Resource Cages: A New Abstraction of the Hypervisor for Performance Isolation Considering IDS Offloading

Kenichi Kourai*, Sungho Arai†, Kousuke Nakamura*, Seigo Okazaki*, and Shigeru Chiba‡
*Kyushu Institute of Technology
†Tokyo Institute of Technology
‡The University of Tokyo

***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 in that it allows IaaS providers to run IDSes outside VMs without any cooperation of users. However, IDS offloading makes performance isolation between VMs difficult because IDSes offloaded from a VM consume resources outside the VM. As a result, the total resource usage of the VM and the offloaded IDSes exceeds the limits configured to the VM. In this paper, we propose a new abstraction of the hypervisor, called a *resource cage*. A resource cage can manage a VM and offloaded IDSes as a group and achieves performance isolation between resource cages, e.g., CPU limits, CPU shares, and memory limits. In addition to performance isolation, it keeps high resource utilization for a VM and offloaded IDSes as much as possible. We have implemented resource cages in Xen and KVM. Our experiments showed that resource cages could control the resource usage of a VM and offloaded IDSes effectively.**

1. Introduction

Infrastructure as a Service (IaaS) such as Amazon EC2 provides virtual machines (VMs) for users. 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, storage, and networks of VMs and alert administrators to attacks if they detect symptoms of intrusion. However, it is difficult for IaaS providers to enforce users to install IDSes in their VMs. Even if 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 outside their target VMs and monitor the VMs securely. IDS offloading allows IaaS providers to run IDSes for VMs without any cooperation of users. However, it makes performance isolation [2] between VMs difficult. In a virtualized system, the hypervisor can guarantee that each VM does not use more than a certain amount of resources such as CPUs and memory. In IDS offloading, IDSes offloaded from a VM consume resources outside the VM. As a result, the total resource usage can exceed the limits configured to the VM.

For resource management considering IDS offloading, this paper proposes a *resource cage*, which is a new abstraction of the hypervisor. A resource cage manages a VM and IDSes offloaded from it as a group. The hypervisor assigns CPUs and memory to resource cages, not VMs. As such, a resource cage achieves performance isolation for a group of a VM and offloaded IDSes. For example, the total CPU utilization of a VM and offloaded IDSes can be kept up to 50% even if the IDSes consume much CPU time. Similarly, a VM and IDSes can use only 1 GB of memory in total even if the IDSes consume large amount of memory. In addition to such performance isolation, a resource cage keeps high resource utilization for a VM and offloaded IDSes as much as possible. When either a VM or an IDS in a resource cage is busy, the resource cage can fully use the CPU time assigned to it.

We have implemented resource cages in Xen 4.1 [3]. The system consists of a VM scheduler, an IDS scheduler, and a memory scheduler. The VM scheduler is based on the credit scheduler in Xen and calculates credits considering resource cages. The IDS scheduler monitors the execution time of IDS processes in the hypervisor and controls the execution of them using the mechanism of the operating system. The memory scheduler monitors the memory usage of IDSes including the page cache and adjusts the memory sizes of VMs. Also, we have implemented resource cages in KVM 1.1.2 [4], which has an architecture largely different from Xen. From our experimental results, it was shown that resource cages could achieve both performance isolation and high resource utilization in IDS offloading.

The rest of this paper is organized as follows. Section 2 describes the issues of performance isolation in IDS offloading. Section 3 proposes a resource cage for managing a VM and offloaded IDSes as a group. Section 4 describes the implementation details in Xen and KVM. Section 5 reports the results of performance isolation using resource cages. Section 6 discusses related work and Section 7 concludes the paper.

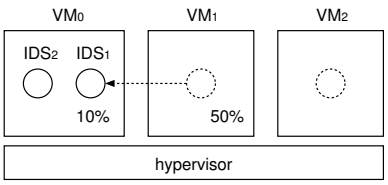


Figure 1: An example of IDS offloading.

2. Motivation

2.1. Performance Isolation in IDS Offloading

Although IDSes play an important role in IaaS clouds, it is difficult that IaaS providers enforce users to install IDSes in their VMs. In IaaS clouds, providers just provide VMs, while users decide installed software. *IDS offloading* is attractive to IaaS providers in that they can deploy IDSes without any cooperation of users. It runs IDSes outside their target VMs and prevents interferences from intruders in the VMs. Using a technique called *VM introspection* [1], offloaded IDSes can monitor the internals of the operating system, file systems, and network packets of the target VMs with no agent software installed.

However, IDS offloading makes performance isolation between VMs difficult. In a virtualized system, each VM is strongly isolated by the hypervisor, which runs underneath all the VMs. For performance, the hypervisor can guarantee that each VM does not use more than a certain amount of resources such as CPUs and memory. For example, the hypervisor can set the *upper limit* of CPU utilization to each VM. It can also set the maximum memory size to each VM. In addition, the hypervisor can set CPU *shares* between VMs. According to the shares, it relatively allocates the CPU time to VMs.

