Synchronized Co-migration of Virtual Machines for IDS Offloading in Clouds

Kenichi Kourai Hisato Utsunomiya
Department of Creative Informatics
Kyushu Institute of Technology
Fukuoka, Japan
{kourai,U_SAN}@ksl.ci.kyutech.ac.jp

***Abstract*—Since Infrastructure-as-a-Service (IaaS) clouds contain many vulnerable virtual machines (VMs), intrusion detection systems (IDSes) should be run for all the VMs. *IDS offloading* is promising for this purpose because it allows IaaS providers to run IDSes in the outside of VMs without any cooperation of the users. However, offloaded IDSes cannot continue to monitor their target VM when the VM is migrated to another host. In this paper, we propose *VMCoupler* for enabling co-migration of offloaded IDSes and their target VM. Our approach is running offloaded IDSes in a special VM called a *guard VM*, which can monitor the internals of the target VM using *VM introspection*. VMCoupler can migrate a guard VM together with its target VM and restore the state of VM introspection at the destination. The migration processes of these two VMs are synchronized so that the target VM does not run without being monitored. We have confirmed that the overheads of kernel monitoring and co-migration were small.**

***Keywords*-IaaS clouds, virtual machines, migration, intrusion detection systems**

I. Introduction

Infrastructure as a service (IaaS) such as Amazon EC2 provides virtual machines (VMs) for the users. The users set up their own operating systems and applications in the VMs. Unfortunately, the systems inside VMs are not always well maintained and can be penetrated by attackers. To protect such systems, intrusion detection systems (IDSes) are useful. They can monitor the operating systems, networks, and storage of VMs and alert administrators to attacks if they detect symptoms of intrusion. However, it is difficult for IaaS providers to enforce the users to install IDSes in their VMs. Even if the users install IDSes, intruders can easily disable such IDSes running in the VMs before attacking against the systems in them.

To solve these problems, IaaS providers can use *IDS offloading* with *VM introspection* [1]. This technique enables IDSes to run in the outside of their target VM and monitor the VM securely. IDS offloading allows IaaS providers to run IDSes for VMs without any cooperation of the users. Using VM introspection, offloaded IDSes can directly obtain detailed information inside the VM. They are protected from intruders in the VM. However, when the target VM of the IDSes is migrated to another host, the IDSes cannot continue to monitor the VM because they are not migrated together with the VM. Consequently, IaaS providers cannot use both IDS offloading and VM migration.

In this paper, we propose *VMCoupler*, which is the system for enabling co-migration of offloaded IDSes and their target VM. Our idea is running offloaded IDSes in a special VM called a *guard VM* and migrating a guard VM together with its target VM. A guard VM enables IDSes to monitor the internals of the target VM using VM introspection. VMCoupler performs co-migration of a guard VM and its target VM, while the guard VM continues to monitor the target VM. To achieve this, VMCoupler preserves the state of VM introspection in a guard VM during co-migration. In addition, VMCoupler synchronizes the migration processes of these two VMs for security. This guarantees that a guard VM always monitors its target VM while the target VM is running.

We have implemented VMCoupler in Xen 4.0.1 [2]. For memory monitoring, VMCoupler allows a guard VM to map memory pages of its target VM. After the co-migration, it restores the memory-mapping state at a destination host. For network monitoring, VMCoupler performs port mirroring at a virtual switch for a guard VM to capture the packets to/from a target VM. It sets up port mirroring again after co-migration. By using networked storage, a guard VM can monitor the storage of its target VM even after co-migration. We conducted several experiments to examine the overheads of monitoring and co-migration and confirmed that these overheads were small.

The rest of this paper is organized as follows. Section II describes the issues in IDS offloading and VM migration. Section III proposes the system for achieving co-migration of offloaded IDSes and their target VM, which is called VMCoupler. Section IV describes the implementation of VMCoupler in Xen. Section V reports the performance of an offloaded IDS and co-migration. Section VI discusses the related work and Section VII concludes the paper.

II. IDS Offloading and VM Migration

Although IDSes play an important role in IaaS clouds as well as in traditional systems, it is difficult that IaaS providers enforce the users to install IDSes in their VMs. In IaaS clouds, the providers just provide VMs, while the users determine installed software. Therefore the providers cannot install any software including IDSes without users' cooperation. They can require the users to install IDSes, but some users may not follow that for various reasons, e.g., performance overhead or less administrative skills. Even if the users cooperatively install IDSes, such IDSes can be disabled easily by intruders to the target VMs. Intruders with sufficient privileges can stop IDSes or make IDSes ineffective.

IDS offloading is attractive to IaaS providers in that they can deploy IDSes without any cooperation of the users. It enables modular and secure monitoring of VMs. It runs IDSes in the outside of the target VM and prevents interferences from intruders in the VM. Using a technique called VM introspection, offloaded IDSes can monitor the internals of the operating system, the network packets, and the file systems of the target VM with no agent software installed. Offloaded IDSes are often run in a privileged VM called the *management VM*, which is used for managing VMs.

On the other hand, IaaS clouds migrate VMs for various purposes. VM migration allows a running VM to be moved from a source to a destination host. In particular, live migration [3] almost does not stop a VM during its migration by transferring most of its states with the VM running. Using VM migration, IaaS providers can maintain physical hosts without interrupting services provided by VMs. They can perform load balancing by migrating heavily loaded VMs to other lightly loaded hosts. Conversely, they can save power if they consolidate lightly loaded VMs into a fewer hosts.

