On-demand Virtualization for Live Migration in Bare Metal Cloud

Jaeseong Im
School of Computing, KAIST
Daejeon, Republic of Korea
ildelusion@kaist.ac.kr

Jongyul Kim
School of Computing, KAIST
Daejeon, Republic of Korea
jongyul@calab.kaist.ac.kr

Jonguk Kim
School of Computing, KAIST
Daejeon, Republic of Korea
jrcforever@kaist.ac.kr

Seongwook Jin
KIWIPLUS
Seoul, Republic of Korea
dexter.jin@kiwiplus.io

Seungryoul Maeng
School of Computing, KAIST
Daejeon, Republic of Korea
maeng@kaist.ac.kr

ABSTRACT

The level of demand for bare-metal cloud services has increased rapidly because such services are cost-effective for several types of workloads, and some cloud clients prefer a single-tenant environment due to the lower security vulnerability of such enviornments. However, as the bare-metal cloud does not utilize a virtualization layer, it cannot use live migration. Thus, there is a lack of manageability with the bare-metal cloud. Live migration support can improve the manageability of bare-metal cloud services significantly.

This paper suggests an on-demand virtualization technique to improve the manageability of bare-metal cloud services. A thin virtualization layer is inserted into the bare-metal cloud when live migration is requested. After the completion of the live migration process, the thin virtualization layer is removed from the host. We modified BitVisor [19] to implement on-demand virtualization and live migration on the x86 architecture.

The elapsed time of on-demand virtualization was negligible. It takes about 20 ms to insert the virtualization layer and 30 ms to remove the one. After removing the virtualization layer, the host machine works with bare-metal performance.

CCS CONCEPTS

• **D.4.0 Software → Operating Systems**;

GENERAL TERMS

Design, Performance, Measurement

KEYWORDS

On-demand Virtualization, Bare-metal Cloud, Live Migration

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09. . . $15.00
https://doi.org/10.1145/3127479.3129254

ACM Reference Format:
Jaeseong Im, Jongyul Kim, Jonguk Kim, Seongwook Jin, and Seungryoul Maeng. 2017. On-demand Virtualization for Live Migration in Bare Metal Cloud. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 12 pages.
https://doi.org/10.1145/3127479.3129254

1 INTRODUCTION

The demand for bare-metal cloud services has increased given the current preference for services with high levels of both performance and security. In addition, certain workloads such as those associated with high-performance computing (HPC) applications, databases, and distributed workloads require predictable performance capabilities of bare-metal cloud services. These requirements are not easy to guarantee in the virtualization-based cloud due to virtualization overhead and the multi-tenancy issue. Performance degradation when using virtualization-based cloud services increases the operating cost [7], while the multi-tenancy issue obviously introduces security vulnerabilities [15].

On the other hand, manageability is one of the most valuable features in the virtualization-based cloud. It allows cloud providers to manage the cloud environment conveniently and efficiently using the characteristics of virtualization. Among them, live migration is a distinct feature of virtualization. It is widely used during node scheduling [4] and load balancing placement operations [10].

However, the bare-metal cloud does not support such an important feature, live migration. Ultimately, manageability is poor compared to that of virtualization-based cloud services. If live migration is used in conjunction with bare-metal cloud services, numerous advantages can be realized in the bare-metal cloud. First, cloud providers can support a scale-up mechanism which migrates bare-metal instances from an underpowered node to a more powerful one (or vice versa for a scale-down operation to reduce power usage and running costs). Secondly, busily communicating machines can be located in close proximity via live migration [13]. Moreover, machine maintenance can be done without terminating bare-metal instances by migrating one machine to another.

Previous work has investigated the migration of bare-metal machines. Similar to virtual machine migration, a process

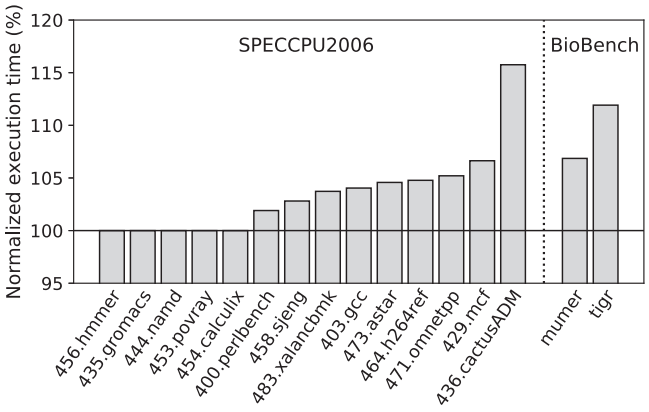

Figure 1: BitVisor overhead in CPU-intensive and Memory-intensive workload

migration technique [18] can also migrate computations. However, this technique is associated with the problem of residual dependency. A migrated process must rely on local resources such as shared libraries and open file descriptors of the source machine (the original location of the process). Accordingly, the source machine must remain powered up, and the performance of the migrated process will decrease due to the cost of communicating with the source machine. Migration without virtualization [13] and Bootstrapped migration [5] utilize OS migration with hibernation. They designed a migration-enabled OS which requires OS modifications. Kooburat and Swift [12] proposed on-demand virtualization to realize both high performance and manageability of virtualization, but it takes a few minutes to migrate the OS owing to hibernation and rebooting. OS-independent live migration for bare-metal cloud [8] migrated a virtual machine using a lightweight hypervisor (BitVisor [19], as used in this paper). As this hypervisor uses a pass-through device model, it monitors OS access to device states to determine unreadable states.

In the paper about live migration with BitVisor [8], the system continuously experiences virtualization overhead. The authors measured the throughput with the Redis [1] database benchmark in BitVisor, with the result showing 6.0% overhead. We also assessed the slowdown of BitVisor compared to a bare-metal machine in SPEC CPU2006 and BioBench [3]. The experiment was performed on an Intel i7-4770 processor with 32GB of DDR3 memory system. Figure 1 shows the execution time of each benchmark normalized to the bare-metal machine. The overhead of the CPU-intensive workload (456.hmmer to 464.h264ref) was only 1.17% on average. However, the average overhead of memory-intensive workloads (483.xalancbmk to 436.cactusADM) was 6.94%. The memory-intensive workloads of BioBench also show low performance on BitVisor. Mummer and tigr showed overhead rates of 6.86% and 11.92% respectively. The overhead mostly occurred due to many TLB misses. Memory-intensive workloads can cause many TLB misses, and these result lots of page table walks using extended page table (EPT).