In IDS offloading, on the other hand, offloaded IDSes are executed in a different execution environment, which is, e.g., the *management VM* in Xen. In other words, offloaded IDSes consume resources in the management VM. This violates performance isolation between VMs. For example, assume that the hypervisor limits the CPU utilization of VM_1 to 50%, as shown in Fig. 1. If an IDS offloaded from the VM consumes 10% of the CPU time in the management VM (VM_0), VM_1 and the IDS can use 60% of the CPU time in total. Since the offloaded IDS is part of VM_1 originally, the total CPU utilization of VM_1 and the IDS should be 50% for fairness. For memory limits, similar problems arise by IDS offloading.

2.2. Existing Resource Controls

The violation of performance isolation due to IDS offloading cannot be resolved by simply combining the existing resource control mechanisms of the hypervisor and the operating system in the management VM. For CPUs, the hypervisor can configure VMs by limits and shares, as described above. Similarly, the operating system can configure IDS processes by CPU limits and shares. Although several resource controls can accomplish performance isolation, they result in low resource utilization. Let us consider two VMs and two offloaded IDSes as in Fig. 1. IDS_1 and IDS_2 are offloaded from VM_1 and VM_2, respectively, and are executed in VM_0, which is the management VM.

TABLE 1: Possible configurations of CPU limits (%) and shares (no unit).

config	VM_1	VM_2	VM_0	IDS_1	IDS_2
A	40%	40%	20%	10%	10%
B	40%	40%	20%	1	1
C	2	2	1	10%	10%
D	2	2	1	1	1

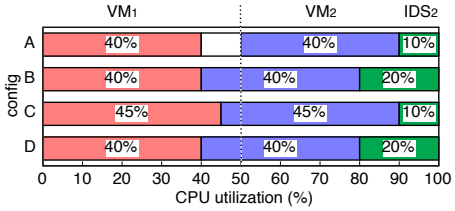


Figure 2: The CPU utilization when IDS_1 is idle.

One of our goals is to limit the total CPU utilization of VM_1 and IDS_1 and that of VM_2 and IDS_2, e.g., to 50%, respectively. Table 1 shows possible configurations of CPU limits and shares to the VMs and IDSes. When all the VMs and IDSes are busy, all the configurations can accomplish the goal. Each IDS receives 10% of the CPU time, while each VM receives 40%. However, when IDS_1 is idle, for example, config A can achieve only low CPU utilization, as shown in Fig. 2. This is because the total CPU utilization of VM_1 and IDS_1 becomes 40% at maximum. When VM_1 is busy, it should be able to receive 50% of the CPU time for higher CPU utilization.

In contrast, the other configurations violate performance isolation when IDS_1 is idle. In config B, IDS_2 can receive 20% of the CPU time. Therefore, the total CPU utilization of VM_2 and IDS_2 becomes 60%. In config C, each VM can use 45% of the CPU time because IDS_2 can use only 10% of the CPU time and the remaining is shared by the two VMs. As a result, the total CPU utilization of VM_2 and IDS_2 exceeds 50%. In config D, IDS_2 can use 20% of the CPU time as in config B. This results in violating performance isolation for VM_2 and IDS_2.

The other goal is to allocate the CPU time between a pair of VM_1 and IDS_1 and that of VM_2 and IDS_2 proportionally. Let us consider assigning CPU shares to the two pairs in a 1:1 ratio. When all the VMs and IDSes are busy, all the configurations can accomplish this goal because both pairs can use 50% of the total CPU time. However, when IDS_1 is idle, for example, performance isolation is violated. From Fig. 2, the ratios of the allocated CPU time become 4:5, 2:3, 9:11, and 2:3 in config A, B, C, and D, respectively.

For memory, our goal is to limit the total memory size of VM_1 and IDS_1, e.g., to 1 GB. As described above, the hypervisor can allocate the specified sizes of memory to

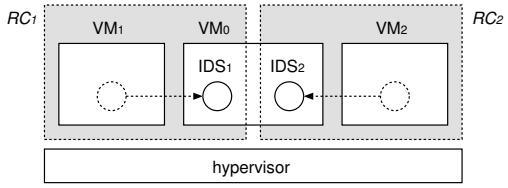


Figure 3: Performance isolation using resource cages.

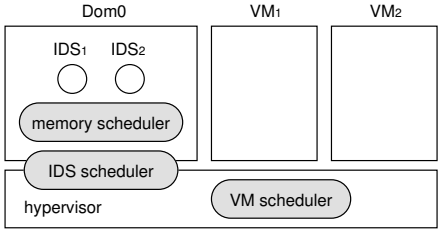


Figure 4: The system architecture in Xen.

VMs. The operating system in the management VM can limit the size of memory used by each process. Therefore, the goal can be accomplished if we allocate 768 MB of memory to VM_1 and limit the memory size of IDS_1 to 256 MB. However, memory utilization can become lower. For example, when IDS_1 uses a small amount of memory, the total memory size of VM_1 and IDS_1 is less than 1 GB. At this time, VM_1 should be able to use the remaining memory of IDS_1, if necessary. In contrast, when IDS_1 needs more memory, it cannot use more than 256 MB of memory even if VM_1 does not need 768 MB of memory. This can cause performance degradation of IDS_1.

