International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 6, Number 1, pages 124–147, January 2016

Unified Performance Profiling of an Entire Virtualized Environment

Masao Yamamoto, Miyuki Ono, Kohta Nakashima
Computer Systems Laboratory, Fujitsu Laboratories Ltd.,
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa, 211-8588, Japan

Akira Hirai
Software Laboratory, Fujitsu Laboratories Ltd.,
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa, 211-8588, Japan

Received: February 15, 2015
Revised: May 6, 2015
Revised: July 2, 2015
Accepted: August 12, 2015
Communicated by Hiroyuki Sato

Abstract

Performance analysis and troubleshooting of cloud applications are challenging. In particular, identifying the root causes of performance problems is quite difficult. This is because profiling tools based on processor performance counters do not yet work well for an entire virtualized environment, which is the underlying infrastructure in cloud computing. In this work, we explore an approach for unified performance profiling of an entire virtualized environment by sampling only at the virtual machine monitor (VMM) level and applying common-time-based analysis across the entire virtualized environment from a VMM to all guests on a host machine. Our approach involves three parts, each with novel techniques: centralized data sampling at the VMM-level, generation of symbol maps for programs running in both guests and a VMM, and unified analysis of the entire virtualized environment with common time by the host-time-axis. We also describe the design of unified profiling for an entire virtual machine (VM) environment, and we actually implement a unified VM profiler based on hardware performance counters. Finally, our results demonstrate accurate profiling. In addition, we achieved a lower overhead than in a previous study as a result of having no additional context switches by the virtual interrupt injections into the guests during measurement.

Keywords: Performance, Profiling, Measurement, Analysis, Virtual machine, Cloud computing

1 Introduction

Performance analysis and troubleshooting of cloud applications are challenging, but these measures provide opportunities to reduce costs by improving the efficiency of resource utilization. Service-level performance management tools for clouds have been enhanced for practical use. Accordingly, these tools now allow system administrators to quickly confirm performance problems and resource-level bottlenecks even in complex virtualized environments. However, no practical system-wide tools for virtualized environments, such as profilers, provide the detailed information that would be required

to know why such problems occur. Currently used solutions to such problems include virtual machine (VM) migration and reallocation of resources. However, such methods may allow the return of the same problems and ineffective use of resources, and they can create potentially higher operational management costs. Moreover, performance-critical applications, such as high performance computing (HPC) and mission-critical (MC) systems, have become increasingly virtualized in recent times. Therefore, program-level performance optimization for applications in virtualized environments has become a requirement. Thus, a practical and useful profiler for VMs is required for general system engineers and administrators, in addition to OS/virtual machine monitor (VMM) developers and researchers. However, in virtualized environments, the practical use of profiling tools based on processor performance counters has not advanced significantly, unlike the case in native environments.

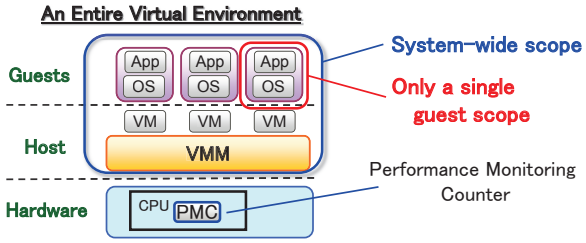


Figure 1: Profiling scopes in an entire viratulized environment.

Figure 1 indicates all scopes in an entire virtualized environment. System-wide profiling is necessary for understanding accurate characteristics of VM performance and for resolving the root causes of VM performance problems. Even in guests, standard profilers have become available using the virtualized performance counters by VMM in recent times. However, there are cases where their profiling results are insufficient to analyze guest programs. This is because standard profilers in guests cannot have a system-wide scope that includes all guests, all VMs, and an underlying VMM in an entire virtualized environment. Standard profilers in guests have only a single guest scope. It is difficult to grasp accurate VM behavior only by an analysis of a single guest without system-wide analysis. This is because a virtualized environment is very complex due to the inclusion of multiple guest-OSes, VMs, and a VMM on a host. For example, in some cases, performance problems in guests are caused by other guests or a VMM. Moreover, guest-inside profiling cannot provide a grasp of steal time. Each guest has a steal time in which it was unable to run because a physical CPU could not be assigned to a guest. Hence, System-wide VM profiling is necessary. According to previous studies, the main issue with system-wide VM profiling is that profiling must be distributed to each VM [19]. Therefore, additional context switches are inevitable for conventional methods [8].

The goal of this paper is to present an approach that should help to overcome the difficulties in conventional VM profiling. We describe a method in which an entire virtualized environment can be profiled, from the VMM layer to the guest-inside. This is unified profiling, which is realized using three novel techniques. The first is centralized data sampling at the VMM level. Execution information to identify programs running in both guests and a VMM is centrally sampled at the VMM level only. The second is generation and collection of symbol map information of programs executed in both guests and a VMM; these operations are performed in each guest and in a VMM only once after the host-level centralized sampling is concluded. As a result, profiling delegation [8, 7] in guests during measurement is no longer required, even in system-wide profiling. In addition, this second technique makes it possible to convert guest sampling data into meaningful symbol names with map information. The third technique is unified analysis of the entire virtualized environment, including guest applications, with the host time as the common base time. This technique helps to provide an understanding of the correct behavior of applications executed in guests. One challenge in achieving success is the determination of a method for converting sampled data into meaningful symbol names of the guest applications by these three techniques, without additional profiling overhead. The main contributions of this paper are as follows:

- This paper proposes a novel unified performance profiling of an entire virtualized environment by sampling only at the VMM level and applying common-time-based analysis across the entire virtualized environment from a VMM to all guests on a host machine.
- This paper presents quantitative evaluations of the proposed method using real applications.
- This paper reveals steal time even in case of no oversubscription of CPU. Such steal time is unknown by existing standard OS tools.

The remainder of this paper is organized as follows: Section 2 introduces conventional work on VM profiling and clarifies unresolved points. Section 3 details the proposed new method for unified VM profiling. Section 4 provides evaluation results for the accuracy and overhead of the new method and presents a case study of performance tuning using the unified VM profiling. Finally, Section 5 concludes this paper with future directions.

2 Conventional Performance Profiling

2.1 For Native Environments

Performance profilers have been widely used in native environments to optimize program performance and diagnose the causes of performance degradation. For example, the profiler collects execution information of a running program each time a specified number of performance event counts has been passed. Then, the profiler analyzes the collected data and indicates the frequency at which various programs or functions are being executed by displaying CPU utilization based on running programs. These statistical profile results can be used to detect program hotspots and determine the cause of performance degradation. Thus, a profiler is an indispensable statistical analysis tool for performance tuning and debugging.

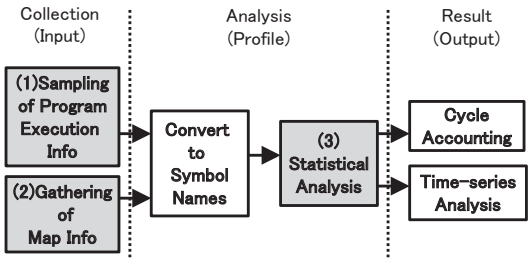


Figure 2: Profiling function blocks and processing flow.

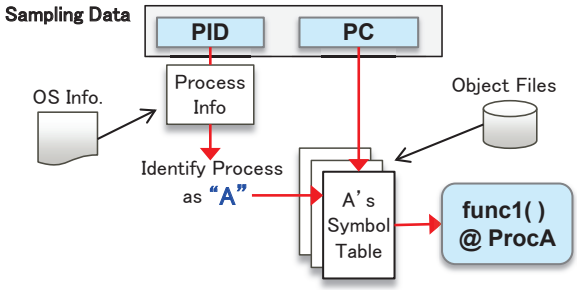


Figure 3: Converting sampling data into symbol names with map info.

Figure 2 illustrates the function blocks and processing flow of profiling. Generally, this process consists of three parts. The first is collection, which becomes the input for the next analysis; the second is analysis, which is the profile analysis itself; and the last is the result, which is the output of the analysis. The collection part consists of two blocks: (1) an execution information-sampling block and (2) a map information-gathering block. Block (3), the statistical analysis block, is in the analysis part. These shaded blocks (1), (2), and (3) are our approach targets.

In the sampling measurement block (1), the running program is interrupted at fixed intervals, and execution information of the interrupted program is sampled. Hardware performance counters trigger the sampling through the counter overflow interrupt after a specified number of event counts. For example, once a profiler sets a configuration of sampling interrupt that is generated after the occurrence of 10,000 "CPU Unhalted Cycles" events, interrupt occurs after 10,000 CPU Cycles. At every interrupt, the profiler captures a process ID (PID) and a program counter (PC) of the

interrupted program as execution information. Finally, PIDs and PCs are sampled periodically through the performance counter overflow interrupt.

Sampling data consisting of a PID and a PC should be converted into a process name and a function name that are meaningful symbol names. For this purpose, the process information and object files must be gathered. A PID can be used to identify the execution process, and a PC value can be used to determine the function name, as shown in Figure 3.