In this work, we used an on-demand virtualization technique to improve the manageability of the bare-metal cloud. The on-demand virtualization technique inserts a virtualization layer (revirtualization) to leverage manageability and removes the virtualization layer (devirtualization) after using management features to secure bare-metal performance on the fly. We did not use hibernation or rebooting when implementing on-demand virtualization for fast transitions between the virtual machine and the native OS.

We show two implementations of on-demand virtualization. First, software-only on-demand virtualization is implemented using a loadable kernel module (LKM). The LKM triggers revirtualization and devirtualization by means of certain tricks. As this method uses a LKM, a kernel root user must be cooperative during the on-demand virtualization process. On the other hand, we implemented hardware-assisted on-demand virtualization, which does not require a LKM, using the hardware features of the x86 architecture. The latter method does not demand cooperation with a tenant.

This work makes the following three main contributions.

- The on-demand virtualization technique was proposed and implemented on the x86 architecture (both Intel and AMD). The details are given in Section 3.
- A technique which decouples device drivers from hardware devices was proposed to migrate devices states. The on-demand virtualization technique virtualizes not only CPU and memory states but also device states.
- The live migration of a bare-metal machine instance exploiting on-demand virtualization was suggested and implemented as a prototype.

The live migration of the bare-metal machine instance succeeded with on-demand virtualization. The network connectivity is maintained after the migration is complete, and there is no virtualization overhead after the virtualization layer is removed. The entire live-migration process except for the time required for memory copying is done within 30 ms.

The rest of this paper is organized as follows: We present related work in Section 2. Sections 3 and 4 describe the main idea, design, and implementation of the work. Section 5 provides the experimental results for live migration. Section 6 discusses future work and concludes the paper.

2 RELATED WORK

Live migration of a bare-metal machine. Several mechanisms have been suggested for the live migration of a bare-metal machine. Unfortunately, most of them end in merely conceptual suggestions without implementation. Kozuch et al. [13] discussed the possibility of introducing a new module for live migration into an OS kernel. Apart from the absence of practical implementation, as the kernel source code requires considerable modification, the solution cannot be applied

to current systems easily. As an instance, the suggestion required a new version of *kmalloc()* which is spread throughout the kernel source code.

Hibernation on Linux was exploited by Chiang et al. [5] for migration. Given that all user and kernel processes are suspended during hibernation, the system enters the proper state to start the migration process. Additionally, privileged software such as a hypervisor is not required. However, a large memory capacity may be required to hold the entire hibernated image of the source machine. If the system has an insufficient memory capacity, the image should be written to disk, leading to considerable downtime during the migration process.

Fukai et al. [8] proposed and implemented a live-migration scheme for a bare-metal machine. It utilized a thin virtualization layer which does not virtualize hardware devices, exposing them to the guest OS directly. The entire migration process is managed by the virtualization layer. In fact, the virtualization layer remains before and after the migration process. As a result, computation is degraded, as shown in Figure 1.

On-demand virtualization. On-demand virtualization was initially proposed by Lowell et al. [14]. They activated the virtualization layer for online maintenance of the OS. The paper exploits the Privileged Architecture Library code (*PALcode*) of the Alpha architecture. Because *PALcode* is executed at a higher privilege level relative to that of ths OS, revirtualization can be implemented readily without additional software techniques. Therefore, the virtualization of physical memory and of I/O devices, which is essential in live-migration schemes, was omitted.

Cho et al. [6] implemented the on-demand activation of a hypervisor in the ARM architecture to provide a trusted execution environment to applications. This work eliminates performance degradation from the hypervisor during the normal execution process. ARM TrustZone, which supports execution state at higher privilege levels, is utilized to perform the revirtualization step.

In the x86 architecture, Omote et al. [17] proposed the use of devirtualization for the rapid deployment of a bare-metal OS image to a cloud server. The concept of devirtualization was illustrated only theoretically, however, and there was no corresponding implementation.

Kooburat and Swift [12] took advantage of hibernation to implement on-demand virtualization. If an OS needs to be virtualized, it is hibernated and saved to storage. A hypervisor is booted from the same machine and loads the hibernated guest OS image as its virtual machine. All memory content should then be written back to storage during the devirtualization and revirtualization process. It takes a few minutes to migrate the OS due to hibernation and rebooting. In addition, the kernel source code should be modified to support the mechanism.

The dynamic insertion and deletion of a virtualization layer for the CPU and memory was implemented as a kernel module by Nomoto et al. [16].A new virtualization layer is created when the kernel module is loaded and destroyed at the point he kernel module is unloaded. Device virtualization was not considered because the work focuses on enhancing the security of the OS. It is important to note that all the previous works cited above depend on the OS.

3 PROPOSED DESIGN

To migrate a bare-metal machine, we designed and implemented on-demand virtualization in the x86 architecture. We can activate and deactivate a virtualization layer using the on-demand virtualization technique. After activating the virtualization layer, the hypervisor migrates the guest OS on a source host to a target host. This live-migration scheme is identical to virtual machine live migration in a virtualization environment.

Assumptions. Our on-demand virtualization platform trusts the tenants of the bare-metal cloud. After deactivating the hypervisor, the guest kernel can modify the hypervisor memory area, including the hypervisor code. If the guest kernel is compromised or if the tenant is originally malicious, the hypervisor and the on-demand virtualization platform can be attacked when hypervisor layer is removed. To emphasize, the problem is unrelated to the multi-tenancy issue.

Our design only considers traditional I/O devices such as network interface cards and storage devices. To maintain network connectivity during the migration step, a Linux bonding driver is applied to the platform (as discussed in section 3.3). Further research is required for other devices.

Our prototype assumes that the source machine and target machine have identical hardware, including the CPU, motherboard, memory (of the same capacity), and NICs.

Next, we describe the assumptions pertaining to the system configuration. The on-demand virtualization platform assumes network-attached storage which uses a network file system (NFS) so as not to migrate a significant amount of storage. In addition, our prototype needs three network interface cards: the native Linux network driver controls the first one, the paravirtualization driver controls the second one (only used during live migration), and last one is for the hypervisor to migrate the system states. Regarding the kernel configuration, the DEVMEM configuration must be set to use a system management mode (SMM, used in hardware-assisted on-demand virtualization as explained later).

