VM exit frequency for the core that performs memory copy is adjusted such that the memory data can be transferred at the line rate of the NIC (approximately once every $786\mu s$), and the VM exit frequency of the other cores is once per second, which is sufficient to record dirty pages. In addition, the CPU is configured to not enter the C2 or deeper C state, because the preemption timer stops at these states. This issue is caused by the architecture that significantly reduce virtualization overhead and changes in device characteristics associated with supporting multi-core. Note that the vCPU settings in the VMCS in the pre-copy phase are the same as in the normal phase, except for this preemption timer setting.

The last issue is how to synchronize among cores in the stop-and-copy phase. When the stop-and-copy phase begins, the hypervisor must gain control of all cores via VM exits. However, if the hypervisor does not take any action, VM exits will occur only once per second and delays will occur among the cores, which will extend downtime. To avoid such delays without incurring additional overhead, the hypervisor configures the CPU to cause VM exits on non-maskable interrupts (NMI), and a pre-defined core sends an inter-processor NMI to other cores when the stop-and-copy phase begins, thereby immediately causing VM exits on all cores. Note that the hypervisor still does not intercept maskable interrupts to reduce virtualization overhead.

### 4.7 Implementation status

As a proof of concept, the current implementation supports a multi-core CPU, interrupt controllers (PIC and APIC), a timer device (PIT), and a Realtek RTL8169 NIC. A brief investigation of device specifications suggests that other devices, such as storage devices and high-end NICs, can also be supported by BLMVisor. The migration code for the Realtek RTL8169 requires only 1,176 LOC, including code to capture unreadable states and reconstruct unwritable states, as described above. This is much smaller than normal device drivers in general OSs. For example, the device driver for the Realtek RTL8169 in Linux version 4.9 (`drivers/net/ethernet/realtek/r8169.c`) contains more than 6,800 LOC. Therefore, it is relatively easy to implement such device-dependent code in BLMVisor.

## 5 Evaluation

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

### 5.1 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

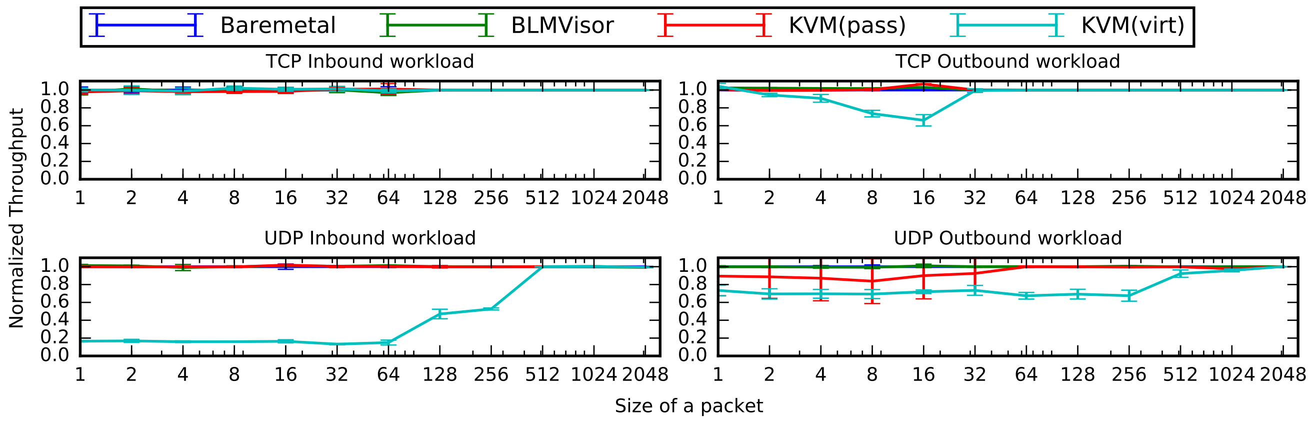

Fig. 4. Network throughput of TCP/UDP inbound/outbound workload with 1 to 2048-byte packets (normalized to bare-metal)

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. iSCSI storage was used for the root file system.