When a VM is migrated, the IDSes offloaded from the VM should be moved together to the same destination host. If the IDSes were not migrated, they could not continue to monitor the target VM that has been migrated to another host. However, they are not automatically moved with the VM as when they run inside the VM. As a result, the target VM would run without monitoring by IDSes. If attackers intrude into the VM, IaaS providers cannot detect that intrusion. To avoid such an insecure situation, IaaS clouds cannot use VM migration with IDS offloading. This would make IaaS clouds lose various advantages of using VMs.

One approach for migrating IDSes is migrating the management VM where IDSes are offloaded. However, the management VM is a special VM and is not migratable. Since only one management VM has to exist in one host, it cannot be moved out or in. One reason of this restriction is that the management VM handles I/O requests from the other VMs. Another approach is offloading IDSes to a regular VM that is different from their target VM. Although a regular VM can be migrated, IDSes in such a VM cannot monitor the target VM because a regular VM does not have such privileges. The other approach is migrating only IDS processes. Process migration has been well studied, but it cannot preserve several process states including monitoring states of the target VM.

A different approach is re-executing offloaded IDSes at the destination host where their target VM is migrated. If IDSes are short-lived, this approach works well. However, re-execution is not feasible for long-running IDSes such as Tripwire [4]. When a target VM is migrated, the execution of such IDSes is aborted at the source host and is restarted from the beginning at the destination host. At this time, for example, Tripwire has to examine many files again. Memory forensic tools may have to analyze the whole kernel data again. Such wasteful resource consumption should be avoided. A possible solution is transferring the state of IDSes to the destination host, but it needs to re-design IDSes.

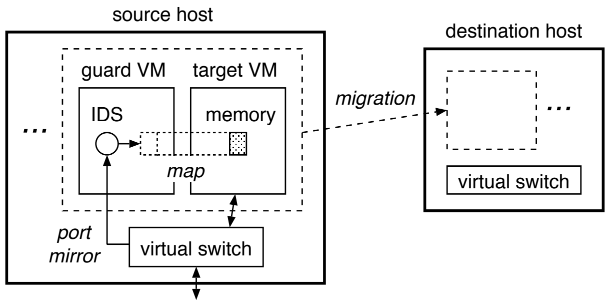

Figure 1. Co-migration of a guard VM and its target VM in VMCoupler.

III. VMCoupler

We propose *VMCoupler* for enabling co-migration of offloaded IDSes and the target VM. VMCoupler provides a special VM called a *guard VM* to run offloaded IDSes. It migrates a guard VM and the target VMs together.

A. Guard VM

A guard VM possesses monitoring functionalities for running offloaded IDSes, as illustrated in Fig. 1. For memory monitoring, it allows IDSes to map memory pages of their target VM. IDSes in a guard VM can read the contents of the mapped memory pages and monitor the state of the target VM. Traditionally, this was allowed only to the management VM. For network monitoring, a guard VM allows IDSes to capture the packets from/to their target VM. To achieve this, VMCoupler performs port mirroring at a virtual switch. Port mirroring duplicates the packets of the target VM to its guard VM. A guard VM provides a dedicated network interface for receiving the duplicated packets. For storage monitoring, a guard VM allows IDSes to read the networked storage used by its target VM. To enable a target VM to be migrated, the networked storage is usually used so that the VM can access its storage at both source and destination hosts.

VMCoupler gives least privilege to a guard VM so that the VM can monitor only one target VM. The management VM binds a guard VM to its target VM and allows the guard VM to map only memory pages of the target VM. It configures port mirroring at a virtual switch so that only the packets of a target VM are delivered to its guard VM. By the access control of the networked storage, a guard VM can access only the storage of its target VM. Even if attackers penetrate a guard VM, they can steal information only from the target VM. In this sense, IDS offloading to a guard VM is securer than that to the management VM. Since the management VM has full privileges for all the VMs, the whole system is compromised if the management VM is compromised.

B. Co-migration with Continuous Monitoring

For the continuity of the monitoring, VMCoupler migrates a guard VM and its target VM together. It groups these two VMs and migrates them in parallel. If we migrate a guard VM just like a regular VM, most of the monitoring states that the guard VM has would be lost. The mapping state of the target VM's memory is not migrated because the traditional migration mechanism assumes that a VM is self-contained. In other words, it is assumed that a VM maps only its own memory pages. The state of port mirroring is also not migrated because the configuration is done in a virtual switch, which is located in the outside of the VM, at the source host.

VMCoupler restores all the monitoring states at a destination host so that a guard VM continues to monitor its target VM. If a guard VM maps memory pages of the target VM at a source host, VMCoupler transfers the mapping state to a destination host. Then it remaps memory pages of the target VM to the address spaces of offloaded IDSes. In addition, VMCoupler reconfigures port mirroring for the packets to/from the target VM at the virtual switch of the destination host. For storage monitoring, a guard VM can continue to monitor networked storage used by the target VM after the migration. VMCoupler does not need to provide any special mechanism for restoring the monitoring state of storage.

C. Synchronized Co-migration

There are two requirements for secure and safe co-migration. One is that a guard VM can always monitor its target VM while the target VM is running. If either a guard VM or a target VM has been migrated earlier, the guard VM could not monitor the target VM running at a different host. The other requirement is that the migration manager for a guard VM can always obtain the necessary information on a target VM. A migration manager is a program for migrating a VM and runs in the management VM at each host. If a target VM has been migrated earlier than a guard VM, the migration manager for the guard VM could not examine the memory allocation to the target VM after that.