3.1 On-demand Virtualization in the x86 architecture

In contrast to the x86 architecture, the Alpha architecture is essentially virtualizable [9]. OSes on the Alpha architecture have relatively lower privilege levels than on the x86 architecture. On Alpha, *PALcode* provides a hardware abstraction layer for OSes. It has the highest privilege level on the Alpha architecture and runs on ring 0 among the protection rings. It carries out cache management, TLB miss handling, and interrupt handling instead of the OSes which are on ring 1. Therefore, OSes can be devirtualized and revirtualized

easily using PALcode. In contrast, OSes on the x86 architecture have higher privilege levels as compared to that on the Alpha architecture. They run on ring 0 and can directly access the machine states. Thus, the x86 architecture does not have more privileged software routines than the OS. This feature makes it difficult to implement devirtualization and revirtualization in the x86 architecture environment.

3.2 Devirtualization and Revirtualization Design

3.2.1 *Lightweight hypervisor.*

On-demand virtualization requires a lightweight hypervisor to devirtualize and revirtualize the host machine. As we will devirtualize and revirtualize only one OS, support of a single virtual machine is sufficient for on-demand virtualization. This restriction allows the lightweight hypervisor to have simpler scheduling and memory management code than general-purpose hypervisors such as Xen and KVM. Although the lightweight hypervisor has limited features compared to those of a general-purpose hypervisor, live migration can be implemented on a lightweight hypervisor easily.

There are certain minimum features that the hypervisor must support to perform live migration. Network capability is necessary for live migration because the OS states of the source host must be sent to the target host by the hypervisor, and the lightweight hypervisor needs to support a paravirtualization driver to maintain network connectivity during the live migration step.

3.2.2 *Devirtualization.*

Our approach is to boot into the hypervisor first and devirtualize the host machine so that the hypervisor initializes devices and reserves a memory area for hypervisor. Devirtualization is also performed when the migration step is finished for bare-metal performance. During devirtualization, the current privilege level (CPL) must be zero because we will load several registers onto the processor, such as control registers and descriptor table registers, in the context handover code. The context handover code is code that loads the processor states of the guest kernel. The load instructions of these special registers require CPL 0. In addition, the interrupt flag bit (IF bit) must be cleared to execute the context handover code without interruption. Interrupts are disabled when the IF bit is cleared.

Figure 2 describes the devirtualization process. There are two ways to start devirtualization. First, we can use a loadable kernel module (LKM). To use the LKM, modification of the kernel source code is unnecessary. The LKM contains the code to extend the running kernel and understandably runs on CPL 0. Our devirtualization LKM executes a CLI instruction to clear IF bit to fulfill the requirements of executing the context handover code. It then executes the devirtualization hypercall. Second, we can execute the devirtualization hypercall directly. Instead, the hypervisor must wait until the OS states satisfies the CPL and IF bit requirements. The hypervisor checks the requirements upon every VMExit from the direct devirtualization hypercall called. When the requirements are met, phase 0 ends.

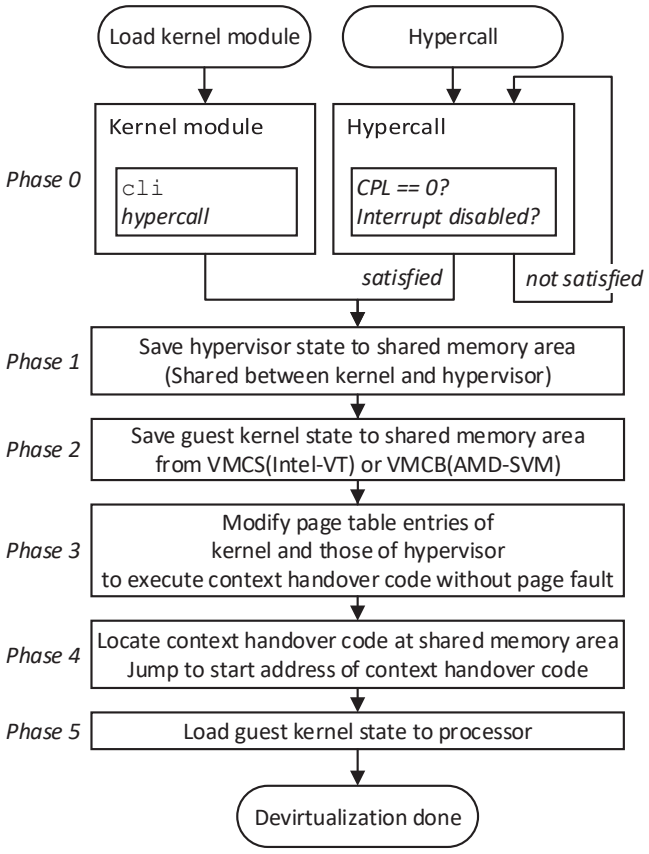

Figure 2: devirtualization process

Whether using a kernel module or hypercall directly, from the next phase, the operations are identical. In phase 1, the hypervisor saves its states to the shared memory area. The shared memory area is reserved at boot time such that the contents cannot change due to general use and is shared between the kernel and the hypervisor. These states will be loaded onto processor when revirtualization is needed. In phase 2, the hypervisor reads guest OS states from virtual machine control structures (VMCS) (or the virtual machine control block (VMCB)). VMCS in Intel and VMCB in AMD are data structures which save the states of the guest OS. These structures contain information about CPU registers, VMEntry and VMExit information, and various types of VM control data. The hypervisor then saves the guest OS states to the shared memory area. The saved guest OS states include the following registers.

- General purpose registers
- Rflags register
- Instruction pointer register
- Control registers
- Debug registers
- Extended-feature-enable register
- System-linkage registers
- User and system segment registers

These registers will be loaded onto the processor later during the execution of the context handover code in phase 5. Before jumping to the context handover code, we must adjust the page table entries of the kernel and those of the hypervisor in phase 3 because a page table base address register will be changed upon the execution of the context handover code. Unless we adjust the page table entries, unexpected behavior or page faults can occur due to unintended page table work. A detailed description of the switching address space is given in Section 4.2.2.

After modifying the page table entries, the hypervisor locates the context handover code at the shared memory area in phase 4. Subsequently, it jumps to the start address of the context handover code. The context handover code loads the guest kernel states which are in the shared memory area onto the processor and executes the far return instruction after pushing the guest instruction pointer to the stack. Consequentially, it jumps to the guest instruction pointer, after which the processor starts to run the guest OS without using the VMRESUME (or VMRUN) instruction.

The VMRESUME (or VMRUN) instruction sets events which will be intercepted during the virtual machine execution step and turns on the memory virtualization features such as an extended page table (EPT) or a nested page table (NPT). As our devirtualization scheme switches from the hypervisor to the OS without the VMRESUME (or VMRUN) instruction, no events will be intercepted by hypervisor, and the memory virtualization feature is turned off continually after devirtualization. Therefore, all virtualization overhead disappears after devirtualization.