The evaluation was performed using a bare-metal machine ("Baremetal"), BLMVisor, and KVM configured to assign the same number of virtual cores (vCPUs) 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 pass-through 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 were used as storage devices by the guest OSs through the iSCSI server. 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.

## 5.2 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.

### 5.2.1 *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.

Fig. 4 shows network throughput. 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 packet in the Linux TCP stack using Nagle's algorithm [38]. 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.

The UDP throughput with small packets also differed among the systems. Specifically, the UDP inbound throughput with "KVM (virt)" with 1 to 64-byte packets fell to only 17% or less of the throughput of "Baremetal," and overhead in the UDP outbound throughput with small packets was approximately 40%. These overheads were caused by virtualization. Even "KVM (pass)" incurred 10% overhead in the UDP outbound throughput with small packets due to interrupt handling. In contrast, the UDP inbound throughput of "BLMVisor" and "KVM (pass)" was similar to that of "Baremetal." In the UDP outbound workload with packets smaller than 32 bytes, the standard deviation of the throughput of "KVM (pass)" was large because the iSCSI heartbeat messages could not reach the server due to the presence of many small packets and the interrupt handling overhead. As a result, the iSCSI daemon reset the NIC's link, thereby disconnecting Netperf's TCP connections.

Note that some packets were dropped in the UDP outbound workload of "KVM (virt)." because the virtio-net interface and tap device did not negotiate; therefore, there was no link speed limitation. In addition, packet transmission between these interfaces only involved data copy in memory. As a result, the throughput between these

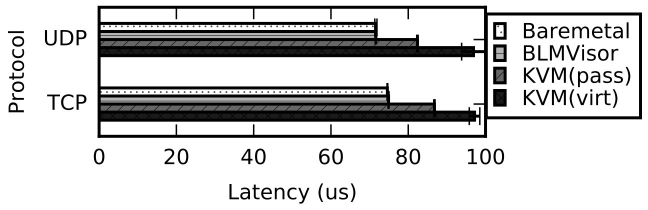

Fig. 5. TCP and UDP network latency

TABLE 2
VM exits per second

| Exit Reason | Proposal | KVM (pass) | KVM (virt) |
|---|---|---|---|
| PAUSE | - | 1613955.1 | 1594912.9 |
| APIC access | - | 62505.3 | 34355.9 |
| IO instruction | - | - | 9860.3 |
| External interrupt | - | 16933.9 | 7437.4 |
| Interrupt window | - | 5889.1 | 3.7 |
| EPT violation | 17.4 | 126.4 | - |
| Exception or NMI | 8.1 | 4.7 | 4.0 |
| Control-register accesses | - | 0.2 | 0.2 |
| Total | 25.5 | 1699459.1 | 1646606.0 |
| Total (excluding PAUSE) | 25.5 | 57502.7 | 51693.1 |

interfaces was much higher than that of the physical NIC. Therefore, some packets were dropped between the tap and physical devices. Note that the throughput shown was calculated by excluding such dropped packets.

Fig. 5 shows the TCP and UDP network latencies. In both TCP and UDP workloads, the overhead was close to zero for "BLMVisor" (TCP: 0.4%; UDP: -0.04%), approximately 15% for "KVM (pass)" (TCP: 16.3%; UDP: 15.0%), and greater than 30% for "KVM (virt)" (TCP: 30.2%; UDP: 35.7%). The KVM overhead with the virtio device was caused by device virtualization. Here, packets were handled by the device drivers of the guest and host OSs and were forwarded via the virtio interface. These processes incurred significant overhead on network performance. Although the overhead was mitigated by PCI pass-through, it was not eliminated entirely. The remaining overhead was likely due to interrupt interposition by KVM.

