# Performance Profiling in a Virtualized Environment

Jiaqing Du
*EPFL, Switzerland*

Nipun Sehrawat
*IIT Guwahati, India*

Willy Zwaenepoel
*EPFL, Switzerland*

## Abstract

Virtualization is a key enabling technology for cloud computing. Many applications deployed in a cloud run in virtual machines. However, profilers based on CPU performance counters do not work well in a virtualized environment. In this paper, we explore the possibilities for achieving performance profiling in virtual machine monitors (VMMs) built on paravirtualization, hardware assistance, and binary translation. We present the design and implementation of performance profiling for a VMM based on the x86 hardware extensions, with some preliminary experimental results.

## 1 Introduction

Virtualization is a key enabling technology for cloud computing. Applications deployed in a virtualization-based cloud, e.g., Amazon's EC2, run inside virtual machines, which are dynamically mapped onto a cluster of physical machines. By providing workload consolidation and migration, virtualization improves the resource utilization and the scalability of a cloud. In a virtualized environment, the virtual machine monitor (VMM) exports a set of virtual machines to the guest operating systems (hereafter referred to as guests) and manages accesses to the underlying hardware resources shared by multiple guests. In this way virtualization makes it possible to run multiple operating systems simultaneously on a single physical machine [11, 3, 8].

In modern processors, the performance monitoring unit (PMU) is indispensable for performance debugging of complex software systems. It is employed by software profilers to monitor the micro-architectural behavior of a program. Typical hardware events monitored include clock cycles, instruction retirements, cache misses, TLB misses, etc. As an example, Table 1 shows a typical output of a profiler. It presents the top-five time-consuming functions of the whole system. Developers can refer to the output of a profiler to identify bottlenecks in a program and tune its performance . PMU-based performance profiling in a non-virtualized environment has been well studied. Mature tools built upon PMUs exist in almost every popular operating system [9, 6]. They are used extensively to tune software performance. However, this is not the case in a virtualized environment.

| % CYCLE | Function | Module |
|---|---|---|
| 98.5529 | vmx_vcpu_run | kvm-intel.ko |
| 0.2226 | (no symbols) | libc.so |
| 0.1034 | hpet_cpuhp_notify | vmlinux |
| 0.1034 | native_patch | vmlinux |
| 0.0557 | (no symbols) | bash |

Table 1: Top five time-consuming functions from profiling both the VMM and the guest

For applications executing in a public cloud, running a PMU-based profiler directly in a guest does not result in useful output, because, as far as we know, none of the current VMMs expose the PMU programming interfaces properly to a guest[1]. As more and more applications are migrated to virtualization-based public clouds, it is necessary to add support for PMU-based profiling in virtual machines, without requiring access to the privileged VMM. Profiling results help customers of public clouds to understand the unique characteristics of their computing environment, identify performance bottlenecks, and fully exploit the hardware and software resources they pay for.

In a private cloud, it is possible to run a profiler directly in the VMM, but some of its intermediate output cannot be converted to meaningful final output without the cooperation of the guest. The data in Table 1 result from profiling a computation-intensive application in the guest. The profiler runs in the VMM, and

[1] It is possible to obtain limited profiling results by running the profiler in the guest in a timer interrupt driven mode.

it monitors CPU cycles. The first row shows that the CPU spends more than 98% of its cycles in the function `vmx_vcpu_run()`, which switches the CPU to run guest code. As the design of the profiler does not consider virtualization, all the instructions consumed by the guest are accounted to this VMM function. Therefore, we cannot obtain detailed profiling data of the guest.

Currently, only XenOprof [10] supports detailed profiling of virtual machines running in Xen, a VMM based on paravirtualization. For VMMs based on hardware assistance and binary translation, no such tools exist. Enabling profiling in the VMM provides users of private clouds and developers of both types of clouds a full-scale view of the whole software stack and its interactions with the hardware. This will help them tune the performance of both the VMM and the applications running in the guest.

In this paper, we address the problem of performance profiling for three different virtualization techniques: paravirtualization, binary translation, and hardware assistance. We categorize profiling techniques in a virtualized environment into two types. *Guest-wide profiling* discloses the runtime characteristics of the guest kernel and its active applications. It only requires a profiler running in the guest, similar to native profiling, i.e., profiling in a non-virtualized environment. The VMM is responsible for virtualizing the PMU hardware, and changes introduced to the VMM are transparent to the guest. *System-wide profiling* discloses the runtime behavior of both the VMM and the active guests. It requires a profiler running in the VMM and provides a full-scale view of the whole software stack: both the VMM and the guest.