To satisfy these two requirements, VMCoupler synchronizes the migration processes of both a guard VM and a target VM, as illustrated in Fig. 2. For security, there are two synchronization points: S_1 and S_4 at source and destination hosts, respectively. S_1 is the point to wait for target VM's stop before guard VM's. A migration manager reaches this

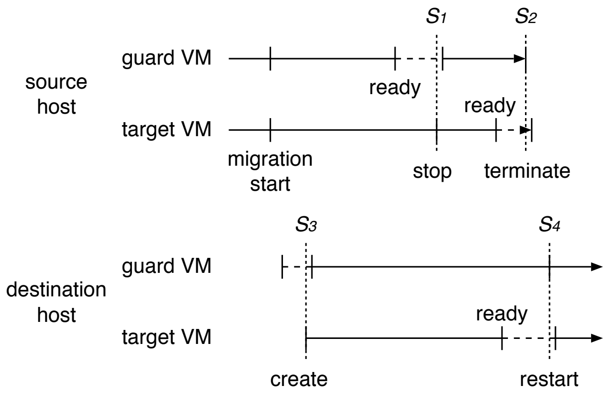

Figure 2. Synchronization points during co-migration.

point when it enters the final stage of live migration. The synchronization at this point guarantees that a guard VM stops after a target VM and that it can monitor a target VM as long as the target VM is running. In contrast, S_4 is the point to wait for guard VM's restart before target VM's. A migration manager reaches this point when it completes to reconstruct a migrated VM. The synchronization at this point guarantees that a target VM is restarted after a guard VM and that it is monitored by a guard VM just after its restart.

For safety, there are also two synchronization points: S_3 and S_2 at destination and source hosts, respectively. S_3 is the point to wait for target VM's creation before guard VM's. A migration manager creates a new VM at the destination host just after migration starts. To restore the mapping of the target VM's memory in a guard VM, a target VM has to have been created. The synchronization at this point guarantees that. In contrast, S_2 is the synchronization point to wait for guard VM's termination before target VM's. A migration manager reaches this point when all states have been transferred. The synchronization at this point guarantees that the migration manager for a guard VM can obtain information on a target VM until the migration of a guard VM completes.

IV. Implementation

We have implemented VMCoupler in Xen 4.0.1 [2]. In Xen, the virtual machine monitor (VMM) runs on top of hardware and executes VMs. The management VM is called *Dom0* and a regular VM including a target VM is called *DomU*. We have developed a guard VM by extending DomU, which is migratable, and we call it *DomM*. DomM runs para-virtualized Linux for monitoring the memory of DomU. In the current implementation, VMCoupler supports para-virtualized Linux running in DomU and targets the x86-64 architecture.

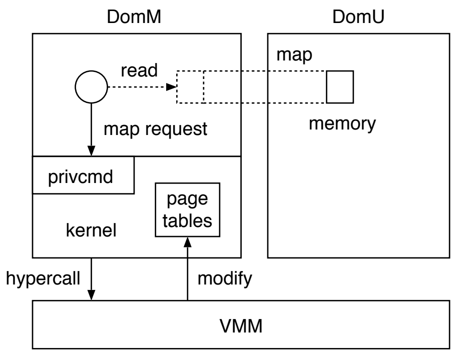

Figure 3. Memory monitoring via the privcmd interface.

A. Memory Monitoring

In Xen, the VMM distinguishes machine memory and pseudo-physical memory to virtualize memory resource. Machine memory is physical memory installed in a host and consists of a set of machine page frames. For each machine page frame, a machine frame number (MFN) is consecutively numbered from 0. Pseudo-physical memory is the memory allocated to VMs and gives the illusion of contiguous physical memory to VMs. For each physical page frame in each VM, a physical frame number (PFN) is consecutively numbered from 0. The VMM maintains the machine-to-physical (M2P) table for the translation from MFNs to PFNs. Para-virtualized Linux maintains the physical-to-machine (P2M) table for translating PFNs to MFNs.

Traditionally, Dom0 maps memory pages of DomU by issuing the update_va_mapping hypercall to the VMM. A hypercall is a mechanism to invoke the function of the VMM. The VMM modifies the page table in Dom0 to map the page specified by an MFN. To obtain such an MFN, Dom0 usually translates a DomU's virtual address by traversing the page tables in DomU. The page directory entry in DomU is obtained by the domctl hypercall. However, VMs except Dom0 cannot map the memory pages of DomU because only Dom0 can issue these hypercalls.

We modified the VMM so that it allows not only Dom0 but also DomM to issue these hypercalls. Dom0 notifies the IDs of DomM and its target DomU of the VMM by a newly created hypercall. This can limit the ability of DomM and allow monitoring only the specified DomU. In addition, we modified the Linux kernel in DomM so that IDS processes can map the memory of DomU (Fig. 3). IDS processes execute such memory mapping through the privcmd interface provided by the Linux kernel that is para-virtualized for Xen. Since privcmd allowed only Dom0 to use its functions, we modified it so that DomM can also use it.

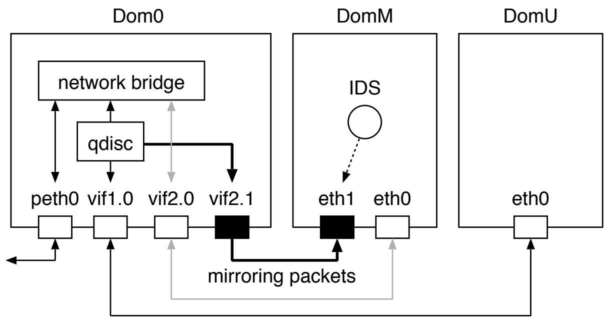

Figure 4. Network monitoring in DomM with port mirroring.