2.2 For Virtualized Environments

Ten years have passed since Xenoprof [19] was introduced, which was the first typical profiler for virtualized environments. However, the practical use of performance counter-based VM profilers has not advanced significantly, unlike the case in native environments. Although each conventional method for VM profiling has advantages, they share a common disadvantage in that the overhead by additional context switches between the VMM and the guest, as shown in Figure 4. The additional context switches are required due to virtual interrupt injections for guest-inside sampling or for interpreting guest data. In this section, first, we analyze the problems with the use of conventional VM profiling methods from the perspective of profiling at both the guest level and the VMM level. Then, we also explain the points to be resolved as part of our challenge to the practical use of performance counter-based VM profiler.

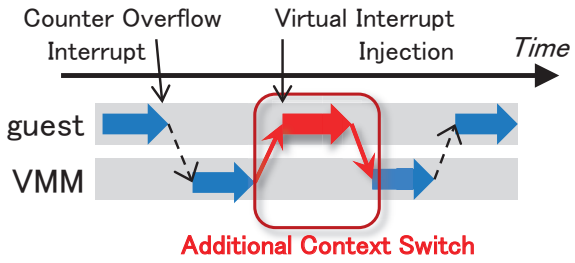


Figure 4: Additional overhead in conventional methods.

Profiling at guest level with virtualized performance counters: Standard profilers (e.g., Intel VTune [6], Oprof [17], and PAPI [22]) based on hardware performance counters have become able to work even in guest environments. Previously, unmodified native profilers were not available in guests because performance counters were not exposed to guests by most VMMs. However, recently, some VMMs provide a function of virtualized performance counters. This function virtualizes performance counters and time-shares them among guests. Specifically, a VMM executes the context switch of hardware performance counters and adjusts some of the counter results to match the expectations of the profiler [24]. VMware has already provided this capability since version ESXi5.1 [13]. Xen is being enhanced to support this function [21]. Although Kernel-based Virtual Machine (KVM) [15] and Linux perf [23] use a similar approach, standard profilers have not been available in KVM guests thus far [12, 16]. The development version of PAPI, even in KVM guests, may use performance counters except for the profiling use. The reason PAPI cannot profile in guests is because counter overflow may not be implemented for PAPI [14]. Thus, in some guest environments, guest-inside profiling has become available by standard profilers using the virtualized performance counters.

In contrast, it is still difficult to grasp accurate VM behavior only by guest-inside profiling without system-wide profiling. This is because there are cases where the root causes of problems in guests lie in other guests or a VMM. Moreover, guest-inside profiling cannot provide a grasp of steal time in which a guest was unable to run because a physical CPU could not be assigned to that guest. Currently, Linux and KVM/Xen support reporting steal time. This is a sort of a paravirtualized function that provides information from the VMM on how often a VM was scheduled out. In PAPI 5, this function is used to implement compensation for steal time in the PAPI_get_virt_usec

routine under KVM. However, this function only provides system-wide steal time values. PAPI only returns per-process results. Therefore, it is hard for PAPI to automatically and completely adjust process time measurements in guests. On this problem, Weaver et al. stated that steal time is only an issue if a machine is over-subscribed with VMs, and in most HPC situations only one task is running per node, so this might not be a critical limitation [28]. On the other hand, in more general virtualized environments, this become a critical limitation. In most general cloud-service situations, VMs and virtual CPUs are overcommitted to physical resources. Therefore, the steal time is an issue to be resolved in this work. For this purpose, system-wide profiling is necessary. Among the above standard profilers, only Oprof supports system-wide profiling for a virtualized environment, but only for Xen. That is Xenoprof, a VMM level profiler described below.

Profiling at VMM level with delegation: A host-based method—that is, VMM level profiling—is expected to be an effective solution because it can grasp the entire virtualized environment with integrated analysis. In fact, the results of the VMM-level approaches of Xenoprof [19] and Du et al. [8] have already proved that centralized data sampling at the VMM level and delegating guest profiling into each guest are very useful methods for VM profiling.

However, the delegation method has a problem. The difficulties of realizing VMM-level profiling lie in sampling the execution information of guest applications and interpreting samples corresponding to guests. Xenoprof and Du et al. resolved these problems by delegating the processes that cannot be done at the VMM level to each guest. However, this delegation profiling has more overhead than native profiling because counter overflow interrupts need to be forwarded to the guest by virtual interrupt injections, which therefore add more context switches between the VMM and the guest. Du et al. stated that although profiling should generate as little performance overhead as possible, adding context switches is inevitable for performance counter-based profiling in virtualized environments [8].

In contrast, Linux perf [23] can sample the program counter (PC) and CPU privilege level of the guest from its virtual CPU-state saving area and can profile a guest Linux kernel running in KVM [15] without such a distributed delegation [29]. However, the identifier corresponding to a process in a guest is not sampled. Although samples of the guest Linux kernel can be interpreted, user applications in a guest and other non-Linux guest OSes cannot be profiled. In contrast, Xenoprof [19] and Du et al. [8] solved this problem by using delegation profiling. That is, there is a trade-off between analysis of user applications and the reduction of overhead that comes with additional context switches between the VMM and the guest.

Therefore, in this work, we aimed to solve both problems simultaneously in system-wide profiling.

Challenge: To enable VMM-level profiling to analyze programs by a function from the VMM layer to the OS and user applications running in guests, without additional context switches.

3 Unified Profiling in a virtualized environment

3.1 Hardware Support for x86 Virtualization

Some facilities of a hardware-assisted virtualization are important. This is because our design of unified VM profiling is based on these facilities. In this section, we review the key features of x86 hardware assistance for virtualization. We use these features to implement two techniques, in the aforementioned our three novel techniques, on the x86 platform, which is the most frequently used platform for virtualized environments. In 2005, since Intel Pentium 4 Processor 662/672, Intel introduced their first-generation hardware support for x86 virtualization, known as Intel Virtualization Technology (Intel VT) [4, 5]. Intel VT for x86 is also referred to as VT-x. This first-generation VT-x started to support CPU virtualization, which is the virtualization of the x86 instruction-set architecture. It can eliminate the need for software-based virtualization (e.g., binary translation, paravirtualization). In 2008, since the Nehalem architecture, Intel introduced their second-generation VT-x, which included memory management unit (MMU) virtualization with extended page table (EPT) and virtual processor identifier (VPID). EPT makes it possible to eliminate the need

for software-managed page table virtualization (e.g., shadow page table). We use these hardware-assisted facilities of Intel VT-x in order to realize unified performance profiling in a virtualized environment. Our first technique, that is centralized data sampling, uses the data structure for CPU virtualization introduced in the first-generation VT-x. Furthermore, in our second technique, the generated symbol maps of guest programs becomes available with EPT enabled, which is introduced in the second-generation VT-x.

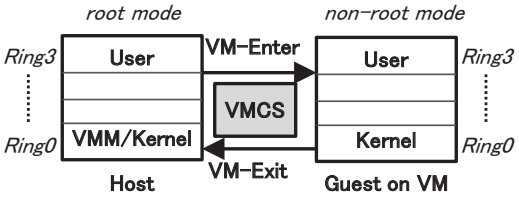


Figure 5: VMX-mode transitions with VMCS.

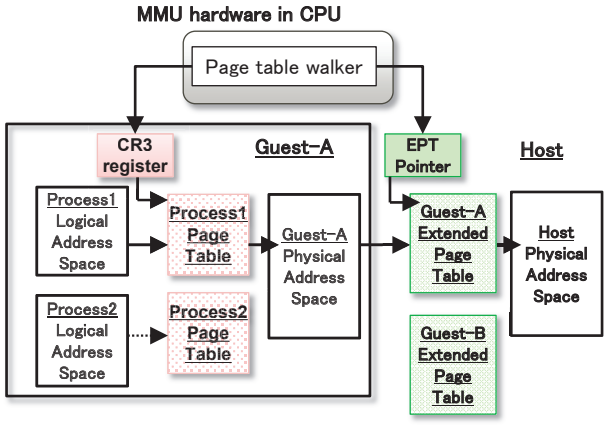


Figure 6: SLAT with EPT on the x86 platform.

Figure 5 shows the hardware-assisted new execution-mode and its mode transitions with a virtual machine control structure (VMCS). The new execution-mode is called virtual machine extensions (VMX) and was introduced with first-generation VT-x. VMX provides two execution modes. One is the root mode in which VMM/Host-kernel is executed. The other is a non-root mode for guest environments. Each mode provides support for all four privilege rings (ring 0 to 3). These two execution modes allow both the VMM and the guest OS to use all four privilege rings by running different modes on the same processor. However, execution of privileged instructions or occurrence of interrupts in non-root mode causes the control to be transferred from non-root mode to root mode. A VM hands over control to a VMM to execute the privileged instruction or the interrupt. This control transfer, from the guest/VM to the host/VMM, is called VM-Exit. Then, once the handling of privileged instruction or interrupt is completed, the VMM returns control back to the VM. This is called VM-Enter. Intel VT helps to accelerate VM-Exit and VM-Enter with the memory structure VMCS. A VMCS is a data structure that holds the processor register states (e.g., PC, control registers) of the guest and the host. These states are saved and restored, automatically by CPU, into the VMCS at VM-Exit/VM-Enter. Furthermore, a VMCS corresponds to a virtual CPU (vCPU) of the VM and can be manipulated by not only hardware but also VMM-level software running in root mode. Therefore, we decided to implement a unified VM profiler as a VMM-level driver.