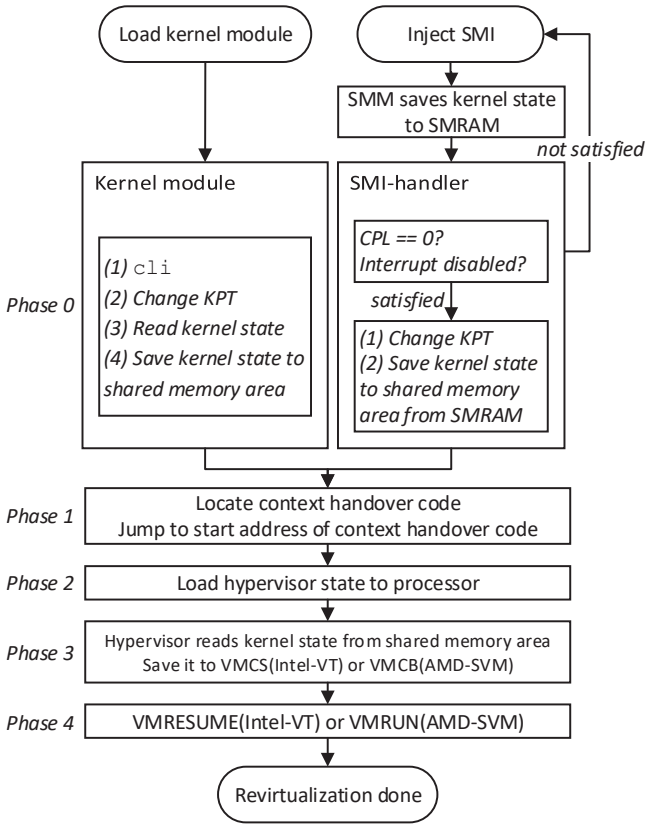

Figure 3: revirtualization process

3.2.3 *Revirtualization.*

We insert a virtualization layer to increase manageability in the bare-metal cloud. As noted earlier in the paper, devirtualization and revirtualization are difficult to implement on the x86 architecture without modification of the OS; specifically, the implementation of revirtualization is much more difficult than that of devirtualization because there is no software routine with higher privileges than the OS. However, in common with devirtualization, revirtualization is feasible by means of a loadable kernel module and the trick of changing the page table entries.

Nevertheless, we wanted the revirtualization scheme to be wholly independent of the OS, including the loadable kernel module. Therefore, we designed and implemented a hardware-assisted revirtualization scheme. There are several necessary requirements for hardware which is capable of revirtualization. It must provide software routines with higher privileges than the OS to stop the OS and must be capable of saving current OS states.

Fortunately, the system management mode (SMM) of the x86 architecture corresponds to hardware that meets the requirements. The SMM is an operating mode which manages system-control activities, such as those related to power management. It operates in an isolated processor environment transparently to the OS. The SMM only can be entered using a system management interrupt (SMI), which is a special external interrupt.

When a SMI is asserted, the processor stops its current execution and saves its states in a separate address space, called SMRAM. Then, a SMI handler is called to service the SMI. Other interrupts and exceptions to the OS are disabled after receiving a SMI. We make the OS load hypervisor states onto the processor and jump to the hypervisor code using the SMI handler.

At this point, we explain the revirtualization process in this section. Figure 3 describes the revirtualization process. We describe phase 0 of the kernel module (software-only) scheme first and that of the SMM (hardware-assisted) scheme later. To revirtualize, the CPL must be zero and the IF bit must be cleared, in common with devirtualization. Hence, the revirtualization LKM initially clears the IF bit. It then changes kernel page table entries so that it can execute the context handover code without a page fault (a detailed description is given in Section 4.2.2). Subsequently, it reads the current processor states (kernel states) and saves them to the shared memory area. The shared memory area is shared by the hypervisor and kernel, akin to the devirtualization process.

The SMM scheme is quite different from the kernel module scheme in phase 0. The SMI is injected to save the kernel

states to SMRAM. Next, the SMI handler checks the kernel states to determine whether CPL is zero and whether the IF bit is cleared. If these requirements are satisfied, it changes the kernel page table entries as the revirtualization kernel module does to prevent a page fault, and it saves the kernel states located in the SMRAM to the shared memory area. In this case, the shared memory area is shared by the SMM as well as the hypervisor and the kernel. If the conditions are not satisfied, revirtualization is not allowed. We periodically inject a SMI until the kernel states meet the requirements for revirtualization.

Both the LKM and SMM scheme locate the revirtualization context handover code in the shared memory area. The kernel module jumps directly to the start address of that code. The SMI handler otherwise changes the rip value to the start address of the revirtualization context handover code in the context-save-area in SMRAM. The processor will jump to that start address when SMM terminates.

From phase 2, the SMM scheme is equivalent of the LKM scheme. As explained earlier in the devirtualization design section, the hypervisor saved its states to the shared memory area prior to devirtualization. The revirtualization context handover code loads these states onto the processor to activate the hypervisor in phase 2. The activated hypervisor reads the guest kernel states which are in the shared memory area and saves them to the VMCS (or VMCB) in phase 3. In phase 4, the hypervisor executes a VMRESUME (or VMRUN) instruction to make the OS run as a guest. In consequence, revirtualization is accomplished.

3.3 Device Model

For a better understanding, we explain the pass-through device model used here and previous work on migrating pass-through devices before we describe our live-migration scheme. Pass-through devices have been widely used in environments where device virtualization is not required to improve the I/O performance. However, as it is not intercepted by the hypervisor, it makes live migration more difficult.

3.3.1 Previous work.

Below are several previous methods which migrate pass-through devices.

The shadow driver suggested by Kadav and Swift [11] tracks the states of a device by intercepting all function calls between the guest OS kernel and the device driver. During migration, the shadow driver redirects the request of the guest OS in the source machine to the driver in the target machine. The returned value is also delivered from the driver of the target machine to the guest OS kernel which is still in the source machine. Although tracking is performed passively with no redirection typically, an interception results in non-negligible performance overhead.

Zhai et al. [20] exploit the bonding driver of Linux with virtual hot plug technology to migrate the pass-through device states. A pass-through driver and a virtualized driver are combined as a bonding driver. Once a hot removal that was injected by a hypervisor is detected, the guest OS disconnects the pass-through driver. After the migration is complete, virtual hot plugging is done by the hypervisor and the bonding driver starts to use the native driver again. A similar mechanism was used in this work, except that virtual hot removal was performed by the guest OS. Section 3.3 presents additional information.