### 5.2.2 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.

TABLE 2 shows the number of VM exits for "BLMVisor," "KVM (pass)," and "KVM (virt)." Most VM exits occurred in KVM due to "PAUSE", which is executed by the guest OS when it is idle; therefore, this event is not critical for performance. The other VM exits that occurred in KVM were related to interrupt handling, such as "APIC access," "External interrupt," and "Interrupt window." In "KVM (virt)," VM exits for "IO instruction" occurred because the guest OS handled I/O requests to virtio devices using PIO access. These results show that, with KVM, I/O emulation and interrupt handling were the primary cause of overhead in latency. In contrast, the number of VM exits that occurred with BLMVisor was less than 26 per second. This confirms that BLMVisor can achieve performance that is comparable to that of a bare-metal system.

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 $\mu$s per core). This is considered negligible overhead.

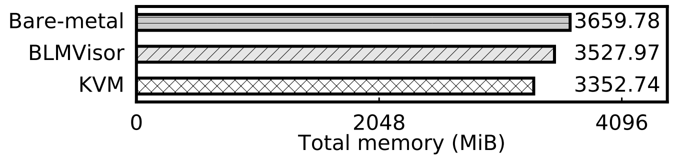

Fig. 6. Total memory available to guest OS

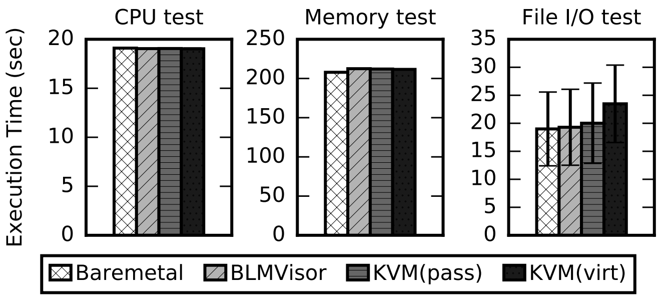

Fig. 7. Execution time of Sysbench tests.

### 5.2.3 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. Fig. 6 shows the amount of memory measured by the `free` command. 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. Thus, BLMVisor's simple architecture contributes to consuming less memory consumption, which will increase the performance of memory-intensive applications.

### 5.2.4 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

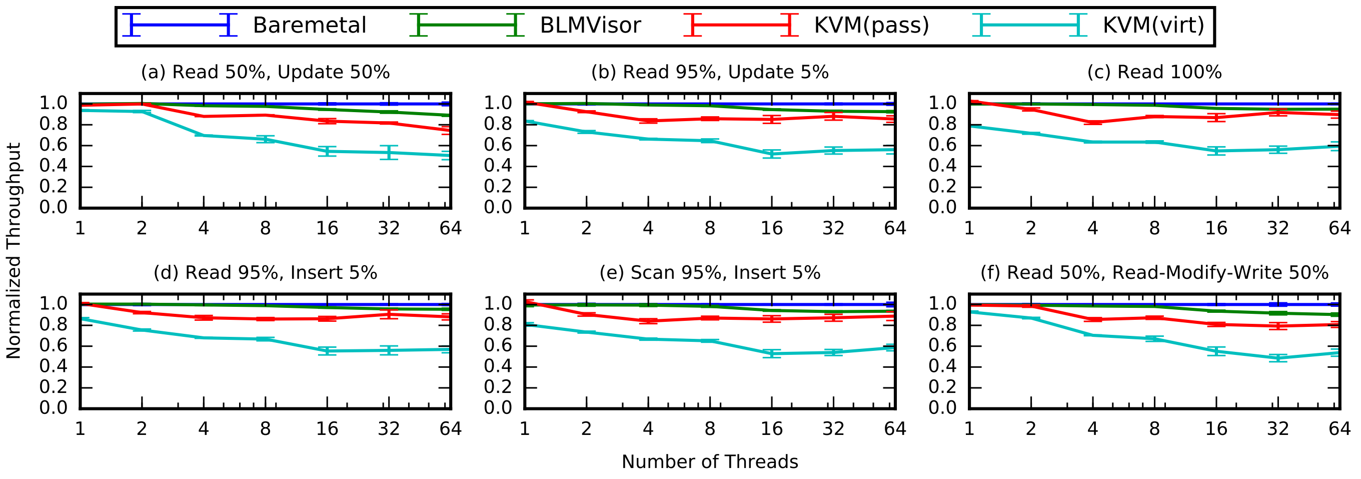

Fig. 8. Redis throughput with various workloads. Horizontal and vertical axes denote the number of client threads and the throughput in number of operations per second (normalized by the bare-metal throughput), respectively.

measured 10 times for each configuration and plotted as the average and standard deviation of the execution time.