The contributions of this paper are as follows. (1) We analyze the challenges of achieving both guest-wide and system-wide profiling for each of the three virtualization techniques. Synchronous virtual interrupt delivery to the guest is necessary for guest-wide profiling. The ability to interpret samples belonging to a guest context into meaningful symbols is required for system-wide profiling. (2) We present profiling solutions for virtualization techniques based on hardware assistance and binary translation. (3) We implement both guest-wide and system-wide profiling for a VMM based on the x86 virtualization extensions and present some preliminary experimental results.

The rest of this paper is organized as follows. In Section 2 we review the structure of a conventional profiler. In Section 3, we analyze the difficulties of supporting guest-wide profiling in a virtualized environment, and present solutions for each of the three aforementioned virtualization techniques. We discuss system-wide profiling in Section 4. In Section 5 we present the implementation of both guest-wide and system-wide profiling for a VMM based on the x86 virtualization extensions and discuss some preliminary experimental results. In Section 6 we survey related work, and we conclude in Section 7.

## 2 Background

As a hardware component, a PMU consists of a set of performance counters, a set of event selectors, and the digital logic to increase a counter after a specified hardware event occurs. When a performance counter reaches a pre-defined threshold, the interrupt controller sends a counter overflow interrupt to the CPU.

Generally, a profiler consists of the following major components:

- *Sampling configuration.* The profiler registers itself as the counter overflow interrupt handler of the operating system, selects the monitored hardware events and sets the number of events after which an interrupt should occur. It programs the PMU hardware *directly* by writing to its registers.
- *Sample collection.* When a counter overflow interrupt occurs, the profiler handles it *synchronously*. In the interrupt context, it records the program counter (PC), the virtual address space of the sampled PC value, and some additional system state.
- *Sample interpretation.* The profiler converts the sampled PC values into routine names of the profiled process by consulting its virtual memory layout and its binary file compiled with debugging information.

Native profiling only involves one OS instance, where all three profiling components reside. They interact with each other through facilities provided by the OS. All of the exchanged information is located in the OS.

In a virtualized environment, multiple OS instances are involved, namely the VMM and the guest(s). The three components of a profiler may be spread among different instances, and their interactions may require communication between them. In addition, not all the conditions necessary for implementing these three components may be satisfied. We present a detailed discussion of implementing both guest-wide and system-wide profiling for each of the three virtualization techniques in the following two sections.

## 3 Guest-wide Profiling

**Challenges.** By definition, guest-wide profiling runs a profiler in the guest and only monitors the guest. Although more information about the whole software stack can be obtained by employing system-wide profiling, sometimes guest-wide profiling is the only way to do performance profiling and tuning in a virtualized environment. As we explained before, users of a public cloud service are not normally granted the privilege to run a profiler in the VMM, which is necessary for system-wide profiling.

To achieve guest-wide profiling, the VMM should provide facilities that enable the implementation of the three profiling components in a guest and PMU multiplexing, i.e., saving and restoring PMU registers. Since sample interpretation under guest-wide profiling is the same as in native profiling and PMU multiplexing is trivial, we only present the required facilities for the sampling configuration and the sample collection components.

To implement the sampling configuration component, the guest should be able to program the physical PMU registers, either directly or through the assistance of the VMM. To implement the sample collection component, the VMM must support *synchronous interrupt delivery* to the guest. In other words, if the VMM injects an interrupt into the guest when it is not running, then that injected interrupt must be handled *immediately* when the guest resumes its execution. For performance profiling, when a performance counter overflows, a physical interrupt is generated and handled by the VMM. If the interrupt occurs when the guest code is executing, the counter overflow is considered to be contributed by the guest. The VMM injects a virtual interrupt into the guest, which drives the profiler to collect a sample. If the guest handles the injected interrupt synchronously when it resumes execution, it can collect a correct sample as in native profiling. If not, at the time when the injected virtual interrupt is handled, the real interrupt context has already been destroyed, and the profiler obtains the wrong sampling information.