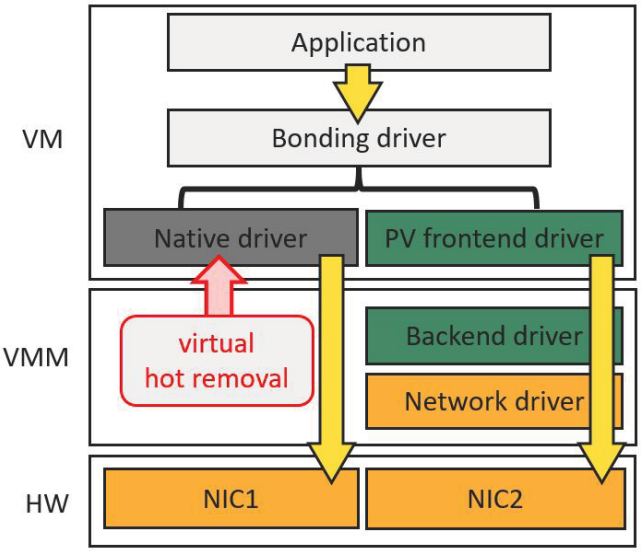

Figure 4: Bonding driver

Fukai et al. [8] had a hypervisor intercept the communication between a guest OS and devices. The unreadable states of devices were inferred from the intercepted data. The write-only states were inferred from the data written to the write-only registers of devices. The internal states were inferred from the data in DMA and I/O communication between the OS and devices. The hypervisor could control the internal states of devices in the target machine by sending dummy packets to the devices. Performance degradation from intercepting states is inevitable, however.

3.3.2 Bonding driver approach.

As it is difficult to pause pass-through devices to save and restore their hardware states, pass-through devices are simply stopped during the migration step. However, during live migration, network connectivity must be maintained.

To maintain network connectivity with a pass-through device, we adopted the scheme established by Zhai et al. [20]. They used a Linux bonding driver that aggregates multiple slave network interfaces into a single logical interface. The bonding driver is included in the Linux mainline today. Under the bonding interface, the slave network driver takes over the network driver states of the primary network driver when the primary driver is turned off.

Figure 4 represents the bonding driver interface used here. We set a native network driver and a paravirtualization (PV) frontend driver as slaves of the bonding driver. When the

host does not need to migrate, the bonding driver uses a pass-through network device via the native network driver by setting the native driver as primary. Therefore, it does not experience overhead caused by device virtualization. When we want to migrate the host, the PV frontend driver is activated by disabling the primary driver. The hypervisor then unregisters the virtio ring buffer to pile network packets in the buffer of the bonding driver and migrates the guest OS states to the target host. After migration is complete, the bonding driver transfers the network packets in its buffer without breaking network connectivity.

3.4 Live Migration

Migration in the bare-metal cloud provides a number of advantages. First, we can migrate heavily communicating machines which are located near each other, migrate a failing machine to a healthy machine, and migrate to a more powerful machine as the load increases [13]. Moreover, machine maintenance is possible without shutting down the host. On-demand virtualization makes it possible to migrate a bare-metal machine. In our on-demand virtualization design, any migration scheme is applicable, e.g., pre-copy memory migration, post-copy memory migration, and a hybrid type because the activated hypervisor can capture the memory access of the OS. However, we simply implemented stop-and-copy migration as a prototype system.

Figure 5 describes our prototype system. In this system, all hosts should initially boot up the hypervisor locally and boot Linux remotely using the Pre-boot eXecution Environment (PXE). Their root file systems are on the PXE server and are accessed by the network file system (NFS). Accordingly, migrating the storage during live migration is unnecessary. At this point, we explain the live migration process.

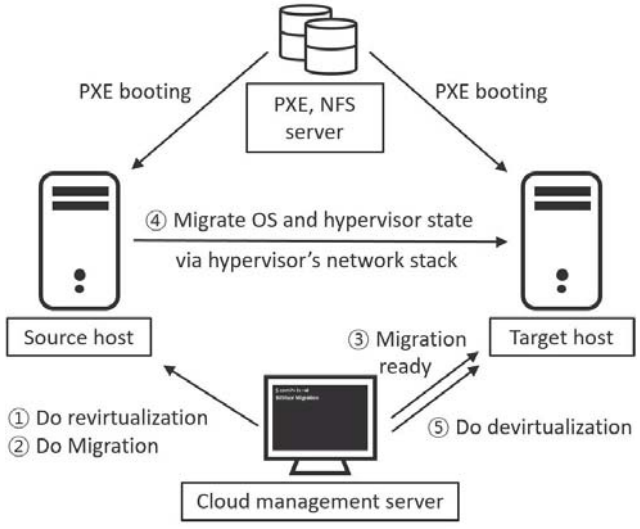

Figure 5: Live migration mechanism

Stage 0 A cloud management server requests revirtualization and live migration at the source host. The source host completes revirtualization. The target host also must be revirtualized.

Stage 1 Both the source host and target host the disable native network driver to replace it with a paravirtualization driver and unregister the virtio ring buffer.

Stage 2 The source host sends the guest OS states, memory contents, and some of the hypervisor states related to the device driver described in Section 4.3.2 to the target host.

Stage 3 The target host receives the states and then loads the guest kernel states to the VMCS or VMCB, the memory contents to memory, and some of the hypervisor states to the hypervisor memory.

Stage 4 The hypervisor at the target host registers the virtio ring buffer to process the stacked network packet at the bonding driver.

Stage 5 The driver of the pass-through network device is activated to disable the PV frontend driver, and the host is devirtualized for bare-metal performance.

4 IMPLEMENTATION

We explain our implementation environment and several implementation issues pertaining to on-demand virtualization and live migration in this section.

4.1 Implementation Environment

As our target system is a x86 architecture-based host, we implemented on-demand virtualization on both an AMD and an Intel processor. Table 1 presents the details of the hardware environment. We used an AMD Phenom TM II X6 1055T processor and an Intel Xeon E3-1230 v5 processor, and we enable only core 0 to ease implementation of the revirtualization and devirtualization process, though multicore implementation of revirtualization and devirtualization is possible. We selected a GIGABYTE GA-880GM-UD2H motherboard for the AMD processor because this motherboard is SMM-unlocked. As the Intel motherboard (GA-X150-PLUS WS) is not SMM-unlocked, we implemented a SMM-based scheme only on the AMD machine. The storage is at a remote NFS server, as stated in Section 3.4.