Figure 6 shows the mechanism of hardware-assisted x86 MMU virtualization, which is commonly called second level address translation (SLAT), with EPT. This additional page table, also called a nested page table, was introduced with second-generation VT-x. Thereby, even in a virtualized environment, the guest OS becomes able to continue to maintain page mappings of each process inside the guest, using the self-managed CR3, as in the case of a native environment, as a pointer to a process page table. In a native environment, the OS uses page tables to translate addresses, from logical addresses to physical ones. On the other hand, in a virtualized environment without hardware assistance for MMU virtualization, the VMM maintains page mappings, from the guest's physical addresses to the host's physical ones, also known as machine addresses, in its internal data structures. The VMM also stores page mappings, from the guest's logical addresses to machine ones, in shadow page tables that are exposed to the hardware for page table walk. Shadow page tables, in other words, map guest-logical pages directly to machine pages. The VMM also needs to keep these shadow page tables synchronized to the guest page tables. To maintain consistency between guest

page tables and shadow page tables, the writes to CR3 by the guest OS—that is, modifications of page tables—are trapped, and the VMM resets CR3 with other values corresponding to shadow page tables. With the introduction of EPT, the VMM can use the hardware support to eliminate the need for shadow page tables on the x86 platform. Hence, the writes to CR3 by the guest OS need not be trapped. The rewrites of CR3 by the VMM are no longer required. Therefore, the profiler can sample the CR3 values set directly by the guest OS.

3.2 Design Overview

In this section, we propose the unified profiling of the entire virtualized environment to tackle the aforementioned challenge, as explained in Section 2.2. The following three novel key techniques are the main points of our proposal. One challenge in achieving success is the determination of a method for converting sampled data into meaningful symbol names of the guest applications by these three techniques, without additional profiling overhead.

Key Technique 1. How to sample data: Centralized data sampling at VMM level
Execution information to identify programs running in both guests and a VMM is centrally sampled at the VMM level only.

Key Technique 2. How to obtain guest apps symbol names: Generating map info of guest apps and interpreting guest sampling data with map info
First, this technique is generation and gathering of symbol map information of programs executed in both guests and a VMM; these operations are performed in each guest and in a VMM after the host-level centralized sampling is concluded. As a result, profiling delegation [8, 7] in guests during measurement is no longer required, even in system-wide VM profiling. Then, guest sampling data are converted into symbol names with symbol map information of guest applications

Key Technique 3. How to analyze data: One common-time-based analysis by host time
The third is unified analysis of the entire virtualized environment, including guest applications, with the host time as the common base time. This technique helps to provide an understanding of the correct behavior of applications executed in guests.

The details of each technique are explained in Section 3.3 through 3.5. In these sections, we describe the implementation of unified profiling for a KVM environment because we determined that developing a VMM-level module driver was easier in KVM than in other VM environments. KVM is full virtualization software based on a Linux kernel and is implemented as a kernel module. Therefore, the VMM-level is the same as the Linux kernel layer, and a VMM-level sampling driver should also be implemented as a Linux kernel module. Thus, sampling driver implementation is KVM-specific. In other words, sampling driver implementation and the loaded position in software stack depend on virtualization type and VMM implementation. Therefore, the implementation of Key Technique 1 will be VMM-specific. Other techniques will be common to almost VMMs.

The hardware platform we used was Intel 64 Nehalem architecture. The performance counter overflow interrupts should be non-maskable interrupts (NMIs) since all software, including the OS, drivers, and VM emulation, can be sampled using an NMI as a sampling trigger. Sampling data are all then recorded into the kernel buffer in turn without aggregation.

3.3 Centralized Data Sampling

The execution information of the VMM and the programs running in guests should be sampled centrally at the VMM level, along with host TSC. Sampling data are all recorded in turn, without aggregation, into the host-kernel buffer held in memory by each physical CPU (pCPU) rather than by each virtual CPU (vCPU).

The following contents should be sampled on each physical CPU at each overflow interrupt.

Physical context information (host context)
(1) Time Stamp Counter (TSC)
(2) Process ID (PID)
(3) Program Counter (PC)
Virtual context information (guest context)
(4) VPID (VIRTUAL_PROCESSOR_ID)
(5) Guest PC (Program Counter)
(6) Guest CR3 (Control Register 3)
(7) Exit reason number

Guest context data to be sampled can all be collected from VMCS. Through VMCS, we sample Guest PC, Guest CR3, VPID, and VM-Exit reason number, as shown in Figure 7.

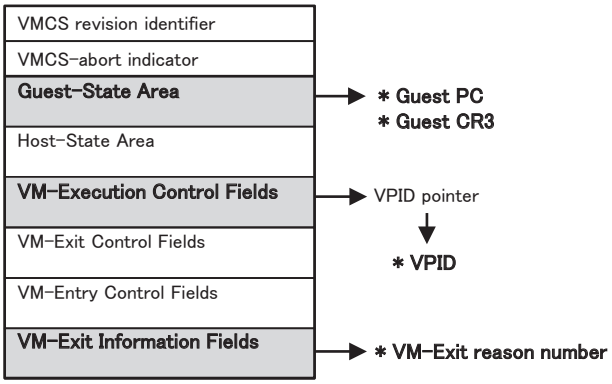


Figure 7: Field contents of VMCS.

We chose the physical TSC value as host time information, which becomes one common base time for unified analysis. There are two reasons we used TSC. The first is that TSC is the most precise clock source, and the second is that it has the lowest reading overhead against the system. However, TSC also has a disadvantage. Although TSC exists in each CPU, we implicitly assume that all TSCs have the same value in the same system. In fact, time skew among TSCs may occur, although it is negligible in many cases. In fact, in our experimental system, maximum skew among CPUs was less than 1 usec, which is negligible compared with using 1 msec as the sampling rate.

A PID is sampled from the management structure area in the OS memory. For example, in Linux, the area is known as the task structure. A PC is sampled from the memory area in which the architecture states of processes interrupted by a counter overflow are saved.

In contrast, virtual context information of the program running in a guest is sampled from the VMCS memory area. For unified profiling, guest CR3 is especially important information that can be mapped to the PIDs of the processes in guests. A PID of a guest program is not saved in the VMCS because the PID is not a register state. Instead of the PID value itself, we can sample guest CR3 from the VMCS as a process identifier of a guest program. The CR3 value is a page table address that specifies a process address space, as shown in Figure 6. Therefore, CR3 holds a process-unique value and can be used instead of PID to show which process is current.

After the sampling measurement, map information should be gathered in each guest. It is necessary to convert a guest CR3 value into a PID. PIDs and CR3 values are both managed by the OS and can be gathered from the OS management structure on each guest OS. We can then generate pair information as map information. For example, the Linux kernel has a PID in its task structure and has a CR3 value in its memory management structure linked from the task structure. We can gather and generate pair information of all processes in a guest kernel. Moreover, the kernel system-map, object map information, and object files should also be collected to analyze guest programs according to function-level.

3.4 Guest Data Interpreting

In Figure 8, the part surrounded by the broken line is the technology we have added to the conventional method, as shown in Figure 3.

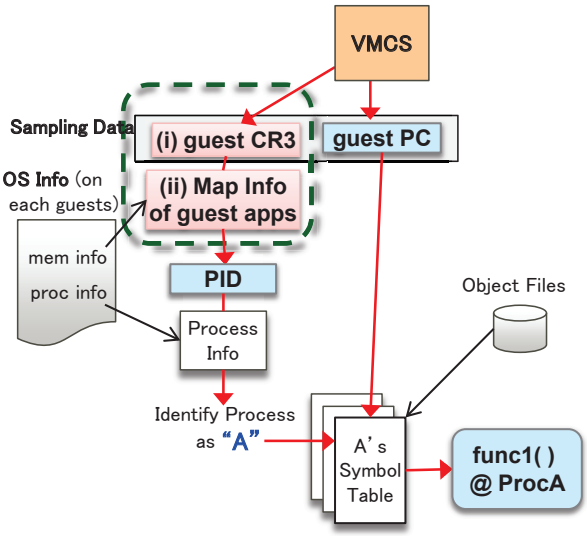


Figure 8: Flow of interpreting guest sampling data with symbol map info of guest apps.

Initially, the process identifier should be sampled to identify executed programs, such as the PID. We decided to sample a value of guest CR3 from VMCS, as shown by (i) in Figure 8, instead of the PID shown in Figure 3. This is because the PIDs in guests cannot be sampled from the VMM. The PIDs in guests are not directly visible to the VMM. Therefore, a VMM-visible process identifier of a guest other than the PID is needed for running programs in guests. For this purpose, we use the page table address, which is used sometimes as the address space identifier (ASID) [11, 3]. The page table is generally a per-process table in all modern operating systems. Consequently, the ASID acts to form a unique global address for each process. Moreover, a control register holds a pointer to the page table. On the x86 CPU, the CR3 register holds the page table address under the control of the OS. The CR3 values of each virtual CPU in the guests are saved into VMCS. Therefore, the guest CR3 is visible to the VMM, and the VMM-level profiler can sample the guest CR3 value from the VMCS, as shown in Figure 8.