**Paravirtualization.** The major obstacle for implementing guest-wide profiling for VMMs based on paravirtualization is synchronous interrupt delivery to the guest. At least in Xen, this functionality is currently not available. External events are delivered to the guest asynchronously. Mechanisms similar to synchronous signal delivery in a conventional OS should be employed to add this capability to paravirtualization-based VMMs.

**Hardware assistance.** The x86 virtualization extensions provide facilities that help implement guest-wide profiling. First, the guest can be configured to allow it to directly access the PMU registers, which are model-specific registers (MSRs) in the x86. Second, the CPU can be configured to automatically save and restore the relevant MSRs. Third, the guest can be configured to exit when an interrupt occurs.The interrupt number is recorded in the exit information field of a control structure. The VMM injects into the guest a virtual interrupt by setting a field of the control structure. Event delivery to a guest is synchronous so the guest profiler samples correct system states. We present our implementation of guest-wide profiling based on the x86 hardware extensions in Section 5.1.

**Binary translation.** For VMMs based on binary translation, synchronous interrupt delivery is also required. Another obstacle is that the sampled PC values point to addresses in the translation cache, not to the memory addresses holding the original instructions. Additional work is required to map the sampled PC values to the original memory addresses. Because only the VMM has enough information to do the translation, without explicit communication between the VMM and the guest, guest-wide profiling is not possible for VMMs based on binary translation. We present more details about this address translation problem and a solution for system-wide profiling in Section 4.

**Where to multiplex the PMU?** Besides the requirements stated previously, another important question is: what are the right places to save and restore the relevant MSRs? The first answer is to save and restore the relevant registers when the CPU switches from running guest code to VMM code, or vice versa. We call this type of guest-wide profiling *CPU switch*. Profiling results of CPU switch reflect the characteristics of the virtualized physical hardware such as CPU and memory, but not the devices emulated by software. This is because the PMU is active only when the guest code is being executed. When the CPU switches to execute the VMM code that emulates the effects of a guest I/O operation, although the monitored hardware events are still being contributed by the guest, they are not accounted to the guest because the PMU is off. The second answer is to save and restore the relevant MSRs when the VMM switches execution from one guest to a different one. We call this *domain switch*. It accounts to a guest all the hardware events triggered by itself and reflects the characteristics of both the virtualized hardware and the VMM. This method may introduce inaccuracy in the profiling results, because the VMM may be interrupted when it is executing on behalf of a guest. Since the registers are only saved and restored on a domain switch, the execution resulting from the interruption will be attributed to that guest. This problem is similar to the resource accounting problem in a conventional operating system [2]. Although sometimes not entirely accurate from the viewpoint of performance profiling, domain switch profiling does give a more complete picture of the overall system. We present the results of guest-wide profiling based on CPU switch and domain switch in Section 5.2.

# 4 System-wide Profiling

**Challenges.** System-wide profiling provides the runtime characteristics of both the VMM and the guests. It first requires that all three components of a profiler run in the VMM. Since the profiler resides in the VMM, it can program the PMU hardware directly and handle the counter overflow interrupts synchronously. The major obstacle for system-wide profiling is to interpret samples not belonging to the VMM, but to the guests. This requires at least the sample interpretation component of a profiler to be present in the guest. As a result, explicit communication between the profiler running the VMM and the sample interpretation component in the guest is required. The interaction rate between the VMM and the guest approaches the rate of counter overflow interrupts. Efficient communication methods, such as zero-copy, should be used to avoid distortions in the profiling results. In addition, the interpretation should be finished in time. Otherwise, if the profiled process terminates before the sample interpretation starts, there will be no clue to finish it because the profiler needs to consult the virtual memory layout of the process to do the interpretation.

**Interpreting guest samples.** One approach to the guest sample interpretation problem is to not let the VMM record samples corresponding to a guest, but delegate this task to the guest. We call this approach *full-delegation*. It requires guest-wide profiling to be already supported by the VMM. With this approach, during the profiling process, one profiler runs in the VMM and another one runs in the guest. The VMM profiler is responsible for collecting and interpreting samples from the VMM. For a sample not belonging to the VMM, a counter overflow interrupt is injected into the corresponding guest. The guest profiler collects and interprets samples from the guest. A system-wide profile can then be obtained by merging the outputs of the VMM and the guest profiler.