The left graph in Fig. 7 shows the execution times of the Sysbench CPU test, which calculated prime numbers. The execution times were similar for all systems because there were few VM exits in the CPU-intensive workloads.

The middle graph in Fig. 7 shows the execution times of the Sysbench memory test, which randomly wrote memory in 16-MiB space. The total memory written was 100 GiB. Here, "BLMVisor," "KVM (pass)," and "KVM (virt)" showed 2.21%, 2.01%, and 1.78% increased execution time, respectively. 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. This issue is discussed in Section 6.3.

The right graph in Fig. 7 shows the execution times of the Sysbench file I/O test, which performed random access to a random file. Here, the number of files was 128 (16 MiB each). The access size was 16 KiB and the number of accesses was 200,000, yielding a total access size of approximately 3 GiB. The read-write ratio was 3:2 and the `fsync()` function was called every 100 accesses. The execution time of "BLMVisor" increased by only 1.6%, whereas that of "KVM (pass)" and "KVM (virt)" increased by 5.4% and 23.6%, respectively. Note that the overhead differences may have been caused by network latencies. Since the iSCSI protocol requires small command and response packets, network latency affects the performance of iSCSI storage.

### *5.2.5 Database benchmark*

To evaluate the performance of real server applications, the throughput of Redis [39] (version 3.0.0) with a YCSB benchmark client [40] and the execution time of an SQL workload on MySQL [41] with a Sysbench OLTP test client [42] were measured.

Fig. 8 shows the Redis throughputs. The workloads were as follows: (a) update heavy workload (read ops:upload ops = 50:50), (b) mostly read workload (read ops:upload ops = 95:5), (c) read only workload (read ops:upload ops = 100:0), (d) read latest workload (read ops:insert ops = 95:5), (e) short ranges workload (scan ops:insert ops = 95:5), and (f) read-modify-write workload (read ops:read-modify-write ops = 50:50). Except for workload (d), the client read or inserted records selected from a Zipfian distribution. In workload (d), the latest updated record was mostly read.

In this experiment, Redis was configured to place no data in storage, i.e., used as a cache server. There were 1,000,000 records in the test database and the client ran for 10 seconds each time. The throughput was measured for 1-64 client threads. The graphs show the average and standard deviation of 20 measurements for each configuration. Here, a higher throughput denotes a better result.

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 4.6–11.0% overhead, "KVM (pass)" incurred a 10.3–25.5% overhead, and "KVM (virt)" incurred a 41.0–49.5% overhead. The worst case was (a) update heavy workload 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 [43]. The overhead in KVM was caused by I/O emulation and interrupt handling in network packet processing.

Fig. 9 shows the execution time of the Sysbench OLTP test with MySQL servers. Note that these measurements were performed on Windows and Linux. MySQL versions 5.5.52 and 5.7.21 were used for Linux and Windows, respectively. The test table contained 10,000 rows, and the number of transactions was 10,000. Each transaction consisted of 14 read queries (`SELECT`) and four write queries (`INSERT`, `UPDATE`, `DELETE`). The execution time was measured for 1–32 client threads. The graph plots the average and standard deviations of 10 measurements for each configuration. The horizontal axis is the number of client threads and the