Next, mapping information is needed to convert the CR3 value into a PID. The CR3 and PID are both managed by the OS. We decided to gather the CR3 and PID from the OS management structure on each guest OS and to use this paired information as map information, as shown by (ii) in Figure 8. As a result, CR3 can be used to identify executed programs via CR3-PID paired information, which we also refer to as map information. Furthermore, gathering on each guest OS can be operated only once after the sampling measurement. As a result, we succeeded in sampling execution information of guest user applications without the overhead of delegation measurement via virtual interrupt injection. In addition, if guest OS is Linux, we can collect map information of exiting processes during sampling measurement. We can hook process-exit with Linux kernel API.

Thus, Key Technique 2 can be used to resolve the challenge of applying the techniques needed in realizing a practical profiler for the cloud. This second technique in this study was to assume a private cloud as the target because its usage restrictions are looser than those in public clouds. That is, gathering map information would be possible in private clouds, and the trade-off mentioned in Section 2.2 could be resolved.

3.5 Common Time Based Analysis

Each guest had a blank time in which it was unable to run because a physical CPU could not be assigned to a guest. We believed that the blank time should be interpolated with host time as a single common time, and we chose the physical TSC value as host time. As a result, unified profiling results reflect the actual time even in guests, including the blank time. This helps us to understand the correct behavior of applications executed in guests.

To interpolate the blank time in each guest, the compensated sampling data of each guest should be generated prior to the analysis. Figure 9 illustrates how to build the compensated virtual guest sampling data as input data to be used in analyzing guest programs.

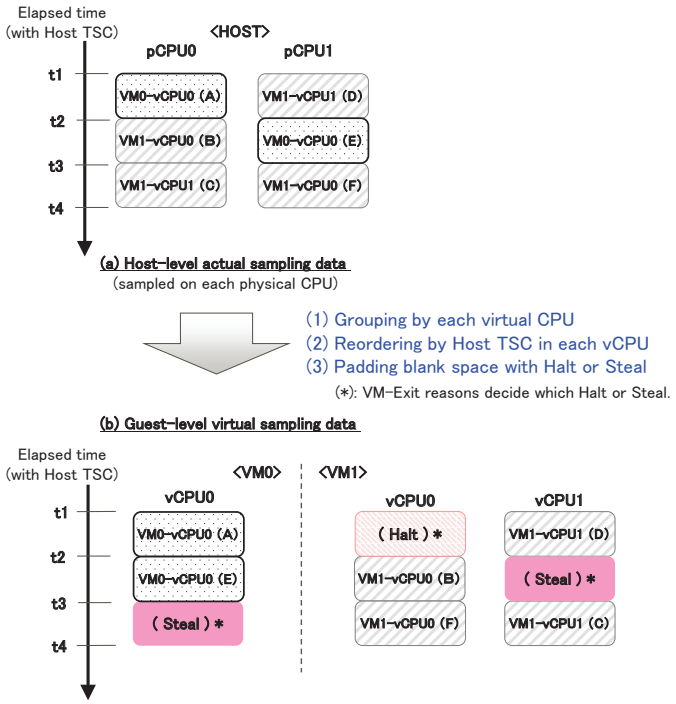


Figure 9: Converting host-level real sampling data to guest-level virtual sampling data, based on host time as common time.

Figure 9 assumes an environment where a system has two pCPUs and two VMs are running: VM0 and VM1. VM0 has one vCPU, and VM1 has two vCPUs. No vCPU is bound to a pCPU. Time axis represents elapsed time by host time. This figure shows sampling data which are sampled from t1 to t4. The virtual guest sampling data are generated with the host time by the virtual CPU unit from real host-level sampling data. The above figure displays the sampling format held in the kernel buffer. We can convert this above format into the below format as virtual guest sampling data. When doing so, the host time is used as the base common time. Thereby, the blank time of each vCPU can be reflected correctly. A blank time is an interval that does not include valid periodic sampling data corresponding to the vCPU. The blank time can be classified into two states, halt

or steal, by the VM-Exit reason. If the exit reason number is 12, the blank time is recognizable as halt. Otherwise, the blank time is handled as steal. When a vCPU is in a steal state, it is ready to run on a pCPU but not assigned to a pCPU. This is because no pCPU is available for the respective vCPUs at that time. Finally, by using the below data as input information for guest analysis, the behaviors of programs in each guest can be analyzed and understood.

4 Evaluation

We implemented a unified profiler as the Linux kernel module driver, which is the same as the VMM-level driver in the KVM environment. The OS, applications, and VMM were all unmodified. Sampling is executed only in a VMM.

We explain in this section the feasibility of implementing unified performance profiling. First, we demonstrate that the applications running in guests can also be analyzed by function-level in each process. Second, we demonstrate that a blank time can be classified into two states: steal or halt. Third, we evaluate its effectiveness in achieving low overhead. The overhead is evaluated quantitatively and is compared with that in previous work. Finally, we present a case study of performance tuning.

Table 1: Summary of experimental environment.

Host Environment	
CPU	Intel Xeon X5570 (Nehalem-EP), 2.93GHz
Num of booted CPU	1 or 2 (§4.1 to §4.3), 4 (§4.4)
Memory	24GB
OS	RHEL Server 6.3[1] 64bit (kernel 2.6.32-279 x86_64)
VMM	KVM (qemu-kvm-0.12.1.2)
Guest Environment	
Num of vCPU	1 or 2 (§4.1 to §4.3), 4 (§4.4)
Memory	4 GB (§4.1 to §4.3), 20 GB (§4.4)
OS	RHEL Server 6.3[1] 64bit (kernel 2.6.32-279 x86_64)
Domain names	guestOS1, guestOS2, guestOS3

Table 2: Measurement setup.

Profile measurement configurations	
Sampling rate	1 msec
Duration	30 sec (§4.1 to §4.3), 60 sec (§4.4)
Sampling base event	CPU_CLK_UNHALTED.REF_P
Workload 1 (§4.1 to §4.2)	
Program	libquantum 0.9.1
Compiler	gcc version 4.4.6 20120305
Optimization	-O2
Invocation	./shor 1397 8
Workload 2 (§4.3)	
Program	MySQL 5.1.61-4.el6 for x86_64
Benchmark	SysBench 0.4.12-5.el6 for x86_64
Workload 3 (§4.4)	
Program	Java HotSpot 64-Bit Server VM, version 1.7.0_79
Benchmark	SPECjbb2013-Composite:Single JVM

[1]Red Hat Enterprise Linux Server release 6.3

The experimental environment is summarized in Table 1. The host machine is a Fujitsu Primergy BX900 blade server with one, two, or four CPU cores available. For the native environment, the OS is the Linux kernel 2.6.32. For the virtualized environment, the VMM is qemu-kvm-0.12.1.2 based on Linux kernel 2.6.32. The guest environment consists of one VM or three VMs activated on one, two, or four pCPUs, and all guests have one, two, or four vCPUs. Guest domain names are guestOS1, guestOS2, and guestOS3. The guest OS version is the same as the host. Table 2 shows the conditions under which the profiling measurements were made. We used libquantum [18], MySQL [1], and Java VM (JVM) [20] as a running program during measurement. They are profile target programs. First, libquantum was used for fundamental accuracy evaluation. Libquantum-0.9.1 is one of the applications included in the SPEC CPU2006 [25]. It is a C library for the simulation of quantum mechanics, with a special focus on quantum computing. The invocation of the Shor command executes Shor's factoring algorithm. Then, MySQL was used for evaluation of overhead against the performance of online transaction processing (OLTP). MySQL is one of components of the widely used LAMP (Linux, Apache, MySQL, and PHP/Perl/Python) implementations. Therefore, it is a popular database for web applications. Moreover, we used Sysbench [2] for measurement of MySQL's OLTP performance. Finally, JVM was used in order to demonstrate performance tuning with unified profiling. We used SPECjbb2013 [26] for measurement of java transaction throughput.

In advance of showing evaluation results, we clarify five terms to classify profiling results. The following five terms are description for identifying profiling approaches. Among them, unified VM profiling is the fifth item that includes both host-level profiling and guest-level profiling.

1. Native profiling: Conventional profiling
(a) Profiled programs are running in HOST
(b) Data sampling is performed in HOST
(c) Profile result shows programs running in HOST (but can not distinguish VMs)
2. Guest-inside profiling: Conventional approaches for VM profiling
(a) Profiled programs are running in GUEST
(b) Data sampling is performed in GUEST or HOST-and-GUEST
(c) Profile result shows programs running in GUEST
3. Host-level profiling: Subset of unified VM profiling
("Host-level" is the same meaning as aforementioned "VMM-level".)
(a) Profiled programs and VMs are running in HOST
(b) Data sampling is performed in HOST
(c) Profile result shows programs and VMs(guest domain names) running in HOST
4. Guest-level profiling: Subset of unified VM profiling
(a) Profiled programs are running in GUEST
(b) Data sampling is performed ONLY in HOST
(c) Profile result shows programs running in GUEST
5. Unified VM profiling: Our approach for VM profiling, consisting of above 3 and 4
(a) Profiled programs are running in both GUEST and HOST
(b) Data sampling is performed ONLY in HOST
(c) Profile result shows programs running in both GUEST and HOST (and can also show VM domain names)

4.1 Profiling results

First, to verify the accuracy of our profiler in function-level, we ran libquantum as the profiled application in both a native environment and a virtualized environment. The native environment has one pCPU. The virtualized environment has one vCPU and consists of one VM: guestOS1. Table 3 shows the output of native profiling. Table 4 presents the results of guest profiling. We compare these results in Figure 10. We expected that the results of VM profiling would be close to those of native profiling. However, from experience, we were also sure that there had to be a slight difference between them because of non-deterministic factors (e.g., the impact of operation of VMM or other VMs).