B. Network Monitoring

When DomU is created, a pair of virtual network interfaces (e.g., vif1.0 and eth0) is created in Dom0 and DomU, respectively. In the bridge-networking mode, the interfaces are connected to a network bridge in Dom0, which is connected to physical network interfaces (e.g., peth0). When a packet is sent from DomU, it is delivered to one of the virtual network interfaces and is transmitted to the outside via the network bridge. When a packet to DomU is sent from the outside, it is delivered to one of the virtual network interfaces via the network bridge. When a packet is sent between DomUs, it is delivered from one virtual network interface to another via the network bridge. Therefore, Dom0 can easily capture all the packets from/to DomU via the virtual network interfaces. However, it is not easy for VMs except Dom0 to capture the packets because any packets are not delivered via these VMs.

To enable DomM to capture the packets from/to DomU, Dom0 duplicates the packets by achieving port mirroring with traffic shaping. Fig. 4 depicts port mirroring in Dom0. The traffic shaping in Linux is performed in queuing disciplines (qdisc), which is attached to a network device. A qdisc enqueues all the packets and dequeues them according to registered filters. For port mirroring, Dom0 creates an additional virtual network interface as a mirror port for DomM (e.g., vif2.1). Then it attaches a qdisc to the virtual network interfaces for DomU (e.g., vif1.0). The qdisc receives all the packets via the virtual network interfaces from/to DomU. It duplicates these packets to the mirror port and DomM can receive them via its network interface for port mirroring.

C. Storage Monitoring

Since DomU has to be migrated, its virtual disks are usually located in networked storage such as an NFS server. The simplest solution for monitoring such virtual disks is that both DomU and DomM mount the same root file system via NFS. One disadvantage of this solution is requiring modification to the system configuration in DomU. In IaaS clouds, it is difficult for IaaS providers to enforce specific configuration to the users. Another solution is that DomM mounts an NFS volume including the disk images of DomU and provides them to DomU as virtual disks. This means that DomM is configured as a driver domain [5] to serve virtual disks to DomU. DomM can easily monitor the disk images as Dom0 can traditionally. However, this configuration can affect the storage performance of DomU. To access virtual disks, DomU has to access networked storage via DomM. Then DomM has to access a NFS server via Dom0 because the DomM's network is virtualized.

Therefore, we adopted the solution involving Dom0. Dom0 mounts an NFS volume and provides DomU with the disk images of DomU in it as virtual disks. DomU has to access Dom0, but Dom0 can directly access the NFS server because of no virtualization. DomM also mounts the same NFS volume and monitors the disk images on it by loopback mounts. As a variant of this solution, we also investigated the way that Dom0 provides the disk images of DomU to DomM as secondary virtual disks. However, the monitoring performance was 59% lower than the solution we adopted.

D. Migration of DomM

To migrate DomM, the migration manager in Dom0 transfers the memory and the CPU state of the DomM from a source to a destination host and continues the execution of DomM. In particular, live migration is often used so that the downtime of a VM becomes short. In this case, the migration manager does not stop DomM and repeats to transfer dirty pages, which are modified in DomM during migration. Finally, it stops DomM and transfers the remaining dirty pages and the CPU state. In our storage configuration, Dom0 mounts an NFS volume including the disk images of DomU at both hosts. Even if DomM is migrated, it can access its virtual disks via Dom0. The network interfaces of the migrated DomM are re-connected to the network bridge in Dom0 at the destination host.

During such memory transfers, the migration manager canonicalizes the page table entries (PTEs) used by the DomM at a source host. This canonicalization is rewriting PTEs so that the page tables do not depend on host-specific memory allocation. Specifically, the migration manager replaces the host-specific MFNs in the PTEs with the corresponding PFNs. This translation is performed with the M2P table in the VMM. At a destination host, the migration manager uncanonicalizes those PTEs so that DomM can run with the host-specific page tables. If DomM maps the memory pages of DomU, the uncanonicalization fails at the destination host. After the canonicalization, the page tables of DomM have entries including the PFNs belonging to DomU. When the migration manager uncanonicalizes them, it cannot distinguish DomU's PFNs from DomM's because PFNs are local in each VM.

In VMCoupler, the migration manager for DomM transfers the memory-mapping state on DomU as well. If DomM maps a memory page of DomU, the migration manager sets a

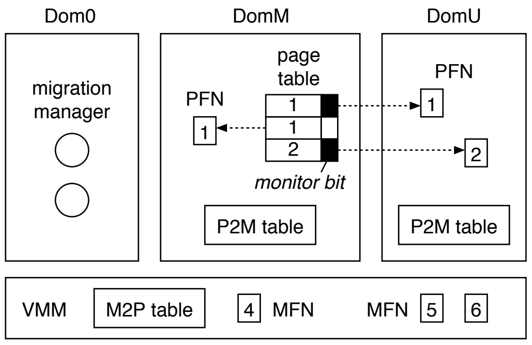

Figure 5. Saving the memory-mapping state for DomU.

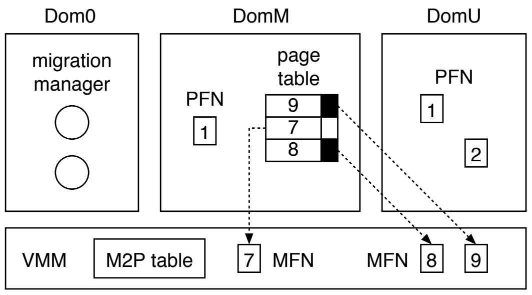

Figure 6. Restoring the memory-mapping state for DomU.

monitor bit to the corresponding PTE, as illustrated in Fig. 5. To examine if a PTE is used for mapping a memory page of DomU, the migration manager re-translates the translated PFN into an MFN. This is done using the P2M table of DomU, which is stored in DomU's memory. If the MFN is equal to the original one before the canonicalization, the migration manager can determine that the PFN belongs to DomU. Otherwise, it re-translates the PFN using the P2M table of DomM to check the validity of the PTE, as performed in the original Xen. The monitor bits are automatically transferred to the destination with memory pages for the page tables.