3. Resource Cage

In this paper, we propose a new abstraction of the hypervisor, called a *resource cage*, for resource management considering IDS offloading. Traditionally, the hypervisor manages only VMs but no processes in them because processes are managed by the operating systems in them. A resource cage manages a VM and IDS processes offloaded from it as a group. The hypervisor assigns CPUs and memory to resource cages, not VMs. Note that the hypervisor does not fully manage IDS processes included in resource cages. It leverages the existing mechanisms for resource management of the operating system as much as possible. The hypervisor just monitors the processes, while the operating system controls them.

A resource cage achieves both performance isolation and high resource utilization for a group of a VM and offloaded IDSes. Let us consider two resource cages for the example in Section 2.2: RC_1 for IDS_1 and VM_1 and RC_2 for IDS_2 and VM_2, as shown in Fig. 3. For example, we can limit the CPU utilization of RC_1 and that of RC_2 to 50%, respectively. When IDS_1 is idle, VM_1 in the same resource cage can receive up to 50% of the CPU time. In contrast, when VM_1 is idle, IDS_1 in RC_1 can receive up to 50% of the CPU time. Even when both IDS_1 and VM_1 are idle, however, RC_2 cannot receive more than 50% of the CPU time.

As another example, we can assign CPU shares to RC_1 and RC_2 in a 1:1 ratio. When either IDS_1 or VM_1 is idle, the other can receive all the CPU time allocated to RC_1 and the CPU allocation to RC_1 and RC_2 is kept to 1:1. Unlike the above example, when both IDS_1 and VM_1 are idle, the surplus CPU time is allocated to RC_2 because of the work-conserving nature of proportional share scheduling.

Also, we can limit the memory of RC_1, e.g., to 1 GB. When IDS_1 uses only a small amount of memory, VM_1 can use the rest of the memory assigned to RC_1. When IDS_1 needs a large amount of memory, the memory allocation to VM_1 is decreased and the decrement can be used by IDS_1. Even when both IDS_1 and VM_1 do not use a small amount of memory, RC_2 cannot use memory that exceeds its upper limit.

Resource cages are created hierarchically. The hypervisor automatically assigns a VM to a resource cage RC_{vm} when the VM is created. In contrast, system administrators manually assign an IDS process to a resource cage RC_{ids} because the hypervisor does not know which process in the management VM is an IDS. When multiple IDSes are offloaded from one VM, they can be assigned to the same resource cage. Then, administrators create a collective resource cage RC by specifying RC_{vm} and RC_{ids}. If any IDSes are not offloaded from a VM, RC consists of only RC_{vm}. For RC_{vm}, RC_{ids}, and RC, administrators can set CPU limits, CPU shares, and memory limits.

4. Implementation

We have implemented resource cages in Xen 4.1 [3]. The resource management using resource cages is achieved by a VM scheduler, an IDS scheduler, and a memory scheduler. In Xen, the management VM is called Dom0 and a regular VM is called DomU. The system architecture is shown in Fig. 4. Also, we explain the implementation of resource cages in KVM 1.1.2 [4].

4.1. Attributes of Resource Cages

Resource cages have attributes as shown in Table 2. Cap, $shares$, and $mcap$ are for resource controls. Cap is used for setting a CPU limit to a resource cage. The default value is zero, which means no CPU limit. $Shares$ is used for setting CPU shares between resource cages. The default value is 256. $RC_{vm}.shares$ and $RC_{ids}.shares$ in the same RC can set CPU shares between a VM and a group of IDS processes, but they are not implemented currently. $Mcap$ is used for setting a memory limit. The default value of $RC.mcap$ is the same as that of $RC_{vm}.mcap$, which is automatically set to the maximum memory size configured for a VM. However, the default value of $RC_{ids}.mcap$ is the half of it because the operating system in a VM can crash if only a too small amount of memory is available.

In contrast, cpu, $cpuav$, and mem are for resource monitoring and specific to RC_{ids}. Cpu maintains instantaneous

TABLE 2: The attributes of resource cages.

type	attribute	purpose
common	cap	CPU limit (%)
	shares	CPU shares
	mcap	memory limit (MB)
RC_{ids} specific	cpu	instantaneous CPU utilization (%)
	cpuav	average CPU utilization (%)
	mem	consumed memory size (MB)

CPU utilization used by offloaded IDSes and $cpuav$ maintains the average CPU utilization using the modified moving average. Mem maintains the size of memory consumed by offloaded IDSes.

4.2. VM Scheduler

By default, Xen uses the credit scheduler, which is a proportional share CPU scheduler. In the credit scheduler, each VM is assigned a weight and a cap. A weight is used for CPU shares, while a cap is for a CPU limit. According to a weight, the scheduler calculates *credits* every 30 ms and distributes them to active virtual CPUs (vCPUs) assigned to a VM. At that time, the distributed credits are restricted by a cap. On the basis of credits, the scheduler schedules vCPUs for physical CPUs (pCPUs) using run queues. It first picks a vCPU with a high priority from a run queue. If there are no such vCPUs, it picks one with a low priority. Once a vCPU is scheduled, it receives the time slice of 30 ms and consumes its credits every 10 ms.