Table 3: Native profiling.

Total samples:29938 OS:USER = 1.17%:98.83%

Samples	%Ratio	Function	Module
18281	61.06	quantum_toffoli	libquantum
5836	19.49	quantum_sigma_x	libquantum
4624	15.45	quantum_cnot	libquantum
674	2.25	quantum_swaptheleads	libquantum
171	0.57	quantum_objcode_put	libquantum
14	0.05	rb_reserve_next_event	vmlinux
12	0.04	update_wall_time	vmlinux
12	0.04	rb_end_commit	vmlinux
11	0.04	run_timer_softirq	vmlinux
10	0.03	ring_buffer_lock_reserve	vmlinux

Table 4: Unified VM profiling. (guest-level profiling)

Total samples:29996 OS:USER:steal = 1.49%:93.47%:5.04%

Samples	%Ratio	Function	Module
16984	56.62	quantum_toffoli	libquantum
6752	22.51	quantum_sigma_x	libquantum
3330	11.10	quantum_cnot	libquantum
1511	5.04	[steal]	(outside)
766	2.56	quantum_swaptheleads	libquantum
198	0.66	quantum_objcode_put	libquantum
28	0.09	apic_timer_interrupt	vmlinux
22	0.07	native_apic_mem_write	vmlinux
17	0.06	__run_hrtimer	vmlinux
17	0.06	pvclock_clocksource_read	vmlinux

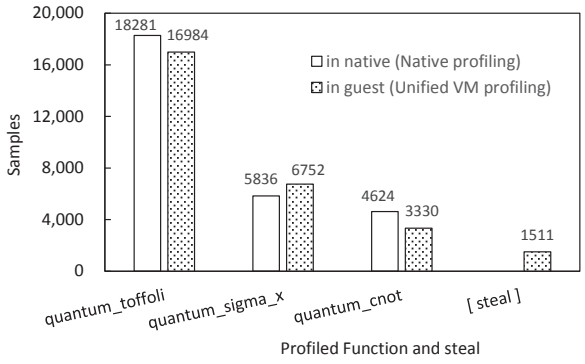


Figure 10: Comparison between native profiling and unified VM profiling.

According to Figure 10, the experimental results are as expected. The tendencies of both results are roughly the same, although in this case, the difference from 3% to 5% can be confirmed, based on the total number of samples. Moreover, our unified profiler is found to be able to indicate steal cycle counts as the cause of the above slight difference. This is the effect of common-time-based analysis by host time. The steal must include non-deterministic causes to be solved. In fact, from the analysis of exit reason number, it can be seen that the causes of these steal cycles include external interrupts (exit-reason number 1) and APIC accesses inside the guest (exit-reason number 44). The former accounts for 82% of these steal cycles, and the latter accounts for 18%. From the results, we found that the impact of steal could occur without other VMs running on the same host. Such steal is unrecognizable with existing standard OS tools.

Table 5: Unified VM profiling. (host-level profiling)

Total samples:29914 OS:USER:VM = 4.71%:0.34%:94.95%

Samples	%Ratio	Function	Module
28402	94.95	[guestOS1]	(VM1)
82	0.27	vmx_vcpu_run	kvm_intel
41	0.14	update_curr	vmlinux
34	0.11	copy_user_generic_string	vmlinux
32	0.11	kvm_arch_vcpu_ioctl_run	kvm
32	0.11	rb_reserve_next_event	vmlinux
31	0.10	(stripped local_functions)	qemu-kvm
29	0.10	ring_buffer_lock_reserve	vmlinux
24	0.08	update_wall_time	vmlinux
24	0.08	x86_decode_insn	kvm

Table 5 shows the output of host-level VM profiling. According to the comparison between the host-level results and the guest-level results in Table 4, the host-level results except guestOS1 and the guest-level result of steal are in close agreement. Thus, the accuracy of our VM profiler can be confirmed.

Next, we executed libquantum in two different processes, process 1 and process 2, in both a native environment and a virtualized environment. Table 6 and Table 7 present the profiling results of these two processes by native profiling and guest-level profiling. Regarding the result in the native environment, Figure 11a shows that the same functions of each process are almost the same in CPU usage. Simultaneously, we observed that CPU usage of each process was also similar constantly on 49.9%:49.9% with top command. Even in the virtualized environment, if similar CPU usage between each process can be seen, we can expect that the same functions of each process have almost the same number of samples and the same CPU usage. Indeed, even in the virtualized environment, we can observe constantly similar CPU usage between each process on 49.9%:49.9% with top command, and this expectation can be also confirmed with our VM profiler, as shown in Figure 11b. Thus, our unified VM profiling is found to be able to distinguish user processes accurately.

Then, we had two vCPUs available in the guest and executed each libquantum process bonded to the specified either vCPU. Table 8 and Table 9 show the profiling results of these two processes by guest-level profiling. Table 8 presents the results of profiling on vCPU0 and Table 9 presents that of profiling on vCPU1. We expected similar tendencies to Figure 11b. In fact, as shown in Figure 12, this expectation can be confirmed. The same functions of each process are almost the same in CPU usage. Furthermore, according to the comparison of the number of samples on between the one vCPU and the two vCPUs, the number of samples in each environment are in close agreement, as shown in Figure 13. Thus, our unified VM profiling can provide good results with multiple vCPUs overcommitted to a pCPU.

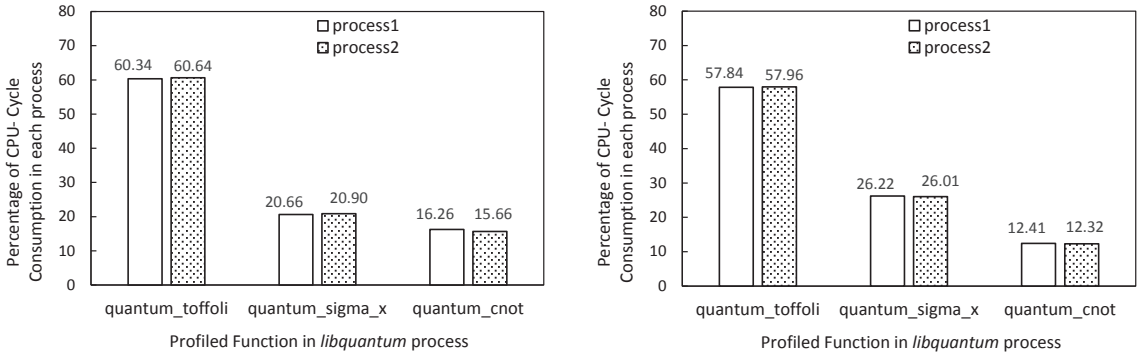
Finally, we had two pCPUs available and pinned each vCPU to the specified either pCPU. Each libquantum process was executed on the specified either vCPU. Table 10 and Table 11 show the profiling results of these two processes by guest-level profiling. Table 10 presents the results of profiling on vCPU0 pinned to pCPU0 and Table 11 presents that of profiling on vCPU1 pinned to

Table 6: Native profiling of two same workloads.

Total samples:29938		OS:USER = 1.27%:98.73%	
Samples	%Ratio	Function	Module
8963	29.94	quantum_toffoli	libquantum(2)
8915	29.78	quantum_toffoli	libquantum(1)
3089	10.32	quantum_sigma_x	libquantum(2)
3052	10.19	quantum_sigma_x	libquantum(1)
2402	8.02	quantum_cnot	libquantum(1)
2314	7.73	quantum_cnot	libquantum(2)
324	1.08	quantum_swaptheleads	libquantum(2)
323	1.08	quantum_swaptheleads	libquantum(1)
91	0.30	quantum_objcode_put	libquantum(2)
78	0.26	quantum_objcode_put	libquantum(1)
15	0.05	__rb_reserve_next	vmlinux
12	0.04	ring_buffer_lock_reserve	vmlinux

Table 7: Unified VM profiling of two same workloads in a guest.

Total samples:30000		OS:USER:steal = 1.75%:93.27%:4.98%	
Samples	%Ratio	Function	Module
8103	27.01	quantum_toffoli	libquantum(1)
8094	26.98	quantum_toffoli	libquantum(2)
3673	12.24	quantum_sigma_x	libquantum(1)
3633	12.11	quantum_sigma_x	libquantum(2)
1739	5.80	quantum_cnot	libquantum(1)
1721	5.74	quantum_cnot	libquantum(2)
1494	4.98	[steal]	(outside)
392	1.31	quantum_swaptheleads	libquantum(1)
384	1.28	quantum_swaptheleads	libquantum(2)
128	0.43	quantum_objcode_put	libquantum(2)
98	0.33	quantum_objcode_put	libquantum(1)
37	0.12	apic_timer_interrupt	vmlinux



(a) Native profiling

(b) Unified VM profiling (guest-level).

Figure 11: Comparison of profiling results in the same functions between two different libquantum processes.

Table 8: Unified VM profiling on vCPU0 in a guest with two vCPUs on one pCPU.