We implemented a prototype hypervisor based on BitVisor [19]. BitVisor was implemented to run a single virtual machine for system security originally. It can use the pass-through device model to ease revirtualization and devirtualization. BitVisor also has a lightweight TCP/IP stack (lwIP) to migrate the states of the guest OS and the hypervisor between BitVisors. To operate BitVisor, 128MB of memory area is

Component	AMD	Intel
Processor	AMD Phenom II X6 1055T	Intel Xeon E3-1230 v5
Motherboard	GIGABYTE GA-880GM-UD2H	GIGABYTE GA-X150-PLUS WS
Main Memory	DDR3, 4GB	DDR4, 4GB
Network Interface	RTL8169 Ethernet Controller	Intel Ethernet Connection I219-V
Hypervisor	BitVisor-based hypervisor	
Kernel Version	4.1.13	4.4.0
Linux Distribution	Ubuntu 14.04	Ubuntu 16.04

Table 1: Implementation Environment

```
LGDT
LIDT
PUSH RFLAGS
POPFQ   # RFLAGS register
MOV CR3 # Address space changed
VMLOAD
MOV DS
MOV ES
MOV DR6
MOV DR7
WRMSR   # EFER
MOV CR0 # Control registers
MOV CR4
MOV CR2
LSS     # Load stack segment register
MOV RSP
MOV RBP
PUSH CS # popped when execute LRETQ
PUSH RIP
MOV GPRS # RAX, RCX, RDX, RBX, RSI, R8~R15, RDI
LRETQ
```

Figure 6: Context handover pseudo code of AMD-V

```
LGDT
LIDT
PUSH RFLAGS
POPFQ   # RFLAGS register
MOV CR3 # Address space changed
MOV FS
MOV GS
WRMSR   # FS_BASE, GS_BASE
WRMSR   # SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP
MOV DR7
WRMSR   # EFER
MOV CR0 # Control registers
MOV CR4
MOV CR2
MOV DS
MOV ES
LSS     # Load stack segment register
# Unset busy bit of TSS descriptor here
LTR     # Load TR selector
# Restore busy bit
MOV RSP
MOV RBP
PUSH CS selector # popped when execute LRETQ
PUSH RIP
MOV GPRS # RAX, RCX, RDX, RBX, RSI, R8~R15, RDI
LRETQ
```

Figure 7: Context handover pseudo code of Intel VT-x

needed, and the processor must support hardware virtualization technology such as Intel VT-x or AMD-V. We used an unmodified version of Ubuntu Linux 14.04 for AMD and 16.04 for Intel with their respective default kernels. BitVisor can deploy Ubuntu 10.04 and later, as well as CentOS 6.3 and later.

4.2 On-demand Virtualization Implementation Issues

We discuss issues pertaining to on-demand virtualization implementation in this section.

4.2.1 Context Handover Code.

Context handover codes switch the context of the processor by loading the hypervisor or the OS states onto the processor. The context handover codes must be executed atomically without interrupts or exceptions; hence, we clear the IF bit before executing the context handover code, as discussed in Section 3.2.

Figure 6 describes the pseudo code of the context handover code in AMD-V. It initially loads the global descriptor table and the interrupt descriptor table and then load the rflags register using POPFQ instruction. Subsequently, it loads the CR3 register so that the address space is changed to another context. A VMLOAD instruction loads the following registers: FS, GS, TR, LDTR, KernelGsBase, STAR, LSTAR, CSTAR, SFMASK, SYSENTER_CS,SYSENTER_ESP, and SYSENTER_EIP. In turn, the context handover code loads segment selectors, debug registers, EFER, control registers, and registers relating to stack. It then pushes the code segment selector and RIP and loads general purpose registers. Finally, it executes the LRETQ instruction, which pops the RIP and code segment selector and then returns to the instruction pointer (RIP).

Figure 7 shows the pseudo code of the context handover code in Intel VT-x. It is similar to that of AMD-V, but it is slightly different as well. As Intel VT-x does not have a VMLOAD instruction, FS_BASE, GS_BASE, SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP, and TR MSRs must be written explicitly. On the other hand, Intel VT-x has instructions about virtualization, which AMD-V does not have. These are the VMXON and VMXOFF instructions. The VMXON instruction makes the processor enter the VMX root operation. In contrast, The VMXOFF instruction makes

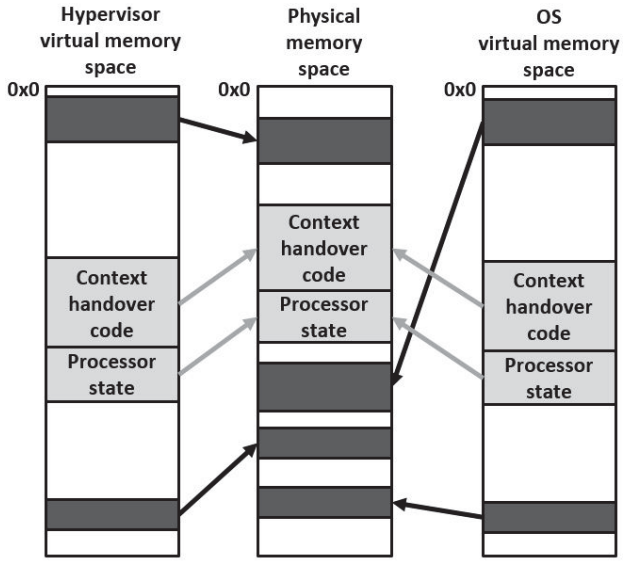

Figure 8: Virtual to physical mapping of context handover code and processor states

the processor run in the VMX non-root operation. Therefore, we execute the VMXOFF instruction upon devirtualization and the VMXON instruction upon revirtualization.

4.2.2 *Switching Address Space.*

While executing the context handover codes, we change the value of the CR3 register. The CR3 register saves the top-level page table base address. When it changes, the mapping of virtual addresses onto physical addresses also changes. Therefore, the virtual addresses of the instruction pointer and the states of the saved registers (to be loaded onto the processor via the context handover code) may index incorrect (meaningless) physical addresses due to the updated page table base address. This causes the processor to encounter a page fault or an unexpected execution. Hence, the virtual addresses of the context handover code and the states of the saved registers of each address space must be identical, and these virtual addresses must index the same physical address, as depicted in Figure 8. To do this, we changed the entries of the kernel page table and the hypervisor page table. The base address of kernel page table can be read in *init_level4_pgt* of the System.map file during the boot process. We carefully choose virtual addresses which both the hypervisor and the OS will not use.