Our VM scheduler is based on the credit scheduler, but it calculates credits C_{cap} from $RC.cap$ and $RC_{ids}.cpu$ as follows:

$$C_{cap} = C_{ts} \times \left(\frac{RC.cap}{100} - \frac{RC_{ids}.cpu}{100} \right)$$

where C_{ts} is the credits distributed per time slice. This means that the CPU limit of the resource cage is decreased temporarily by the CPU time consumed by offloaded IDSes. In addition, the VM scheduler calculates credits C'_{cap} from $RC_{vm}.cap$ as follows:

$$C'_{cap} = C_{ts} \times \frac{RC_{vm}.cap}{100}$$

C'_{cap} is the credits that the VM can receive at maximum. If $RC.cap$ or $RC_{vm}.cap$ is zero, the maximum CPU allocation to the VM, e.g., 200% for two active vCPUs, is used instead.

Similarly, the VM scheduler calculates credits C_w from $RC.shares$ and $RC_{ids}.cpu$ as follows:

$$C_w = C_{tot} \times \frac{RC.shares}{S_{tot}} \times VM_{cpu} - C_{ts} \times \frac{RC_{ids}.cpu}{100}$$

where C_{tot} is the product of C_{ts} and the total number of vCPUs in the system, S_{tot} is the sum of the CPU shares assigned to all the VMs, and VM_{cpu} is the number of vCPUs assigned to a target VM. This means that the CPU shares of the resource cage are also decreased temporarily by the CPU time consumed by offloaded IDSes.

Finally, the VM scheduler distributes credits of the minimum value among C_{cap}, C'_{cap}, and C_w to the vCPUs of a target VM. The credits decreased by the value of $RC_{ids}.cpu$ are distributed to Dom0.

4.3. IDS Scheduler

Our IDS scheduler monitors the CPU utilization of IDS processes running in Dom0 from the hypervisor. To record the CPU time consumed by each IDS process, the IDS scheduler measures the execution time of a process by monitoring the switches between virtual address spaces. Since a process has one virtual address space, the hypervisor can identify a process by a virtual address space, as proposed in [5]. A virtual address space is uniquely identified by the address of the page directory used by a process. When the operating system in Dom0 sets the address to the CR3 register in a vCPU, the hypervisor is invoked because the instruction for changing CR3 is privileged. At that time, the IDS scheduler accumulates the CPU time from when the target address is set to CR3 until the value of CR3 is changed as the execution time of the corresponding process.

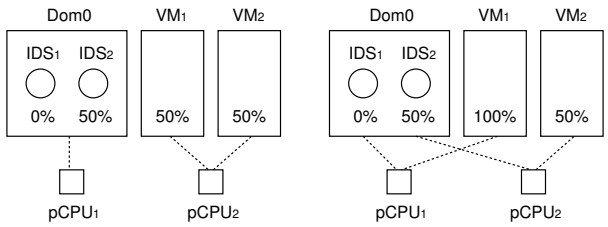
In the current implementation, Dom0 notifies the hypervisor of the address of the page directory of an IDS process by using a hypercall. In the Linux kernel, the address is stored in the mm_struct structure, which is followed from task_struct. This requires modifying the operating system in Dom0, but it is acceptable because Dom0 is managed by IaaS providers.

According to the monitored CPU utilization of IDS processes, the IDS scheduler schedules the processes so that they do not consume more CPU time than configured. Every 100 ms, it re-calculates $RC_{ids}.cpuav$ and then calculates the runnable time T_{run} of an IDS process from $RC_{ids}.cap$ and $RC_{ids}.cpuav$ as follows:

$$T_{run} = \min \left(100 \times \frac{RC_{ids}.cap}{RC_{ids}.cpuav}, 100 \right)$$

If the average CPU utilization of an IDS process exceeds the limit, the runnable time is decreased from 100 ms. Otherwise, it is increased. If $RC_{ids}.cap$ is zero, T_{run} is always 100 ms. The remaining time is the waiting time. The IDS scheduler runs an IDS process during the runnable time, while it stops the process during the waiting time. It repeats this for each IDS process included in RC_{ids} in turn.

Currently, this scheduling part of the IDS scheduler is based on cpulimit [6] and runs in Dom0. It stops a process using the SIGSTOP signal and restarts it using the SIGCONT signal. This could be also implemented using the CPU bandwidth controller in Linux control groups (cgroups), which was introduced in Linux 3.2. To implement this part in the hypervisor, we can use the Monarch scheduler [7]. The Monarch scheduler can indirectly suspend and resume processes by manipulating run queues in the operating systems from the hypervisor.



(a) Not shared with Dom0 (b) Shared with Dom0

Figure 5: The assignment of multiple pCPUs.