Total samples:30000		OS:USER:steal = 0.75%:46.77%:52.48%	
Samples	%Ratio	Function	Module
14994	49.98	[steal]	(on vcpu1)
8669	28.90	quantum_toffoli	libquantum(1)
3210	10.70	quantum_sigma_x	libquantum(1)
1678	5.59	quantum_cnot	libquantum(1)
750	2.50	[steal]	(outside)
345	1.15	quantum_swaptheleads	libquantum(1)
122	0.41	quantum_objcode_put	libquantum(1)
26	0.09	apic_timer_interrupt	vmlinux
18	0.06	pvclock_clocksource_read	vmlinux
8	0.03	_spin_lock	vmlinux

Table 9: Unified VM profiling on vCPU1 in a guest with two vCPUs on one pCPU.

Total samples:30000		OS:USER:steal = 0.90%:46.37%:52.72%	
Samples	%Ratio	Function	Module
15006	50.02	[steal]	(on vcpu0)
8612	28.71	quantum_toffoli	libquantum(2)
3176	10.59	quantum_sigma_x	libquantum(2)
1641	5.47	quantum_cnot	libquantum(2)
811	2.70	[steal]	(outside)
353	1.18	quantum_swaptheleads	libquantum(2)
121	0.40	quantum_objcode_put	libquantum(2)
38	0.13	apic_timer_interrupt	vmlinux
28	0.09	pvclock_clocksource_read	vmlinux
14	0.05	native_apic_mem_write	vmlinux

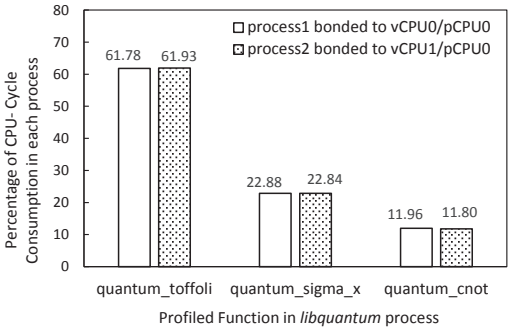


Figure 12: Comparison of profiling results in the same functions between two different libquantum processes on each vCPU.

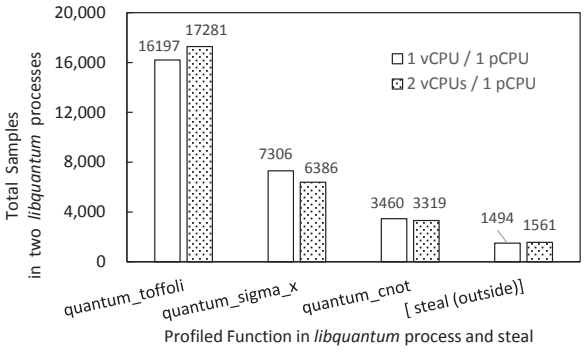


Figure 13: Comparison of the total number of samples in the same functions of libquantum between the cases of using one vCPU and two vCPUs.

Table 10: Unified VM profiling on vCPU0 in a guest with two vCPUs on two pCPUs.

Total samples:29998		OS:USER:steal = 1.05%:93.71%:5.23%	
Samples	%Ratio	Function	Module
17243	57.48	quantum_toffoli	libquantum(1)
6743	22.48	quantum_sigma_x	libquantum(1)
3545	11.82	quantum_cnot	libquantum(1)
1570	5.23	[steal]	(outside)
458	1.53	quantum_swaptheleads	libquantum(1)
114	0.38	quantum_objcode_put	libquantum(1)
38	0.13	pvclock_clocksource_read	vmlinux
26	0.09	apic_timer_interrupt	vmlinux
24	0.08	native_apic_mem_write	vmlinux
9	0.03	rb_reserve_next_event	vmlinux

Table 11: Unified VM profiling on vCPU1 in a guest with two vCPUs on two pCPUs.

Total samples:30000		OS:USER:steal = 1.05%:96.36%:2.59%	
Samples	%Ratio	Function	Module
17873	59.58	quantum_toffoli	libquantum(2)
6520	21.73	quantum_sigma_x	libquantum(2)
3697	12.32	quantum_cnot	libquantum(2)
777	2.59	[steal]	(outside)
653	2.18	quantum_swaptheleads	libquantum(2)
155	0.52	quantum_objcode_put	libquantum(2)
25	0.08	native_apic_mem_write	vmlinux
23	0.08	pvclock_clocksource_read	vmlinux
21	0.07	apic_timer_interrupt	vmlinux
16	0.05	rb_reserve_next_event	vmlinux

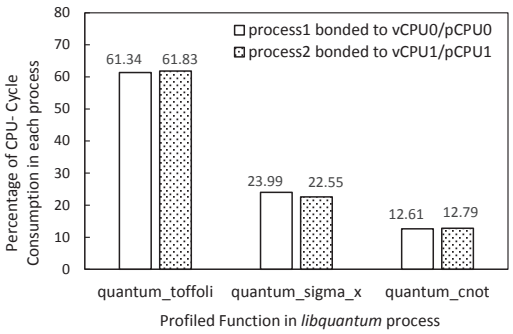


Figure 14: Comparison of profiling results in the same functions between two different libquantum processes on each vCPU.

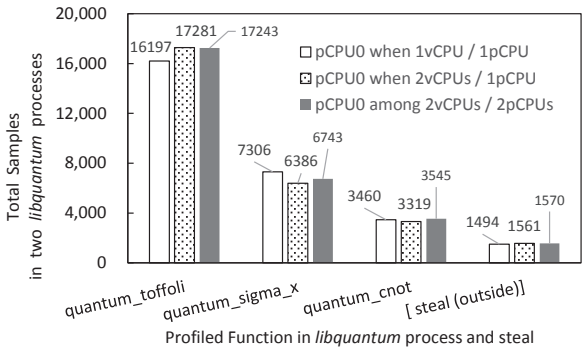


Figure 15: Comparison of the total number of samples in the same functions on pCPU0 among the multiple vCPUs/pCPUs environments.

pCPU1. We expected similar results to Figure 12. In fact, as shown in Figure 14, this expectation can be confirmed. Moreover, we expected that the number of samples per pCPU would be close to previous ones. According to Figure 15, the experimental results are as expected. Thus, our unified VM profiler works well with multiple pCPUs.

4.2 Blank Time Classification: Steal or Halt

We need to distinguish between halt and steal for performance debugging and tuning. Halt is scheduled and controlled by OS. Consequently, the cause of halt exists inside self OS environment. By contrast, the cause of steal exists outside self OS environment. Steal is caused by the operation of a VMM. There is the cause of steal in a VMM or in other VMs.

First, in this section, we boot three guests (guestOS1, guestOS2, and guestOS3) on one pCPU in order to verify the accuracy of steal results in guest-level profiling. Each guest has one vCPU. In each guest, we run libquantum in the same way. Table 12 presents the results of guest-level profiling for guestOS1-domain. Table 13 shows the results of host-level profiling. According to Table 12, the steal in VM1 is found to be in a ratio of 69.39%. Moreover, the total ratio except VM1 in host-level is 69.39% from Table 13. These ratios are the same, which indicates the accuracy of the steal analysis by our unified profiler.

Table 12: Unified VM profiling (guest-level results of guestOS1), with stolen by other VMs.

Total samples:29999 OS:USER:steal = 0.79%:29.82%:69.39%

Samples	%Ratio	Function	Module
20816	69.39	[steal]	(outside)
5156	17.19	quantum_toffoli	libquantum
2286	7.62	quantum_sigma_x	libquantum
1160	3.87	quantum_cnot	libquantum
265	0.89	quantum_swaptheleads	libquantum
73	0.24	quantum_objcode_put	libquantum
21	0.07	apic_timer_interrupt	vmlinux
11	0.04	pvclock_clocksource_read	vmlinux
10	0.03	ring_buffer_lock_reserve	vmlinux
9	0.03	native_apic_mem_write	vmlinux

Table 13: Unified VM profiling (host-level results), running 3VMs.

Total samples:29932 OS:USER:VM = 6.81%:0.68%:92.51%

Samples	%Ratio	Function	Module
9402	31.41	[guestOS3]	(VM3)
9161	30.61	[guestOS1]	(VM1)
9125	30.49	[guestOS2]	(VM2)
137	0.46	vmx_vcpu_run	kvm_intel
67	0.22	update_curr	vmlinux
59	0.20	rb_reserve_next_event	vmlinux
59	0.20	(stripped local functions)	qemu-kvm
50	0.17	copy_user_generic_string	vmlinux
49	0.16	kvm_arch_vcpu_ioctl_run	kvm
39	0.13	ring_buffer_lock_reserve	vmlinux

Then, we shut down two guest domains, guestOS2 and guestOS3. we activate only one guest, guestOS1, in order to verify the accuracy of a halt result (i.e., idle) in guest-level profiling. This time, we run libquantum for only 10sec during measurement in the guestOS1. The duration of measurement is 30sec, as shown in Table 2. For remaining 20sec, the guestOS1 stays in idle. Therefore,

we expected that the idle ratio, in both the guest and the host, would account for about two-thirds. Table 14 shows the results of guest-level profiling and Table 15 shows that of host-level profiling. In fact, from the results in Table 14, the idle (halt_exit) in a guest is found to be in a ratio of 67.40%. Meanwhile, the ratio of idle function in a host is 66.73%, as shown in Table 15. These ratios are close to each other and account for about two-thirds, as was expected. Therefore, it can be seen that the unified VM profiling can correctly grasp the ratio of a guest's halt. Furthermore, from the results in this section, we found that the unified VM profiling could classify a blank time into two states: steal or halt. This is the effect of both exit-reason-number sampling and common-time-based analysis by host time.