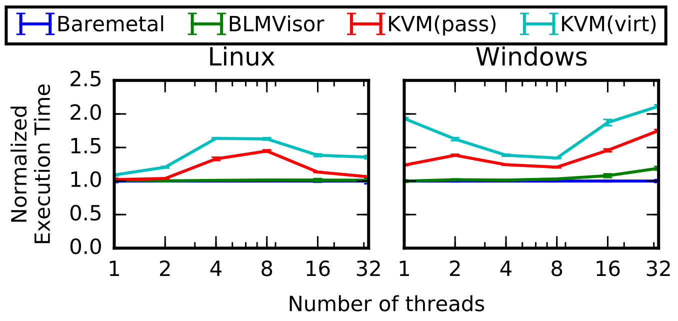

Fig. 9. Execution time of Sysbench OLTP test with MySQL server on Linux and Windows. The horizontal and vertical axes represent the number of client threads and execution times (normalized by that of bare-metal) of the benchmark, respectively.

vertical axis is the execution time of the benchmark. Here, lower execution time indicates a better result.

In Linux, "BLMVisor" increased execution time by 1.6% in the worst case (eight threads). In contrast, "KVM (pass)" increased execution time by 2.1–44.7%, and "KVM (virt)" increased execution time by 9.0% in the best case (one thread) and 63.6% in the worst case (four threads). In Windows, "BLMVisor" increased execution time by less than 8% with 16 threads or fewer, and increased execution time by 19.0% with 32 threads. In contrast, "KVM (pass)" increased execution time by 20.7–75.2%, and "KVM (virt)" increased execution time by 34.3–111.5% (the best case was eight threads 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.

## 5.3 Performance during live migration

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

### 5.3.1 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. As described in Section 4.2, the current implementation fixes the memory transfer speed at approximately 1 Gbps. Therefore, this measurement includes the maximum overhead of the pre-copy phase.

Fig. 10 and Fig. 11 show the results for Linux and Windows, respectively. The Netperf client measured the throughput at 100-ms intervals. Live migration began three seconds after initiating the Netperf benchmark. The pre-copy 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.

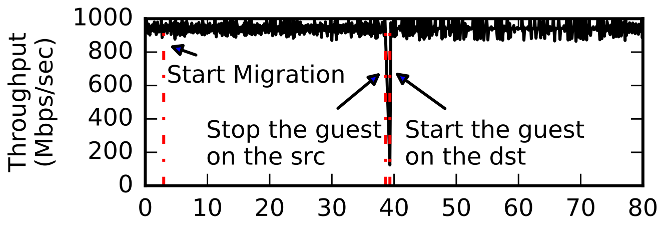

Fig. 10. Network throughput of Linux during live migration

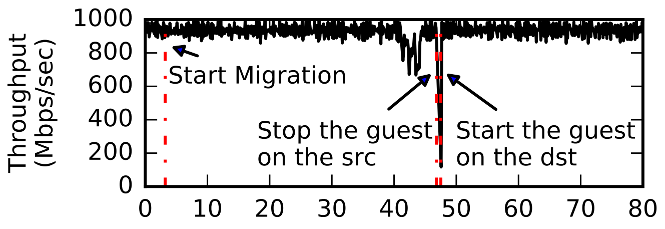

Fig. 11. Network throughput of Windows during live migration

### 5.3.2 Number of VM exits

To analyze the performance impact of live migration, the number of VM exits during the Linux live migration was measured. Here, the workload was the same as that in Section 5.2.2. The number of VM exits was essentially the same as that under normal execution conditions, except for those of the preemption timer. On average, the preemption timer caused 837.5 VM exits per second, and the hypervisor consumed approximately 320 ms per second ($\approx$ 1,276 M cycles / 4.0 GHz) to handle the VM exits. Note that most VM exits occurred in the fixed core to transfer memory data.

The Sysbench CPU test was used during live migration to clarify the impact of this overhead on CPU performance. Here, single-thread execution times were measured for a CPU core during normal execution, the pre-copying core during live migration, and the other core during live migration. The execution times were 8.16, 11.93 (46.2 % increase), and 8.17 (0.1 % increase) seconds, respectively. These results demonstrate that, as expected, the performance impact of the pre-copy core was moderately large, while the performance impact on the other cores was negligible. The additional operation of the hypervisor in the other cores was recording dirty pages. Therefore, this result suggests that recording dirty pages on each core is a reasonable design.