4.4. Consideration in Multicore Support

For performance isolation, resource cages work well for not only single core but also multicore. However, for high resource utilization, we need care about the assignment of pCPUs to VMs. Let us consider the example of Fig. 5(a). In this example, pCPU_1 is assigned to Dom0 and pCPU_2 is assigned to both VM_1 and VM_2. Resource cage RC_1 consists of VM_1 and IDS_1, whereas RC_2 consists of VM_2 and IDS_2. Assume that the CPU limits of RC_1 and RC_2 are 100% and those of IDS_1 and IDS_2 are 50%, respectively. If IDS_1 becomes idle, VM_1 should receive the CPU time for it. However, the surplus CPU time of pCPU_1 cannot be used by VM_1 because the pCPU is not shared with that VM. Therefore, RC_1 for IDS_1 and VM_1 can receive only 50% from pCPU_2.

To improve resource utilization, pCPUs assigned to a VM have to be shared with Dom0. In Fig. 5(b), pCPU_1 is shared between VM_1 and Dom0, while pCPU_2 is shared between VM_2 and Dom0. Even if IDS_1 becomes idle, VM_1 can receive the CPU time for it. In addition, the IDS scheduler needs to assign pCPU_2 to IDS_2 by setting CPU affinity. This is because the CPU utilization of RC_1 decreases to 50% if IDS_2 receives 50% of the CPU time from pCPU_1, not pCPU_2, and VM_1 receives only 50% of the CPU time from pCPU_1.

4.5. Memory Scheduler

An IDS mainly uses two types of memory: process memory and the page cache. Process memory is the physical memory consumed by a process and the size is called the resident set size (RSS). The page cache is the cache created in the kernel when a process reads a file from a disk or writes data to a file. Although the page cache is not the memory belonging to a process but that of the kernel, it should be considered as the memory consumed by an IDS. For example, an IDS examining disks reads many files and creates a large amount of page cache in the kernel.

Our memory scheduler monitors the memory usage using memory cgroups introduced in Linux 2.6.25. A memory cgroup consists of offloaded IDS processes and the memory usage is accounted for in the cgroup. The memory scheduler can obtain the total of the RSS of IDS processes and the size of the page cache created by them from `memory.usage_in_bytes` in a cgroup. The memory size consumed by IDSes is recorded in $RC_{ids}.mem$. The memory scheduler periodically calculates a new memory size M_{new} of a VM from $RC_{ids}.mem$ and $RC_{vm}.mcap$ as follows:

$$M_{new} = \min(RC.mcap - RC_{ids}.mem,\ RC_{vm}.mcap)$$

According to the calculated size, the memory scheduler changes the memory allocation to the VM using libvirt [8]. Libvirt is a library for managing VMs without depending virtualization software. When it issues a hypercall for changing the memory size of a VM, the hypervisor sends a request to the memory balloon driver [9], which is a device driver running in the VM. When the driver receives a request for decreasing the memory size, it allocates a necessary amount of memory using the memory allocation mechanism in the operating system and turns the state of the memory unavailable. Then, it returns the allocated memory to the hypervisor, so that physical memory available in the VM decreases. In contrast, when the driver receives a request for increasing the memory size, it turns the state of a requested amount of memory available again.

The memory scheduler limits the memory usage of IDS processes by `memory.limit_in_bytes` in a cgroup. A memory cgroup can limit the maximum of the total size of the process memory of IDSes and the page cache used by them.

4.6. Resource Cages in KVM

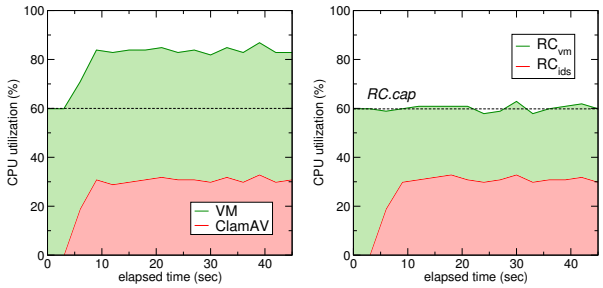
We have also implemented resource cages in KVM, which has an architecture different from Xen. Xen runs the hypervisor directly on hardware and runs VMs on top of the hypervisor. In contrast, KVM runs the hypervisor as a Linux kernel module and QEMU-KVM processes on top of the host operating system and runs a VM as part of a QEMU-KVM process. Therefore, the implementation of resource cages is straightforward in KVM because a VM in KVM is managed as a process. We could easily implement resource cages using cgroups in Linux. RC_{ids}, RC_{vm}, and RC are created by the cgroup of IDS processes, that of a process for a VM, and that for grouping two cgroups, respectively. Using cgroups, the host operating system can naturally schedule a group of IDS processes and a VM with CPU limits and shares. It can also limit the memory size of the cgroups.

5. Experiments

We conducted experiments to confirm performance isolation in IDS offloading. In the experiments, we created three resource cages: RC_{ids} for an offloaded IDS, RC_{vm} for its target VM, and RC for grouping RC_{ids} and RC_{vm}.

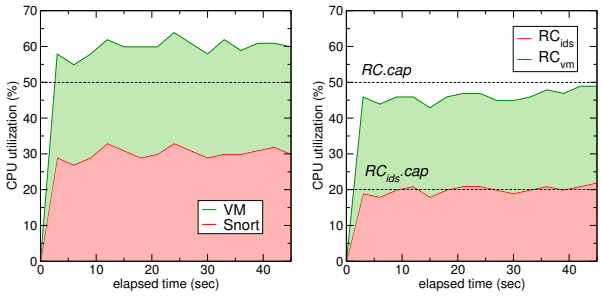
5.1. Limiting CPUs

In this experiment, we used a PC with one Intel Core i7 2.8 GHz processor, 8 GB of memory, 1 TB of SATA HDD,



(a) No resource cages (b) Resource cages

Figure 6: The CPU utilization in offloading ClamAV.