4.3 Live Migration Implementation Issues

As described in Section 3.4, both the source and the target host boot up the hypervisor first by local storage and boot up the guest OS remotely by PXE. We especially used iPXE, which is open-source network boot firmware, as it supports I219-V NIC. We implemented live migration on the Intel Xeon machine. BitVisor can send processor states and memory contents using the Intel pro1000 driver and the TCP/IP stack of BitVisor, lwIP. lwIP is open-source software which provides a light TCP/IP stack, widely used in embedded systems. In the next subsection, we will explain the states, memory contents, and hypervisor states which are sent.

4.3.1 *Guest Processor State and Memory Contents Sent.*

The hypervisor sends the states of the registers in VMCS (or VMCB) of the source host to the target host. The guest processor states which are migrated are presented in Figure 7. They are identical to the states which are changed during on-demand virtualization. After sending the guest processor states, the hypervisor sends the memory contents marked as usable on a physical RAM map. The physical RAM map can be seen at boot time or using the *dmesg* command. We made the usable area of the source host and the target host identical.

4.3.2 *Migrating Hypervisor State.*

Before explaining the migration process of the hypervisor states, we address the device driver of BitVisor. BitVisor supports the Intel pro1000 network driver. Therefore, we used two identical NICs compatible with the pro1000 driver, both of which were the Intel 82540EM. The first transfers the states (using lwIP stack) and the second is coupled with a paravirtualized driver to maintain network connectivity. The pro1000 driver of the second NIC is especially initialized to communicate with the virtio backend driver, which communicates with the virtio (PV) frontend driver of the guest OS using the virtio ring buffer. As BitVisor changes from the source host to the target host, the address of the virtio ring buffer also changes. Therefore, we must transfer the address of the virtio ring buffer. In addition, BitVisor uses a shadow descriptor for the DMA to monitor the DMA activity between the devices and the memory. We transfer the buffer addresses of the shadow descriptors of the BitVisor of the source host to that of the target host as well.

5 EVALUATION AND RESULTS

In this section, two experiments are presented to evaluate the proposed on-demand virtualization process and its application. The first experiment measured and compared the elapsed times of four different schemes during on-demand virtualization. Next, the service downtime arising from the live migration of a bare-metal host was evaluated. The prototype of live migration adopting on-demand virtualization was used for the experiment. The evaluation environment is equal to that described in Section 4.

5.1 Elapsed Time of On-demand Virtualization

The elapsed time of the on-demand virtualization was measured on both the AMD and Intel processors. Two different implementations of devirtualization, a kernel module version and a hypercall version, were evaluated. In the case of revirtualization, a kernel module version and an SMM version (a hardware-assisted version) were evaluated. The experiment for the SMM version could be performed only with the AMD

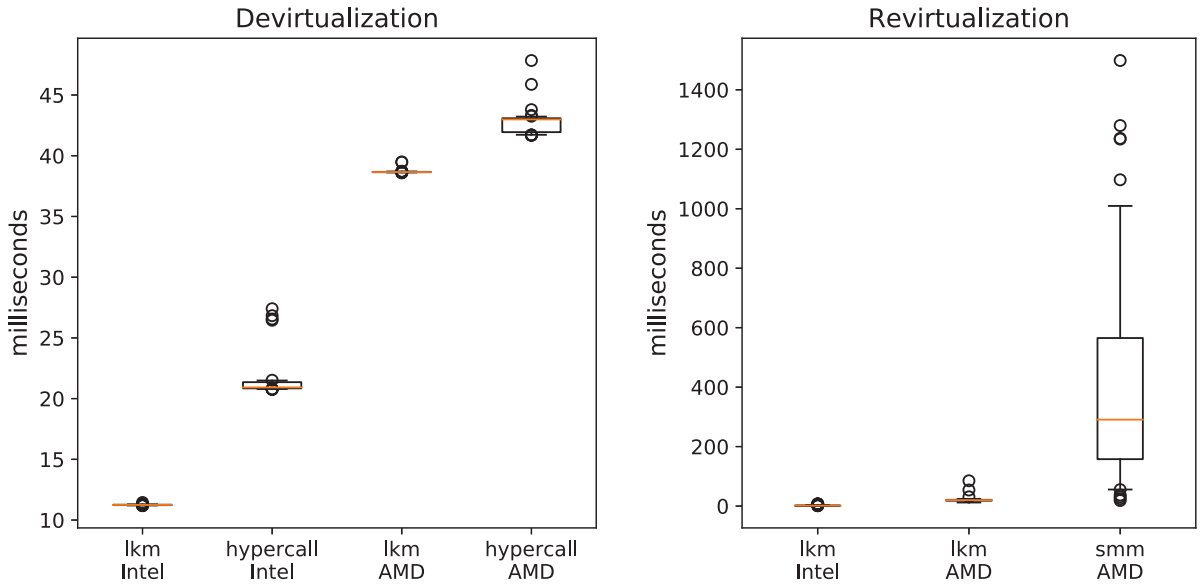

Figure 9: Elapsed time of devirtualization and revirtualization

Scheme	Implementation	Architecture	Average (ms)	95th percentile (ms)	Standard Deviation
Devirt.	Kernel module	AMD	38.67	38.70	0.12
		Intel	11.25	11.29	0.03
	Hypercall	AMD	42.75	43.22	0.84
		Intel	21.27	21.49	1.17
Revirt.	Kernel module	AMD	20.36	24.00	7.98
		Intel	1.73	3.51	1.25
	SMM	AMD	414.51	1013.82	317.32

Table 2: Elapsed time of on-demand virtualization

processor, as we had only an AMD processor and motherboard for which SMM was unlocked. The experiment was repeatedly performed 100 times in each case.

The result is presented in Table 2 and illustrated in Figure 9. In Figure 9, the boxes depict the 25th and 75th percentiles and the whiskers are the 5th and 95th percentiles. Dots are plotted for outliers. The median value is indicated by the horizontal line in the boxes.

The devirtualization scheme using a hypercall took less time on average than the scheme with a kernel module on both the AMD and Intel machines because the scheme with the hypercall must wait until the state of the processor registers meets the proper condition (CPL is 0 and the IF bit is cleared) upon VMExit, resulting in a longer elapsed time. The elapsed time of devirtualization on the AMD machine was longer than that on the Intel machine due to the performance difference between the two machines (AMD Phenom II X6 1055T is 2.8GHz and Intel Xeon E3-1230 v5 is 3.4GHz).