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 % increase) 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.

### 5.3.3 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.

## 6 Discussion

This section discusses the potential applicability and future development of BLMVisor.

### 6.1 Checkpointing

Checkpointing VMs is useful for management and fault tolerance [44]. Note that live migration and checkpointing mechanisms are similar. They differ only in whether they send machine states to a file or a network and whether they restore machine states on the same machine or a different machine. Therefore, with only slight modification, BLMVisor can be adapted to a checkpointing system.

### 6.2 Device support

Recall that BLMVisor is device dependent. To support new devices, such as 10-GbE NICs, NVMe devices, InfiniBand devices, and GPUs, a new migration module to save and restore device states must be developed. Developers of such migration modules must at least partially understand device specifications. However, developers only need to enumerate the registers of the essential states to be saved and restored. This is much easier than writing device drivers, which requires understanding the logic and control flow of the given device. The migration modules for devices in the prototype system require far fewer LOC than the device drivers. In future, automation or assistant technologies for developing device drivers will help develop migration modules.

### 6.3 Dynamically starting and stopping the hypervisor

Before and after live migration, BLMVisor has nothing to do; however, it still incurs some overhead for some VM exits and paging events (i.e., the EPT). If virtualization can be turned off and on as required, overhead in normal execution becomes zero and performance equal to that of bare-metal machines can be achieved. This dynamic starting and stopping of the hypervisor involves two issues. The first issue is protection, i.e., if the hypervisor resides in memory, it is not protected when the EPT is turned off. This will not be problematic if the guest OS can be trusted; however, this may be mitigated by technology that can verify memory content. The second issue is how to start the hypervisor. Since virtualization is turned off, there is no event to pass control to the hypervisor. This could be mitigated by installing some agent program in the guest OS.

## 7 Conclusions

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/APIC, PIT, 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.

## Acknowledgments

This work was supported by JSPS KAKENHI Grant Number JP16H02798. The authors would like to thank Enago (www.enago.jp) for the English language review. The authors would also like to thank Dr. Yushi Omote for his technical advice, and the reviewers for useful comments.

## 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] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive Fault Tolerance for HPC with Xen Virtualization," in *Proceedings of the 21st Annual International Conference on Supercomputing*, Jun. 2007, pp. 23–32.
[3] A. Polze, P. Troger, and F. Salfner, "Timely Virtual Machine Migration for Pro-active Fault Tolerance," in *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, Mar. 2011, pp. 234–243.
[4] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive Fault Tolerance Using Preemptive Migration," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb. 2009, pp. 252–257.
[5] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint VM Placement and Routing for Data Center Traffic Engineering," in *Proceedings of the the 31st Annual IEEE International Conference on Computer Communications (IEEE INFOCOM 2012)*, Mar. 2012, pp. 2876–2880.
[6] *NVM Express Revision 1.2.1*, 2016. [Online]. Available: http://www.nvmexpress.org/specifications/
[7] "3D XPoint$^{TM}$: 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] "DPDK." [Online]. Available: http://dpdk.org/
[9] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Jun. 2012, pp. 101–112.
[10] "Storage Performance Development Kit." [Online]. Available: http://www.spdk.io/