Table 14: Unified VM profiling (guest-level results), with halt executed.

Total samples:30000 OS:USER:IDLE:steal = 0.54%:30.42%:67.40%:1.64%

Samples	%Ratio	Function	Module
20220	67.40	[idle]	(halt_exit)
4987	16.63	quantum_toffoli	libquantum
2326	7.76	quantum_sigma_x	libquantum
1063	3.54	quantum_cnot	libquantum
492	1.64	[steal]	(outside)
392	1.31	quantum_gate1	libquantum
237	0.79	quantum_swaptheleads	libquantum
61	0.20	quantum_objcode_put	libquantum
50	0.17	__mulsc3	libquantum
13	0.04	apic_timer_interrupt	vmlinux

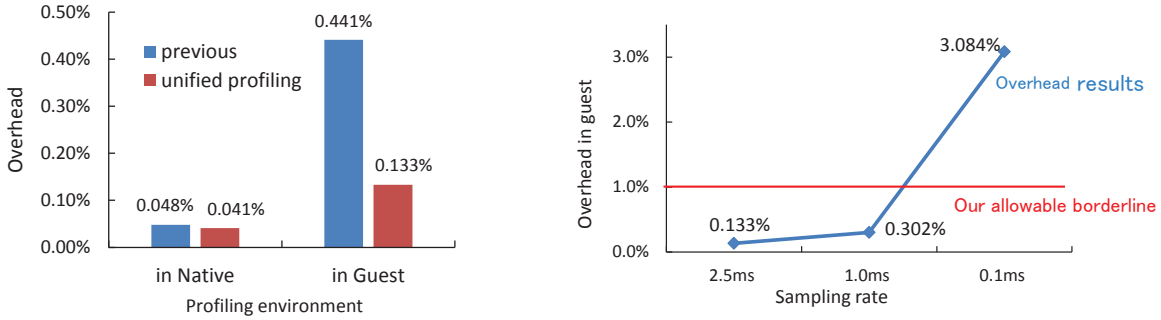
Table 15: Unified VM profiling (host-level results), with halt-exit from a VM.

Total samples:29943 OS:USER:IDLE:VM = 2.17%:0.17%:66.73%:30.93%

Samples	%Ratio	Function	Module
19982	66.73	poll_idle	vmlinux
9260	30.93	[guestOS1]	(VM1)
25	0.08	vmx_vcpu_run	kvm_intel
21	0.07	ring_buffer_lock_reserve	vmlinux
17	0.06	update_curr	vmlinux
17	0.06	do_select	vmlinux
15	0.05	rb_reserve_next_event	vmlinux
15	0.05	(stripped local functions)	qemu-kvm
13	0.04	trace_clock_local	vmlinux
11	0.04	apic_timer_interrupt	vmlinux

4.3 Profiling Overhead

For practical use, there must be low overhead in addition to the accuracy of profiling results. On the other hand, in order to improve accuracy, there must be more sampling data because profiling accuracy is based on statistical analysis method. For this purpose, either fine-grained sampling or long-term sampling is available. In general, the former is used. However, the shorter sampling rate is, the higher overhead becomes. That is, there is a trade-off between overhead and accuracy. Consequently, we need to grasp the shortest sampling rate within the allowable overhead. In general, service-level performance management tools are accepted in overhead from 5% to 10%. In contrast, the allowable overhead of profiling appears to be less than 5%, because profilers should be used even in performance-critical cases. Profiling measurement inevitably produces performance overhead against the system because of the handling of counter overflow interrupts and data sampling processes. Ideally, however, we require a lower overhead of 1% or less for practical use. Accordingly, we evaluated the overhead to examine whether the unified profiling could be used in practice.



(a) Verification of the effect of eliminating the delegation at 2.5 msec-sampling-rate.

(b) Exploration of practical sampling rate value for unified VM profiling.

Figure 16: Overhead of unified VM profiling.

First, we used an evaluation workload similar to that of Du et al. [8]. The workload code consists of two functions, which perform floating arithmetic. One consumes about 80% of CPU cycles, and the other consumes about 20%. We determined the overhead by comparing the execution times, with and without profiling, of a computation-intensive workload that executed a fixed number of iterations. Previous research [8] indicated the native profiling overhead was 0.048% at 2.5 msec-sampling-rate, while in our native environment, the overhead of profiling at that rate is 0.041%, as shown in Figure 16a. These results are in close agreement. Therefore, the base condition of our experiment environment is nearly the same as that in the previous study. Moreover, KVM system-wide profiling overhead in the previous study was 0.441% at 2.5 msec-sampling-rate. According to the previous KVM system-wide results, we estimated that the overhead of VM profiling without delegation would be 0.125%. In fact, Figure 16a shows that our result of 0.133% is close to the expected result. Therefore, the effect of eliminating the delegation can be confirmed.

Figure 16b shows the overhead results (blue curve) and the allowable borderline (red line) of our profiling overhead. From the results, we found that, with unified profiling, we could use 1 msec as the sampling rate for practical use, even in virtualized environments. In the guest environment, the overhead of unified profiling at 1.0 msec-sampling-rate is 0.302%, and that of 0.1 msec-sampling-rate is 3.084%. The result of the 1 msec-sampling-rate has an overhead sufficiently lower than 1%.

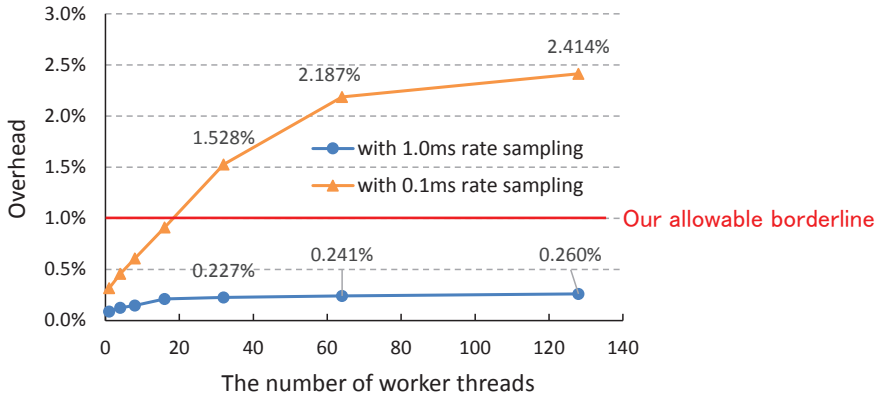


Figure 17: Overhead against OLTP throughput of MySQL.

Next, we used MySQL [1] for evaluation of overhead against the performance of online transaction processing (OLTP). MySQL is widely used open-source relational database management system (RDBMS). MySQL is one of components of the widely used LAMP (Linux, Apache, MySQL, and PHP/Perl/Python) implementations. Therefore, it is a popular database for web applications.

Moreover, we used Sysbench [2] for measurement of MySQL's OLTP performance. In this evaluation, we had two pCPUs and two vCPUs available, and pinned each vCPU to the specified either pCPU. Furthermore, a MySQL daemon process and a Sysbench process were bonded to the specified either vCPU. Sampling was continuous during the measurement by Sysbench. We ran the measurement five times, reporting the mean and standard deviation. Figure 17 shows the overhead against OLTP throughput, imposed by unified profiling. As a result, we can confirm that 1 msec-sampling-rate is sufficient for practical use. In addition, we found that overhead became lower than previous simple evaluation. This is because profiling overhead depends on CPU utilization. Figure 18 and Figure 19 presrent the OLTP throughputs and CPU usage. On the previous overhead test, CPU usage was almost 100%. On the other hand, in case of MySQL, CPU usage was less than about 93%, as shown in Figure 19.

Figure 18: OLTP throughput by MySQL.

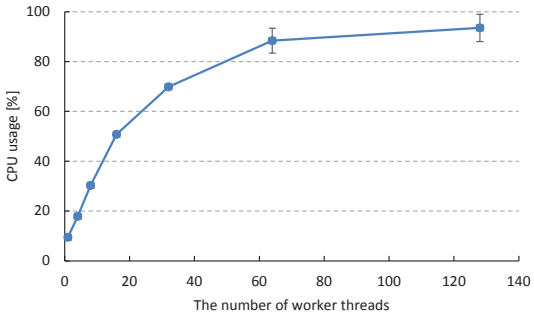


Figure 19: CPU utilization by OLTP.

4.4 Performance Tuning Using Unified VM Profiler

In this section, we identify a root cause of low performance of SPECjbb2013 [26] executed in a guest. The SPECjbb2013 benchmark has been developed from the ground up to measure performance based on the latest Java application features. It is relevant to all audiences who are interested in Java server performance. In this case, the host has four pCPUs and 24 GB physical memory. The guest has four vCPUs and 20 GB memory. Each of the vCPUs is pinned to each of the pCPUs. To understand a characteristic of the behavior of JVM in a guest, we profile the guest in which SPECjbb2013 is running. Sampling is executed around at the timing of max-jOPS, with 1.0 ms-sampling-rate and 60 sec duration.