In the revirtualization case, the scheme with SMM required more time to be revirtualized, using 414 ms in average. Similar to the devirtualization scheme with a hypercall, the SMM-based scheme also periodically generates a SMI to check whether the condition is met. In the experiment, the SMI signal was generated by a timer interrupt. This scheme invoked a timer interrupt every 3 ms by setting the value of *PMTimer2* to 6, based on the fact that *PMTimer2* has granularity of 500 μs [2]. Specifically, the SMI handler checks the state of the processor registers every 3 ms, inducing the longest elapsed time. Besides, the standard deviation of the SMM revirtualization scheme was very large, as shown in Table 2, because the condition of revirtualization is checked and met randomly in the SMM scheme different from the LKM scheme (the condition is checked every VMExit).

Devirtualization took longer than revirtualization because the hypervisor had to store its state into memory and change its page table during the devirtualization process, whereas those processes were not required during revirtualization.

5.2 Downtime of Live Migration

The downtime of the live migration process was evaluated on two Intel hosts with the specifications shown in Table 1. As noted in Section 3, the prototype was implemented adopting

a simple stop-and-copy migration technique. The experiment was performed ten times.

A preparation process is required to send the host state (except the memory content) from the source machine to the target machine. During this process, the live-migration module reads the guest processor state from the VMCS and the hypervisor state including a virtio ring buffer address and a shadow descriptor address. The read states are ready to be sent after they are packed into packets. The preparation process took 24.8 ms on average.

Sending of the memory content was mainly responsible for the increase in the downtime of the live migration process. Among the 4GB memory resource, approximately 1.57GB was usable area that should be sent to the target host. The rest of the memory space was reserved for components such as the PCI bus and video ROM. It took 152 s on average to send the usable memory content and the host state. The SSH connection from an external client was maintained after the migration step was completed.

6 CONCLUSION AND FUTURE WORK

We designed and implemented on-demand virtualization on the x86 architecture for the manageability of bare-metal cloud services. Through on-demand virtualization we can migrate a bare-metal host without any OS modifications. We implemented kernel-module-based on-demand virtualization and hardware-assisted on-demand virtualization, which does even not need a kernel module.

Several problems must be addressed in the future. It is necessary to implement on-demand virtualization that supports a multicore system. We plan to synchronize multiple cores to start the devirtualization and revirtualization steps simultaneously. We also have to reduce the number of NICs in our system. Our system has three NICs currently for the bonding driver and migration, but we can reduce this number using a SR-IOV-supported NIC. We need to secure our platform against malicious cloud clients as well. The hypervisor memory area must be protected from the devirtualized guest OS.

On-demand virtualization takes advantage of other features of the virtualization layer. For instance, on-demand virtualization can save the snapshot of an OS in the bare-metal cloud for a rapid restart of a bare-metal instance or can use a checkpoint-recovery mechanism not only for live migration.

ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (No. 2016R1A2B4014109).

REFERENCES

[1] 2009. Redis. (2009). Retrieved August 7, 2017 from https://redis.io/

[2] 2011. AMD SB700 series developer guide. (2011). Retrieved August 15, 2017 from http://support.amd.com/TechDocs/43366.pdf

[3] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, Chau-Wen Tseng, and Donald Yeung. 2005. BioBench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Austin, TX, USA, 2–9.

[4] Andreas Berl, Erol Gelenbe, Marco D. Girolamo, Giovanni Giuliani, Hermann D. Meer, Minh Q. Dang, and Kostas Pentikousis. 2010. Energy-Efficient Cloud Computing. *Comput. J.* 53, 7 (1 Sept. 2010), 1045–1051. https://doi.org/10.1093/comjnl/bxp080

[5] Jui-Hao Chiang, Maohua Lu, and Tzi-cker Chiueh. 2011. Bootstrapped Migration for Linux OS. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, New York, NY, USA, 209–212. https://doi.org/10.1145/1998582.1998631

[6] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-assisted On-demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 565–578. http://dl.acm.org/citation.cfm?id=3026959.3027011

[7] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 171–172. https://doi.org/10.1109/ISPASS.2015.7095802

[8] Takaaki Fukai, Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. 2015. OS-Independent Live Migration Scheme for Bare-Metal Clouds. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. 80–89. https://doi.org/10.1109/UCC.2015.23

[9] Robert P. Goldberg. 1974. Survey of virtual machine research. *Computer* 7, 6 (June 1974), 34–45. https://doi.org/10.1109/MC.1974.6323581

[10] Jinhua Hu, Jianhua Gu, Guofei Sun, and Tianhai Zhao. 2010. A Scheduling Strategy on Load Balancing of Virtual Machine Resources in Cloud Computing Environment. In *Proceedings of the 2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming (PAAP '10)*. IEEE Computer Society, Washington, DC, USA, 89–96. https://doi.org/10.1109/PAAP.2010.65

[11] Asim Kadav and Michael M. Swift. 2009. Live Migration of Direct-access Devices. *SIGOPS Oper. Syst. Rev.* 43, 3 (July 2009), 95–104. https://doi.org/10.1145/1618525.1618536

[12] Thawan Kooburat and Michael Swift. 2011. The Best of Both Worlds with On-demand Virtualization. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1991596.1991602

[13] Michael A Kozuch, Michael Kaminsky, and Michael P Ryan. 2009. Migration without virtualization. In *12th Workshop on Hot Topics in Operating Systems*. IEEE, Monte VeritÃă, Switzerland.

[14] David E Lowell, Yasushi Saito, and Eileen J Samberg. 2004. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. ACM, Boston, MA, USA, 211–223.

[15] Mohamed A. Morsy, John Grundy, and Ingo Müller. 2010. An Analysis of the Cloud Computing Security Problem. In *In Proceedings of APSEC 2010 Cloud Workshop*. https://doi.org/10.1.1.185.438

[16] Tsutomu Nomoto, Yoshihiro Oyama, Hideki Eiraku, Takahiro Shinagawa, and Kazuhiko Kato. 2010. Using a hypervisor to migrate running operating systems to secure. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*. IEEE, Seoul, South Korea, 37–46.

[17] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. 2015. Improving agility and elasticity in bare-metal clouds. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 145–159.

[18] Michael L Powell and Barton P Miller. 1983. Process migration in DEMOS/MP. In *Proceedings of the ninth ACM symposium on Operating systems principles*. ACM, New York, NY, USA, 110–119.

[19] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, et al. 2009. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, New York, NY, USA, 121–130.

[20] Edwin Zhai, Gregory D Cummings, and Yaozu Dong. 2008. Live migration with pass-through device for linux vm. In *The 2008 Ottawa Linux Symposium*. ACM, Ottawa, ON, Canada, 261–268.