[11] "Seastar." [Online]. Available: http://www.seastar-project.org
[12] "ScyllaDB." [Online]. Available: http://www.scylladb.com/
[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.
[14] Y. Omote, T. Shinagawa, and K. Kato, "Improving Agility and Elasticity in Bare-metal Cloud," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, 2015, pp. 145–159.
[15] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, 2009, pp. 199–212.
[16] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 2011)*, 2011, pp. 203–216.
[17] "SoftLayer." [Online]. Available: http://www.softlayer.com
[18] "Bare Metal Cloud Services — Oracle Cloud." [Online]. Available: https://cloud.oracle.com/en_US/bare-metal
[19] "Internap." [Online]. Available: http://www.internap.com
[20] "Rackspace." [Online]. Available: http://www.rackspace.com/
[21] 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.
[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, pp. 225–230.
[23] 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 2009)*, May 2009.
[24] J. G. Hansen and E. Jul, "Self-migration of Operating Systems," in *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.
[25] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, "Containers checkpointing and live migration," in *Proceedings of the Linux Symposium*, 2008, pp. 85–92.
[26] 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.
[27] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: A Thin Hypervisor for Enforcing I/O Device Security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2009)*, Mar. 2009, pp. 121–130.
[28] T. Fukai, Y. Omote, T. Shinagawa, and K. Kato, "OS-Independent Live Migration Scheme for Bare-Metal Clouds," in *Proceedings of the 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC 2015)*, Dec 2015, pp. 80–89.
[29] 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.
[30] A. Kadav and M. M. Swift, "Live Migration of Direct-access Devices," *SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 95–104, Jul. 2009.
[31] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang, "CompSC: Live migration with pass-through devices," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2012)*, Mar. 2012, pp. 109–120.
[32] Y. Dong, Y. Chen, Z. Pan, J. Dai, and Y. Jiang, "ReNIC: Architectural extension to SR-IOV I/O virtualization for efficient replication," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 40:1–40:22, Jan. 2012.
[33] K. Dey, D. Mishra, and P. Kulkarni, "Vagabond: Dynamic network endpoint reconfiguration in virtualized environments," in *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC '14)*, 2014, pp. 21:1–21:13.
[34] X. Xu and B. Davda, "SRVM: Hypervisor support for live migration with passthrough SR-IOV network devices," in *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*, 2016, pp. 65–77.
[35] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software: Practice and Experience*, vol. 21, no. 8, pp. 757–785, 1991.
[36] A. Barak and O. La'adan, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing," *Journal of Future Generation Computer Systems*, vol. 13, no. 45, pp. 361 – 372, 1998.
[37] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002, pp. 361–376.
[38] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, Internet Engineering Task Force, Jan. 1984. [Online]. Available: https://www.rfc-editor.org/info/rfc896
[39] "Redis." [Online]. Available: http://redis.io/
[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 143–154.
[41] "MySQL." [Online]. Available: https://www.mysql.com/
[42] "Sysbench," https://launchpad.net/sysbench.
[43] 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.
[44] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013, pp. 3:1–3:16.

**Takaaki Fukai** received a bachelor's degree from the Department of Information Technology, Okayama University, Japan, in 2013, and a master's degree from the Department of Computer Science, University of Tsukuba, Japan, in 2015. He is currently working toward a PhD degree in computer science at the University of Tsukuba. His research interests include operating systems, virtualization, and cloud computing. He is a student member of the IEEE and the IEEE Computer Society.

**Takahiro Shinagawa** received B.E. and M.S. degrees from the University of Tokyo in 1998 and 2000, respectively. He received a Ph.D. degree in Science from the University of Tokyo in 2003. He was an Assistant Professor at Tokyo University of Agriculture and Technology, and a lecturer at the University of Tsukuba. Since 2011, he has been an Associate Professor at the University of Tokyo. His research interests include operating systems, virtualization, and secure computing. He is a member of ACM, IEEE, and USENIX.

**Kazuhiko Kato** received BE and ME degrees from the University of Tsukuba, Japan, in 1985 and 1987, respectively. He received a PhD degree from the University of Tokyo, Japan, in 1992. From 1989 to 1993, he was a research associate in the Department of Information Sciences, Faculty of Sciences, University of Tokyo. He was an Assistant Professor and an Associate Professor in Institute of Information Sciences and Electronics, University of Tsukuba. Since 2004 he has been a Professor in the Department of Computer Science, Graduate School of System Information Engineering, University of Tsukuba. His research interests include operating systems, distributed systems, and secure computing. He received distinguished paper awards from JSSST and IPSJ in 2004, 2005 and 2013, and from the IEEE/ACM UCC Conference 2015.