At the destination host, the migration manager correctly restores the memory-mapping state using the monitor bits. If a monitor bit is set in a PTE, the migration manager considers the PFN included in the PTE as the one of DomU and replaces it with the corresponding MFN, which is allocated to DomU. Fig. 6 shows the page table after reconstruction. For this purpose, the migration manager cannot use the P2M table in DomU yet because the P2M table is reconstructed by the guest operating system itself of DomU after DomU is resumed. Therefore, the migration manager constructs its own P2M table of DomU from the list of MFNs allocated to DomU and the M2P table.

For network monitoring, Dom0 removes the filters set up for port mirroring at the source host after it stops DomM at the final stage of its migration. At the destination host, Dom0 adds the filters for port mirroring again before it restarts DomM. As such, DomM can monitor all the packets that DomU receives. For storage monitoring, DomM can continue to monitor the disk images of DomU after migration because the images are located in an NFS server. Any special mechanisms are not required for the continuity of the storage monitoring because the network connection to the NFS server is kept.

E. Binding DomM to DomU

Dom0 binds DomM to its target DomU by specifying the universally unique identifier (UUID) of DomM. As described in Section IV-A, Dom0 manages a pair of IDs of DomM and its target DomU. However, such domain IDs are local numbers inside a host and not valid in another host after migration. Therefore, we used UUIDs, which are globally unique numbers. An administrator assigns a fixed UUID to DomU and specifies it as the target for DomM. To this end, we added a new configuration option target_uuid to DomM. Since the configuration of DomM is transferred to the destination at migration, Dom0 at the destination can re-bind DomM to DomU on the basis of the UUID of DomU.

F. Synchronized Co-migration

Two migration managers migrate DomU and DomM synchronously as shown in Fig. 2. First, they create new VMs at a destination host and synchronize their migration processes at S_3. To wait for DomU's creation, the program for resuming DomM repeatedly looks up DomU by its UUID specified in the target_uuid configuration option. If it can find that DomU, it proceeds the migration of DomM.

After the migration managers reach the final stage of live migration, they synchronize their migration process at S_1. To wait for DomU's stop, the migration manager for DomM repeatedly obtains information on DomU until DomU becomes the stopped state. While the migration manager waits for DomU's stop, it continues to transfer dirty pages of DomM to the destination. This can keep the number of dirty pages to be transferred at the final stage as small as possible. However, if the migration manager has to wait for a long time at S_1, the total amount of transferred memory increases. This depends on the memory access patterns in DomM. Therefore, we also prepare an option to transfer no dirty pages during this waiting time.

When the migration managers terminate old VMs at the source host, they synchronize the tasks at S_2. To wait for DomM's termination, the migration manager for DomU repeatedly obtains information on DomM while DomM exists. Finally, they synchronize the restart of the new VMs at S_4. The migration manager for DomU repeatedly examines the state of DomM until DomM is restarted. It can identify the DomM monitoring the DomU because a pair of DomM and DomU is registered to the VMM at the above S_3.

V. Experiments

We conducted experiments to examine the continuity of monitoring across co-migration and to measure the performance of monitoring and co-migration. For server machines hosting VMs, we used two PCs with one Intel Quad Core 2.83 GHz processor, 8 GB of memory, and a gigabit Ethernet. We used Xen 4.0.1 and ran Linux 2.6.32.38 in Dom0, DomM, and DomU. By default, we allocated one virtual CPU and 512 MB of memory to DomM, one virtual CPU and 1 GB of memory to DomU, and four virtual CPUs and the rest of the memory to Dom0. For a NFS server, we used NAS with one Intel Xeon X5640 3.16 GHz processor, 32 GB of memory, 16 TB of RAID-5 disks, and a gigabit Ethernet. These PCs and NAS are connected with a gigabit Ethernet switch.

A. Memory Monitoring

We executed the integrity checker of the DomU kernel in DomM. The integrity checker calculates the hash value of the memory area for the kernel text and detects tampering with it. We compared the hash value with the one pre-calculated from the kernel image and confirmed that the integrity checker could correctly monitor the kernel in DomU. Even if we co-migrated DomM and DomU during the integrity check, the checker could continue to run and complete its check in the destination host.

Next, we measured the time needed for the integrity check. For comparison, we executed the integrity checker in Dom0 as traditional and measured the time. We ran the integrity checker 100 times. On average, it took 135 ms and 203 ms for the integrity checks in DomM and Dom0, respectively. The integrity check running in DomM was 33% faster than that in Dom0.

According to our analysis of the implementation in Xen and Linux, we found that the number of virtual CPUs allocated to a VM affected the performance of memory mapping. When DomM and Dom0 map memory pages of DomU, they issue the hypercall for obtaining the state of virtual CPUs. They allocate a buffer passed to the hypercall and lock it by using the mlock system call so that the corresponding memory pages are not paged out. Since the system call waits for all the CPUs to synchronously complete pending operations on memory pages, the execution time is proportional to the number of CPUs.

Fig. 7 shows the time for the integrity check when we changed the number of virtual CPUs allocated to DomM and Dom0. The results show that the time is proportional to the number of virtual CPUs. For four virtual CPUs, the performance in DomM was changed largely. This is probably because the PC had only four physical CPUs and CPU contention occurred between DomM and Dom0.