(a) No resource cages (b) Resource cages

Figure 7: The CPU utilization in offloading Snort.

and gigabit Ethernet. We ran Xen 4.1.0 and Linux 2.6.32 in Dom0. We created a VM with one vCPU and ran Linux 2.6.16 in the VM. For a client, we used a PC with Intel Core i7 2.67 GHz, 12 GB of memory, and gigabit Ethernet.

Offloading ClamAV. We offloaded ClamAV [10], which is a host-based IDS for detecting viruses that infect files. We achieved the offload of ClamAV by monitoring a virtual disk of a target VM in Dom0. First, we measured the CPU utilization of the offloaded ClamAV and its target VM when we did not use the resource cages. We ran a CPU-intensive task in the VM and limited the CPU utilization of the VM to 60%. Our goal was to suppress the total CPU utilization to 60%. Fig. 6(a) shows the changes of the CPU utilization of ClamAV and the VM. While ClamAV was not running during the first five seconds, the VM consumed 60% of the CPU time, as we configured. When ClamAV started running, however, it consumed 30% of the CPU time. As a result, the total CPU utilization exceeded 60%.

Next, we used the resource cages and limited the CPU utilization to 60% for RC. We did not configure the CPU limit for RC_{vm} or RC_{ids}. As shown in Fig. 6(b), the CPU time assigned to RC_{vm} was decreased when ClamAV started running and the CPU utilization of RC was always kept to 60% successfully.

Offloading Snort. We offloaded Snort [11], which is a network-based IDS for checking network packets. We achieved the offload of Snort by monitoring a virtual network interface of a target VM in Dom0. First, we measured the CPU utilization of the offloaded Snort and its target VM without the resource cages. We ran the Apache web server in the VM and sent 4000 requests per second using httperf [12] from the client. We limited the CPU utilization of the VM to 50%. Our goal was to suppress the total CPU utilization to 50%. As shown in Fig. 7(a), the VM and Snort consumed approximately 30% of the CPU time, respectively, and then the total CPU utilization exceeded 50%.

Next, we limited the CPU utilization to 50% for RC and 20% for RC_{ids}. We did not limit that for RC_{vm}. Fig. 7(b) shows that the CPU utilization of RC did not exceed its CPU limit of 50%. It was 45% at first and 5% lower than the upper limit, but it increased to 50% gradually. For offloaded IDSes, the CPU utilization of RC_{ids} was always suppressed to 20% as configured.

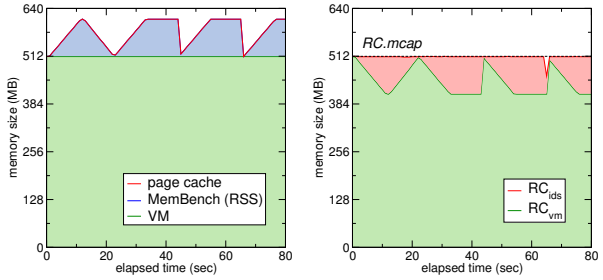
In addition, we measured the throughput of the web server in the VM (1) when we did not offload Snort, (2) when we offloaded Snort but did not use the resource cages, and (3) when we used the resource cages for the offloaded Snort and the VM. When we simply offloaded Snort out of the VM, the web server in the VM could use more CPU time and therefore the throughput increased by 15%. With the resource cages, the throughput was almost the same as that before the offload because the web server could use almost the same CPU time as when Snort ran in the VM.

5.2. Limiting Memory

In this experiment, we used a PC with one Intel Xeon X5675 3.06 GHz processor, 16 GB of memory, and 146 GB of SAS HDD. We ran Xen 4.1.3 and created a VM with one vCPU and 512 MB of memory. We allocated the rest of the memory to Dom0 and ran Linux 3.2.0 in Dom0 and the VM.

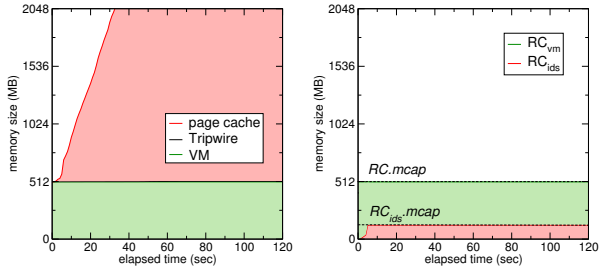
Offloading MemBench. We offloaded MemBench, which repeatedly allocated 100 MB of memory by malloc, wrote data in it, and deallocated it at approximately 10 MB/s. Our goal was to suppress the total memory size consumed by the VM and the offloaded MemBench to 512 MB. Fig. 8(a) shows the changes of the measured memory usage when we did not use the resource cages. MemBench consumed 100 MB of process memory at maximum, but used little page cache. Even when MemBench allocated memory, the memory size of the VM was not reduced. Therefore, the total memory size exceeded 512 MB.