Table 16: SPECjbb2013 profile in a guest.

Total samples:240007 OS:USER:IDLE(halt):steal = 7.43%:81.83%:0.01%:10.73%

Samples	%Ratio	Function	Module
74427	**31.01**	**(jvm internal functions)**	**java**
26637	**11.10**	**SpinPause**	**libjvm.so**
25749	10.73	[steal]	(outside)
24123	**10.05**	**ParMarkBitMap::live_words_in_range**	**libjvm.so**
9174	3.82	ParallelTaskTerminator::offer_termination	libjvm.so

Table 16 shows the sampling result. Table 16 indicates that the ratio of SpinPause is 11.10% and the ratio of ParMarkBitMap::live_words_in_range is 10.05%. They don't usually appear in a profiling of SPECjbb2013. ParMarkBitMap::live_words_in_range routine is included in garbage collection (GC) implementation. Consequently, lock contention seems to have occurred because of GC. The setting of JVM Heap memory size was 8 GB. Therefore, we increase heap size, from 8

GB to 16 GB. As a result, the performance improved, as shown in Figure 20. From the result in Table 17, we can confirm that ParMarkBitMap::live_words_in_range and SpinPause disappear and the ratio of JVM improve.

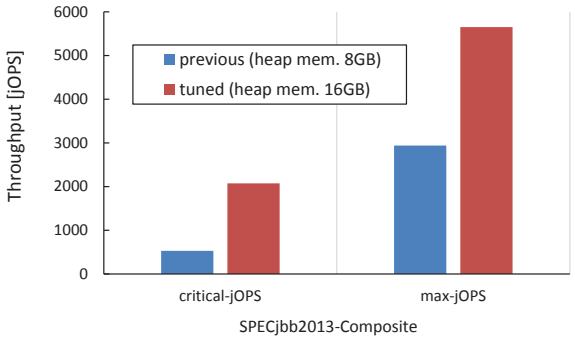


Figure 20: Tuning result using unified VM profiler.

Table 17: Tuned SPECjbb2013 profile in a guest. (increased heap memory size of JVM)

Total samples:240009 OS:USER:IDLE(halt):steal = 5.44%:90.08%:0.02%:4.46%

Samples	%Ratio	Function	Module
180019	**75.01**	**(jvm internal functions)**	**java**
10715	4.46	[steal]	(outside)
3376	1.41	JVM_LatestUserDefinedLoader	libjvm.so
2265	0.94	PSPromotionManager::copy_to_survivor_space	libjvm.so
1628	0.68	pvclock_clocksource_read	vmlinux

5 Conclusions and Future Work

Virtualized environments have become popular as the platform underlying cloud computing. However, performance profilers are not yet in practical use for VMs. Therefore, diagnosing performance problems of applications executed in guests is still difficult.

In this paper, we presented the problems and challenges towards practical profiling of virtualized environments and proposed three novel techniques for unified profiling of an entire virtualized environment. We actually developed a unified profiler and demonstrated that unified profiling works well, as was expected, even in a virtualized environment. The unified profiling results reflect the actual time even in guests, with the host time, including the blank time in which it was unable to run due to having no physical CPU available. As a result, this proposed method, with hardware performance counters, might help to identify the reason for performance degradation of workloads in both guests and a host. Consequently, more effective use of resources is expected to help lower management costs.

In the future, scale-out profiling for multiple hosts needs to be discussed in terms of cloud environments. In addition, not only the conventional reactive profiling use but also proactive use is suggested for more practical use in datacenters. For this purpose, a performance analysis tool like a profiler based on processor performance counters should be running at all times and should automatically indicate the symptoms of problems. Our central goal is to realize such a practical profiler for cloud computing. For this purpose, our objective includes three phases shown in Figure 21: (a) inventing a host-wide performance profiling method for a virtualized system with CPU performance counters; (b) making infrastructure for scale-out profiling across multiple hosts; and (c) enhancing the sampling and analysis methods for continuous profiling. The first is a basic method for one

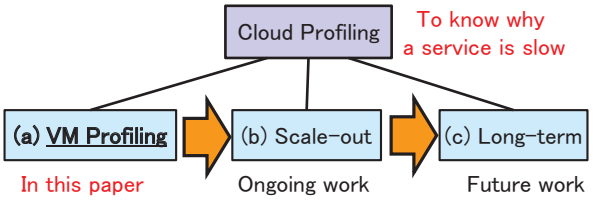


Figure 21: Approach concept of cloud performance analysis.

host and is indispensable for performance debugging of a complex virtualized system. This method is the core technology for cloud performance analysis. The second is required to scale out above the basic method for multiple hosts in a cloud environment. The third is required for performance troubleshooting of in-service cloud environments. It is very hard to reproduce the problems of cloud environments, like Mandelbugs [9, 10, 27]. This is because program behaviors on each VM in a cloud are complex and non-deterministic. Therefore, to debug and resolve problems, it is helpful to monitor performance events continuously and analyze them in real-time. Moreover, it is necessary to collect the execution information of running programs during the operation at proper collecting interval. In this study, we focused on the first phase.

References

[1] MySQL. *http://www.mysql.com/*.

[2] SysBench. *https://github.com/akopytov/sysbench/*.

[3] S. Jones A. Arpaci-Dusseau and R. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *ATEC '06 Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 1–14, 2006.

[4] Intel Corp. Hardware-assisted Intel Virtualization Technology (Intel VT). *http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/hardware-assist-virtualization-technology.html*.

[5] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manual volume 3: System programming guide. *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*.

[6] Intel Corp. Intel VTune Amplifier. *http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/*.

[7] J. Du, N. Sehrawat, and W. Zwaenepoel. Performance profiling in a virtualized environment. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, 2010.

[8] J. Du, N. Sehrawat, and W. Zwaenepoel. Performance profiling of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments, VEE'11*, 2011.

[9] M. Grottke and K. S. Trivedi. A classification of software faults. In *Supplemental Proc. 16th International IEEE Symposium on Software Reliability Engineering*, pages 4.19–4.20, 2005.

[10] M. Grottke and K. S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *IEEE Computer*, 40(2):107–109, 2007.

[11] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *ISCA '93 Proceedings of the 20th annual international symposium on computer architecture*, pages 39–50, 1993.

[12] Red Hat, Inc. Red Hat Enterprise Linux 7: Virtualization Tuning and Optimization Guide: §8.3. Virtual Performance Monitoring Unit (vPMU). *https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Virtualization_Tuning_and_Optimization_Guide/sect-Virtualization_Tuning_Optimization_Guide-Monitoring_Tools-vPMU.html*.

[13] VMware, Inc. Knowledge base: Using virtual cpu performance monitoring counters (2030221). *http://kb.vmware.com/kb/2030221*.

[14] Matthew Johnson, Heike McCraw, Shirley Moore, Phil Mucci, John Nelson, Dan Terpstra, Vince Weaver, and Tushar Mohan. PAPI-V: Performance monitoring for virtual machines. In *2012 41st International Conference on Parallel Processing Workshops*, pages 194–199, September 2012.

[15] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Linux Symposium*, 2007.

[16] Avi Kivity. Performance monitoring for KVM guests. In *KVM Forum*, 2011.

[17] J. Levon and P. Elie. Oprofile: A system profiler for linux. *http://oprofile.sourceforge.net*.

[18] Libquantum. the C library for quantum computing and quantum simulation. *http://www.libquantum.de/*.

[19] A. Menon, J.R. Santos, Y. Turner, G.J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual Execution Environments, VEE'05*, 2005.

[20] Oracle Corporation. jdk-7u79-linux-x64.rpm. *http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html*.

[21] Boris Ostrovsky. Xen PMU: Perf support in Xen. In *Xen Project Developer Summit*, 2013.

[22] PAPI. PAPI 5.0.0 (aka PAPI-V). *http://icl.cs.utk.edu/PAPI/news/news.html?id=300*.

[23] perf. Linux profiling with performance counters. *https://perf.wiki.kernel.org/index.php/Main_Page*.

[24] B. Serebrin and D. Hecht. Virtualizing performance counters. In *5th Workshop on System-level Virtualization for High Performance Computing, Euro-Par 2011:HPCVirt*, August 2011.

[25] Standard Performance Evaluation Corporation. SPEC CPU 2006. *https://www.spec.org/cpu2006/*.

[26] Standard Performance Evaluation Corporation. SPECjbb2013. *https://www.spec.org/jbb2013/*.

[27] Kishor S. Trivedi, Rajesh Mansharamani, Dong Seong Kim, Michael Grottke, and Manoj Nambiar. Recovery from failures due to mandelbugs in it systems. In *17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 224–233, December 2011.

[28] Vincent M. Weaver, Dan Terpstra, Heike McCraw, Matt Johnson, Kiran Kasichayanula, James Ralph, John Nelson, Phil Mucci, Tushar Mohan, and Shirley Moore. PAPI 5: Measuring power, energy, and the cloud. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2013)*, pages 124–125, April 2013.

[29] Zhang Y. Enhance perf to collect KVM guest os statis-tics from host side. *http://lwn.net/Articles/378778*.