In general, memory monitoring in DomM can be faster than that in Dom0. For DomM, one or small number of virtual CPUs are sufficient if only one or several IDSes

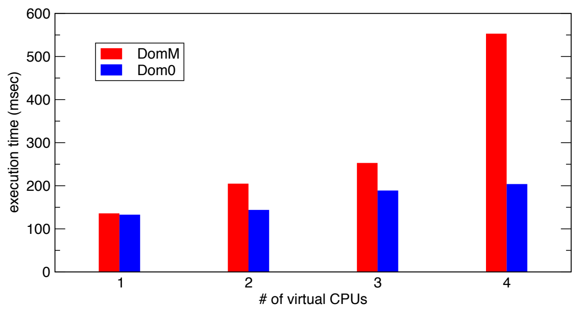

Figure 7. The impacts of various numbers of virtual CPUs on the kernel monitoring.

are running. In contrast, Dom0 requires many virtual CPUs because it has to handle I/O requests from all the VMs.

B. Storage Monitoring

We executed Tripwire [4] in DomM to scan the DomU's disk. Tripwire saves the correct state of the file systems to its database and detects changes to them. We confirmed that Tripwire in DomM could monitor the DomU's disk correctly as it ran inside DomU. Then, we co-migrated DomM and DomU while Tripwire in DomM was monitoring the DomU's disk. Across the migration, Tripwire could complete the integrity check of the entire disk.

Next, we measured the time needed for the integrity check by Tripwire in DomM. For comparison, we also executed Tripwire in Dom0 and measured the time for the check. On average, it took 18.9 seconds and 4.5 seconds for the integrity check in DomM and Dom0, respectively. The time in DomM was 4.2 times longer than that in Dom0. The primary reason is network virtualization. Tripwire in DomM and Dom0 had to access the storage of DomU in the NFS server. Since the network of DomM is virtualized, its network access is performed via Dom0.

C. Network Monitoring

We executed Snort [6] in DomM to check the DomU's packets. Snort is a signature-based network IDS. Snort in DomM could capture all the packets to/from DomU. When we mounted portscans to DomU using nmap, Snort in DomM could detect the portscans. Next, we performed co-migration of DomM and DomU while Snort in DomM monitored the packets for DomU. Since port mirroring in Dom0 was disabled after DomU stopped and enabled before DomU restarted, Snort did not drop any packets that DomU received.

To examine the overhead of network monitoring in DomM, we measured the increase of the CPU utilization in DomM by running Snort. As a result, the CPU utilization did not increase.

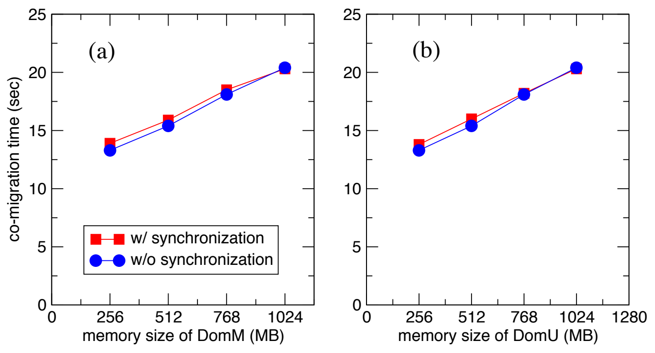

Figure 8. The co-migration time for various combinations of the memory sizes of DomM and DomU.

D. Co-migration Time

We measured the time needed for synchronized co-migration of DomM and DomU when we changed the size of memory allocated to the VMs. First, we allocated 1 GB of memory to DomU and changed the memory size of DomM from 256 MB to 1 GB. For comparison, we migrated two independent DomUs in parallel without synchronization. We fixed the memory size of one DomU to 1 GB and changed the memory size of the other DomU. The time we measured was from when we started co-migration until the migration of both VMs completed.

Fig. 8(a) shows the co-migration times. As the memory size of DomM became larger, the co-migration time was increasing. This is because the total migration time of two VMs depends on the total memory size to be transferred. The difference between co-migration with and without synchronization was small. The co-migration time was increasing as the memory size of DomM became smaller. However, even when the memory size of DomM was 256 MB, the synchronization increased the co-migration time only by 0.6 second.

Next, we fixed the memory size of DomM to 1 GB and changed that of DomU from 256 MB to 1 GB. Fig. 8(b) shows the co-migration times. Similar to the above experiment, the time for co-migration with synchronization was almost the same as that without synchronization.

E. Downtime

We measured the downtime of DomU during synchronized co-migration with DomM. Since various services are running in DomU, its downtime should be short. The downtime of DomU can increase at the synchronization points S_2 and S_4, at which DomU waits for DomM. First, we allocated 1 GB of memory to DomU and changed the memory size of DomM from 256 MB to 1 GB. The downtime we measured was the time in which DomU is not running at either source or destination host. We measured the downtime 10 times.

Fig. 9(a) shows the average downtimes. As the memory size of DomM became larger, the downtime was increasing

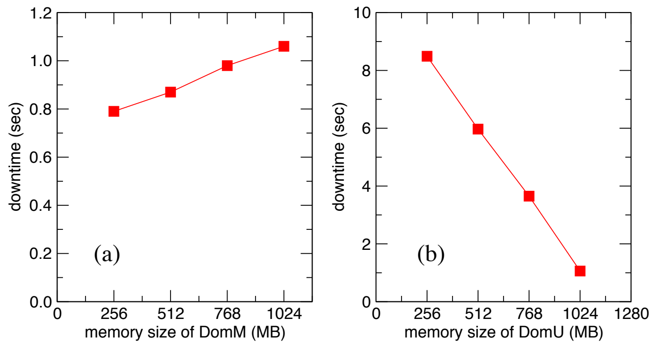

Figure 9. The downtimes of DomU for various combinations of the memory sizes of DomM and DomU.

Table I
THE WAIT TIMES AT EACH SYNCHRONIZATION POINT FOR VARIOUS MEMORY SIZES OF DOMM (MS).