Next, we used the resource cages and limited the memory size of RC to 512 MB. As shown in Fig. 8(b), the memory allocated to the VM was decreased as the MemBench allocated more process memory. In contrast, as MemBench deallocated the memory, the memory size of the VM was increased. As a result, the total memory size consumed by MemBench and the VM was kept to 512 MB. At the 65 second, the total size was smaller because it took time to reallocate the memory to the VM due to the memory ballooning mechanism.



(a) No resource cages (b) Resource cages

Figure 8: The memory usage in offloading MemBench.



(a) No resource cages (b) Resource cages

Figure 9: The memory usage in offloading Tripwire.

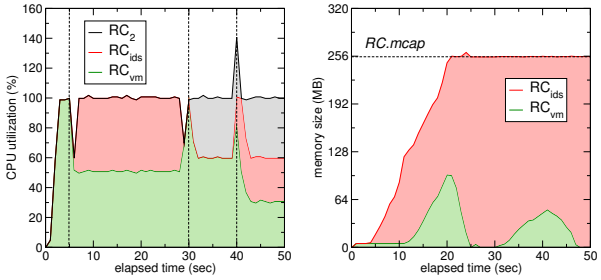
Offloading Tripwire. We offloaded Tripwire [13], which is a host-based IDS for checking the integrity of disks. First, we ran the offloaded Tripwire without the resource cages. The result is shown in Fig. 9(a). The offloaded Tripwire used only 130 KB of process memory at maximum, but it used a large amount of page cache because it read many files from the disk to check their contents. Without limitation, the page cache consumed by Tripwire became more than 3.5 GB in 120 seconds. In total, the memory size of the VM and Tripwire largely exceeded 512 MB.

Next, we used the resource cages and limited the memory size of RC to 512 MB. In addition, we limited the memory size of RC_{ids} for Tripwire to 128 MB. As shown in Fig. 9(b), The total memory size of Tripwire and the VM was kept to 512 MB. As configured, Tripwire did not use more than 128 MB.

5.3. Resource Control in KVM

To confirm that resource cages for KVM also work well, we offloaded Tripwire onto the host operating system. In this experiment, we used a PC with an Intel Xeon E5630 processor, 6 GB of memory, and 250 GB of SATA HDD. We ran QEMU-KVM 1.1.2 and Linux 3.2.0 as the host operating system. We created a VM with one vCPU and 512 MB of memory and ran Linux 2.6.27 in the VM.

First, we examined that resource cages enable controlling CPU utilization. We ran a CPU-intensive task in the VM and assigned CPU shares to RC_{vm} and RC_{ids} in a 1:1 ratio. In addition, we ran a CPU-intensive task in another VM and created a resource cage RC_2 for it. We assigned



(a) CPU utilization (b) Memory usage

Figure 10: Resource usage in KVM with resource cages.

CPU shares to RC and RC_2 in a 3:2 ratio. We did not set the CPU limits to any resource cages.

Fig. 10(a) shows the results. Since the offloaded Tripwire in RC_{ids} or the CPU-intensive task in RC_2 did not run during the first five seconds, RC_{vm} could receive the entire CPU time. When the offloaded Tripwire started running, RC_{ids} and RC_{vm} received the CPU time in a 1:1 ratio. After the CPU-intensive task in RC_2 started running and the offloaded Tripwire in RC_{ids} stopped at the 30 second, RC and RC_2 received the CPU time in a 3:2 ratio as configured. Since all the VMs and the IDS were busy after the 40 second, RC_{ids}, RC_{vm}, and RC_2 received the CPU time in a 3:3:4 ratio. The reason why the total CPU utilization was 140% at the 40 second is the time lag in CPU statistics.

Next, we measured the memory usage of RC_{ids} and RC_{vm} when we ran MemBench inside the VM. We limited the memory of RC to 256 MB and did not limit that of the other resource cages. As shown in Fig. 10(b), the real memory allocation to the VM was very small at first. Unlike Xen, a VM consumes only a small amount of memory in KVM if there are no active applications. After the offloaded Tripwire started running, it created more page cache, but the memory size of RC was less than 256 MB. After the total memory size reached 256 MB, old page cache was replaced with new one. The memory of RC_{vm} was paged out due to the memory pressure by Tripwire.

6. Related Work

SEDF-DC [14] is a VM scheduler for enforcing performance isolation between VMs considering I/O processing. In Xen, a device driver is split into a backend driver in Dom0 and a frontend driver in DomU. Therefore, the CPU time consumed by a backend driver is not accounted for DomU. SEDF-DC measures the CPU time consumed for DomU in Dom0 and schedules DomU considering it. In addition, the ShareGuard mechanism limits the CPU utilization of I/O processing for DomU in Dom0. This is similar to resource cages in that SEDF-DC creates a group of DomU and its backend driver in Dom0. The differences are that SEDF-DC provides no new abstraction like resource cages and that it is specific to network I/O processing. It estimates the CPU utilization from the number of packets and limits the CPU utilization using a packet filter.

LRP [15] enables accounting the CPU time consumed for network processing for the corresponding process. When a process uses networks frequently, most of its CPU time is consumed in the kernel, but that is not accounted for the process appropriately. In LRP, network processing in the kernel is performed in the context of the corresponding process and its CPU time is accounted for each process. Resource containers [16] extend LRP and introduce new resource management into the kernel. They manage resources such as CPUs and memory by a unit different from a process. The process scheduler schedules each process using information on the resource usage recorded in resource containers that the process belongs to. Unlike LRP and resource containers, resource cages are a new abstraction of the hypervisor.

Resource pools in VMware vSphere [17] enable resource management for a group of VMs. A resource pool is assigned to multiple VMs and performance isolation is achieved between resource pools. Resources such as CPUs and memory are shared between VMs in a resource pool. Group-based memory deduplication [18] has been proposed to deduplicate the memory of VMs only in such a group. Like resource cages, resource pools can be grouped into hierarchies and a parent resource pool can contain child resource pools. However, since a VM is a minimum unit in a resource pool, a resource pool cannot group VMs and processes.

In combination with a guard VM proposed in VMCoupler [19], resource pools would be useful for performance isolation considering IDS offloading. VMCoupler is a system enabling co-migration of offloaded IDSes and their target VM. It runs offloaded IDSes in a special VM called a guard VM, which can introspect the memory, storage, and network in its target VM. A resource pool can group a guard VM and its target VM and control their resource usage. However, many guard VMs are necessary for monitoring many target VMs because one guard VM can monitor only one target VM. Compared with a simple IDS process, a guard VM including the operating system increases resource consumption.

7. Conclusion

In this paper, we proposed a resource cage, which is a new abstraction of the hypervisor for resource management considering IDS offloading. A resource cage manages a VM and offloaded IDS processes as a group and achieves performance isolation such as CPU limits, CPU shares, and memory limits. Also, it can keep high resource utilization for a VM and offloaded IDSes. We have implemented resource cages in Xen and KVM, which have largely different architectures. We conducted experiments using offloaded Snort, ClamAV, MemBench, and Tripwire. The experimental results showed that resource cages could control resource usages under IDS offloading effectively.

Our future work is to implement CPU shares between a VM and IDS processes inside the same resource cage in Xen. Since a VM and a process are managed by the hypervisor and the operating system, respectively, in Xen, it is difficult to relatively control their resource usages. In addition, we are planning to control the usage of storage and network in resource cages. Since Dom0 handles storage and network accesses of VMs in Xen, the operating system in Dom0 would be suitable for controlling the I/O of both VMs and offloaded IDS processes. Another direction is to apply resource cages to processes other than offloaded IDSes, e.g., device emulators for VMs.

Acknowledgment

This work was partially supported by JSPS KAKENHI Grant Number JP16K00101.

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] B. Verghese, A. Gupta, and M. Rosenblum, "Performance Isolation: Sharing and Isolation in Shared-memory Multiprocessors," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 181–192.

[3] 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.

[4] Red Hat, Inc., "Kernel Based Virtual Machine," http://www.linux-kvm.org/.

[5] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking Processes in a Virtual Machine Environment," in *Proc. USENIX Annual Technical Conf.*, 2006.

[6] A. Marletta, "CPU Usage Limiter for Linux," https://github.com/opsengine/cpulimit.

[7] H. Tadokoro, K. Kourai, and S. Chiba, "A Secure System-wide Process Scheduler across Virtual Machines," in *Proc. Pacific Rim Int. Symp. Dependable Computing*, 2010, pp. 27–36.

[8] Red Hat, Inc., "libvirt: The Virtualization API," http://libvirt.org/.

[9] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proc. Symp. Operating Systems Design and Implementation*, 2002.

[10] Sourcefire, Inc., "Clam AntiVirus," http://www.clamav.net/.

[11] M. Roesch, "Snort – Lightweight Intrusion Detection for Networks," in *Proc. USENIX System Administration Conf.*, 1999.

[12] D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[13] G. Kim and E. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker," in *Proc. Conf. Computer and Communications Security*, 1994, pp. 18–29.

[14] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing Performance Isolation across Virtual Machines in Xen," in *Proc. Int. Conf. Middleware*, 2006, pp. 342– 362.

[15] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," in *Proc. Symp. Operating Systems Design and Implementation*, 1996, pp. 261– 275.

[16] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proc. Symp. Operating Systems Design and Implementation*, 1999, pp. 45–58.

[17] VMware Inc., "Server Virtualization & Cloud Infrastructure: VMware vSphere," http://www.vmware.com/products/vsphere.

[18] K. Sangwook, K. Hwanju, and L. Joonwon, "Group-Based Memory Deduplication for Virtualized Clouds," in *Proc. Int. Conf. Parallel Processing – Volume 2*, 2011, pp. 387–397.

[19] K. Kourai and H. Utsunomiya, "Synchronized Co-migration of Virtual Machines for IDS Offloading in Clouds," in *Proc. Int. Conf. Cloud Computing Technology and Science*, 2013, pp. 120–129.