	DomM		DomU	
size (MB)	S_1	S_3	S_2	S_4
256	7736	0	14	25
512	5200	0	18	51
768	2675	0	19	61
1024	74	0	92	69

gradually. When we changed the memory size of DomM from 256 MB to 1 GB, the downtime increased only by 162 ms. This means that synchronized co-migration does not affect the downtime largely.

Next, we measured the downtimes of DomU when we fixed the memory size of DomM to 1 GB. We changed the memory size of DomU from 256 MB to 1 GB. The results are shown in Fig. 9(b). It is shown that the downtime was dramatically increasing as the memory size of DomU became smaller. However, the memory size of DomM is usually smaller than that of DomU because DomM only monitors DomU. Therefore, the configuration of VMs in Fig. 9(b) is a special case.

F. Breakdown of Synchronization

To examine the breakdown of synchronization, we measured the wait times at each synchronization point. First, we fixed the memory size of DomU to 1 GB and changed that of DomM from 256 MB to 1 GB. Table I shows the average wait times when we measured them 10 times.

As the memory size of DomM became smaller, the wait time increased at S_1. The reason is that the memory transfer of DomM completes in a shorter time than that of DomU and DomM is ready to stop earlier. Note that the wait time at S_3 was always zero. In other words, DomU was created before DomM at the destination host without synchronization. In contrast to S_1, the wait time was decreasing at S_2 as the memory size of DomM was smaller. This is because the remaining state of DomM can be transferred faster than that of DomU after S_1. For S_4, the wait time decreased as the memory size of DomM became smaller. This means that

Table II
THE WAIT TIMES AT EACH SYNCHRONIZATION POINT FOR VARIOUS MEMORY SIZES OF DOMU (MS).

	DomM		DomU	
size (MB)	S_1	S_3	S_2	S_4
256	0	0	7696	51
512	0	0	5109	101
768	0	0	2732	69
1024	74	0	92	69

DomM of a smaller memory size can complete migration and restart earlier than DomU.

Next, we fixed the memory size of DomM to 1 GB and changed that of DomM from 256 MB to 1 GB. Table II shows the wait times at each synchronization point. At S_1, DomM did not wait for DomU's stop because DomU completed to transfer its memory earlier. Instead, the wait time at S_2 became longer as the memory size of DomU was smaller because DomU reached the point earlier.

G. Co-migration of Write-intensive VMs

We examined the impacts on the performance of co-migration when VMs modified its memory aggressively. We ran a program that modified the specified number of memory pages at the specified interval. Note that the program modified one byte per page. The memory sizes of DomM and DomU were 512 MB and 1 GB, respectively.

First, we measured the co-migration time when we ran the program in DomU. The program modified 125000 pages (about 500 MB) at a time. Fig. 10(a) shows the results for various write intervals. As the write interval increased, the co-migration time became longer when the interval was less than 750 ms. This is because the number of the iterations of transferring dirty pages exceeded the maximum. The migration manager transfers only the pages that are modified in the previous iteration but are not modified in the current one. When the frequency of the memory writes is too high, most of the dirty pages are not transferred. As a result, the iteration proceeded faster and reached the maximum count earlier.

On the other hand, when the interval was more than 750 ms, the co-migration time did not increase and it was about 40 seconds. This is because the total amount of transferred memory exceeded the maximum. Since the frequency of memory writes was not so high, the migration manager transferred more dirty pages to the destination host. Therefore, the amount of transferred memory reached the maximum at the certain time.

Next, we measured the co-migration time when we ran the program in DomM. The program modified 100000 pages (about 400 MB) at a time because the memory size of DomM was 512 MB. Fig. 10(b) shows the results for various write intervals. Similar to write-intensive DomU, the co-migration time was proportional to the interval when the

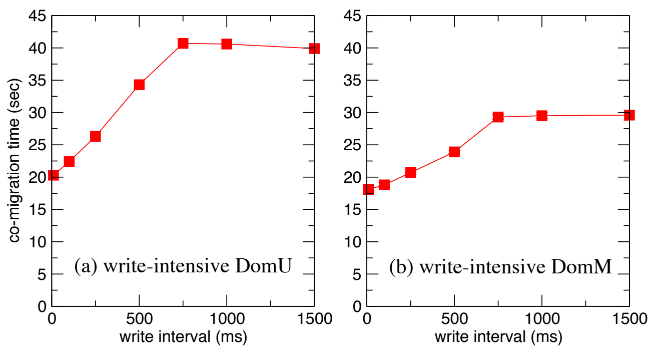

Figure 10. The co-migration time for write-intensive DomU and DomM.

interval was less than 750 ms. It was about 30 seconds when the interval was more than 750 ms.

VI. Related Work

For Xen, various special-purpose VMs have been proposed to divide the privileges of Dom0, which is called Dom0 disaggregation. Driver domains [5] run device drivers in VMs different from Dom0. IDSes can be run in driver domains to monitor networks and storage. Stub domains [7], [8] enable running QEMU used for device emulation. Since they are allowed to access the memory of DomUs for device emulation, IDSes in them can monitor the memory. Also, they can intercept device accesses and check the integrity. DomB [9] is used to boot DomU, instead of Dom0. It loads a kernel image into the DomU's memory and sets up DomU. Xoar [10] disaggregates Dom0 into many single-purpose VMs called service VMs. However, these VMs are not designed to be migratable because they are helpers for Dom0.

A self-service cloud (SSC) computing platform [11] provides users with special-purpose VMs called service domains (SDs) to monitor their own VMs. SDs can introspect the memory of target VMs and monitor accessed disk blocks and issued system calls. Also, they can intercept disk accesses and encrypt disk blocks. In SSC, a user's meta-domain consists of user's own Dom0 (Udom0), DomUs, SDs, and mutually trusted service domains (MTSDs). These VMs should be migrated together because of their strong association, but SSC does not support such co-migration.

Live gang migration [12] efficiently achieves concurrent migration of multiple co-located VMs. To reduce the migration overhead, it transfers memory contents that are identical across VMs only once. It tracks identical memory contents across VMs and performs memory de-duplication for all the migrated VMs. It also applies differential compression to nearly identical memory pages. Unlike VMCoupler, live gang migration does not synchronize between the migration process of multiple VMs. This approach can be incorporated into VMCoupler to reduce the co-migration time of a guard VM and its target VM.

Process migration cannot preserve the monitoring states of a target VM during the migration of IDS processes. Since IDS processes map the memory of another VM, the operating system itself in Dom0 cannot migrate the mapping state. In addition, most of the systems supporting process migration such as libckpt [13] and BLCR [14] do not preserve process states completely. For example, open files and sockets are usually closed. Distributed operating systems such as Amoeba [15] and MOSIX [16] can migrate processes with process states preserved, but the requests are simply forwarded to a source host. Therefore the source host cannot be stopped.

Compute capsules [17] and the pod abstraction in Zap [18] enable a group of processes to be migrated as a unit. They provide a thin virtualization layer on top of the operating system and group processes with a private namespace. In Zap, particularly, migrated processes can preserve network connections and inter-process communication between them, including shared memory. In some sense, this is similar to our co-migration mechanism of VMs. Like the other systems supporting process migration, Zap cannot migrate offloaded IDS processes with the memory-mapping state of a target VM.

VII. Conclusion

In this paper, we proposed VMCoupler, which enables synchronized co-migration of offloaded IDSes and their target VM. Offloaded IDSes are run in a guard VM and monitor its target VM using VM introspection. VMCoupler synchronizes the migration processes of a guard VM and a target VM so that a guard VM can always monitor a running target VM. Our experiments showed that the overheads of monitoring and co-migration were small and that the downtime of a target VM was short.

Our future work is decreasing the downtime of a target VM due to the synchronized co-migration. We need to add another synchronization point that a target VM waits for a guard VM to be ready for the final stage of the migration. Another direction is extending VMCoupler to support various combinations of guard VMs and target VMs as a group. Currently, one guard VM is necessary for one target VM, but one guard VM could monitor multiple target VMs. In this case, VMCoupler has to migrate more than two VMs simultaneously, so that it could impact the migration performance and the downtime of target VMs. Also, we are planning to extend VMCoupler to domains other than IDS offloading, such as out-of-band remote VM management [19].

Acknowledgment

This research was supported in part by JST, CREST.

References

[1] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live migration of virtual machines,” in *Proc. Symp. Networked Systems Design and Implementation*, 2005, pp. 273–286.

[4] G. Kim and E. Spafford, “The design and implementation of Tripwire: A file system integrity checker,” in *Proc. ACM Conf. Computer and Communications Security*, 1994, pp. 18–29.

[5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, “Safe hardware access with the Xen virtual machine monitor,” in *Proc. Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, 2004.

[6] M. Roesch, “Snort – lightweight intrusion detection for networks,” in *Proc. USENIX System Administration Conf.*, 1999.

[7] J. Nakajima and D. Stekloff, “Improving HVM domain isolation and performance,” in *Xen Summit September 2006*, 2006.

[8] S. Thibault, “Stub domains,” in *Xen Summit Boston 2008*, 2008.

[9] D. G. Murray, G. Milos, and S. Hand, “Improving Xen security through disaggregation,” in *Proc. Intl. Conf. Virtual Execution Environments*, 2008, pp. 151–160.

[10] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, “Breaking up is hard to do: Security and functionality in a commodity hypervisor,” in *Proc. Symp. Operating Systems Principles*, 2011, pp. 189–202.

[11] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, “Self-service cloud computing,” in *Proc. Conf. Computer and Communications Security*, 2012, pp. 253–264.

[12] U. Deshpande, X. Wang, and K. Gopalan, “Live gang migration of virtual machines,” in *Proc. Intl. Symp. High Performance Distributed Computing*, 2011, pp. 135–146.

[13] J. S. Plank, M. Beck, G. Kingsley, and K. Li, “Libckpt: Transparent checkpointing under Unix,” in *Proc. USENIX Winter 1995 Technical Conference*, 1995, pp. 213–223.

[14] J. Duell, P. Hargrove, and E. Roman, “The design and implementation of Berkeley lab’s Linux checkpoint/restart,” LBNL, Tech. Rep., 2002.

[15] S. J. Mullender, G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren, “Amoeba a distributed operating system for the 1990s,” *IEEE Computer*, vol. 23, no. 5, pp. 44–53, 1990.

[16] A. Barak and R. Wheeler, “MOSIX: An integrated multiprocessor UNIX,” in *Proc. USENIX Winter 1989 Technical Conference*, 1989, pp. 101–112.

[17] B. K. Schmidt, “Supporting ubiquitous computing with stateless consoles and computation caches,” Ph.D. dissertation, Stanford University, 2000.

[18] S. Osman, D. Subhraveti, G. Su, and J. Nieh, “The design and implementation of Zap: A system for migrating computing environments,” in *Proc. Symp. Operating Systems Design and Implementation*, 2002, pp. 361–367.

[19] T. Egawa, N. Nishimura, and K. Kourai, “Dependable and Secure Remote Management in IaaS Clouds,” in *Proc. Intl. Conf. Cloud Computing Technology and Science*, 2012, pp. 411–418.