An alternative approach to this problem is to let the VMM profiler collect all the samples and delegate the interpretation of guest samples to the corresponding guest [10]. We call this approach *interpretation-delegation*. With this solution, the VMM saves the samples belonging to a guest in a buffer shared with the guest[2]. Every time it adds a sample to a guest's buffer, the VMM signals that guest that there are pending samples. After a guest receives the signal, it notifies its sample interpretation component to interpret the samples, in the same way as a native profiler. The results are sent back to the VMM and merged with those produced by the VMM to obtain a system-wide profile.

**Paravirtualization.** For VMMs based on paravirtualization, system-wide profiling can be implemented by the interpretation-delegation approach, as demonstrated by XenOprof [10]. Xen has a powerful hypercall mechanism, which allows for easy sharing of a communication buffer between the VMM and the guest. The full-delegation approach may also work if the VMM supports guest-wide profiling.

**Hardware assistance.** Since the x86 hardware extensions have the necessary facilities for implementing guest-wide profiling, the full-delegation approach can be employed to achieve system-wide profiling. This approach only requires minor changes to the VMM, as our implementation of system-wide profiling for an open-source VMM in Section 5.1 shows. Similar to XenOprof, system-wide profiling can also be achieved by the interpretation-delegation approach. This requires implementing an efficient communication path between the guest and the VMM, and extending the profilers running both in the VMM and in the guest. Therefore, it involves much more work than the full-delegation approach.

**Binary Translation.** For VMMs based on binary translation, system-wide profiling can only be achieved through the interpretation-delegation approach. Even if the VMM supports synchronous interrupt delivery to the guest, the sampled PC values always point to memory addresses of the translation cache, where the translated instructions reside. Only the VMM has enough information to translate the sampled PC values back to the addresses where the original guest instructions reside. This address translation can be achieved as follows. During the process of an original guest instruction being translated and stored in the translation cache, we save the reverse mapping from the address(es) of one or more translated instructions to the address of the original guest instruction in a *reverse address translation cache*, a counterpart of the translation cache. For each memory address in the translation cache, there is a corresponding entry in the reverse address translation cache, containing the address of the corresponding original guest instruction. For each PC value sampled from a guest context, the VMM is responsible for rewriting it with the original instruction address by looking up the reverse address translation cache. This should be done before the guest profiler reads the sampled PC value. A possible rewriting point is in the virtual interrupt injection procedure. This rewriting process is transparent to the sample interpretation component in the guest.

# 5 Implementation and Evaluation

We describe our extensions to the kernel-based virtual machine (KVM) [8] that implement both guest-wide and system-wide profiling. We present some preliminary ex-

[2] The cooperation of the guest may also be needed. For example, the VMM cannot directly figure out the corresponding virtual address space of the sampled PC value.

perimental results to show the feasibility of implementing guest-wide profiling. We leave the implementation of system-wide profiling for a VMM based on binary translation for future work.

## 5.1 Profiling for KVM

KVM is a Linux kernel subsystem that leverages hardware virtualization extensions to add a virtual machine monitor capability to Linux. With KVM, the VMM is a set of kernel modules in the host Linux operating system while each virtual machine resides in a normal user-space process. Although KVM supports multiple hardware architectures, we choose the x86 with virtualization extensions to illustrate our implementation, because the x86 version of KVM has the most mature code.

The virtualization extensions augment the x86 with two new operation modes: *host mode* and *guest mode*. KVM runs in host mode and its guests run in guest mode. Host mode is compatible with conventional x86 while guest mode is very similar to it but deprivileged in certain ways. Guest mode supports all four privilege levels and allows direct execution of the guest code. The virtual machine control structure (VMCS) controls various behaviors of a virtual machine. Two transitions are defined: a transition from host mode to guest mode called a VM-entry, and a transition from guest mode to host mode called a VM-exit. Regarding performance profiling, if a performance counter overflows when the CPU is in guest mode, the currently running guest is forced to exit, i.e., the CPU switches from guest mode to host mode. The VM-exit information in VMCS indicates that the current VM-exit is caused by a non-maskable interrupt (NMI). By checking this field, KVM is able to decide whether a counter overflow is contributed by a guest. We assume all NMIs in a profiling session are caused by counter overflows. KVM can also decide this by checking the content of all performance counters.

Our guest-wide profiling implementation requires no modifications to the guest OS and the profiler. The profiler reads and writes the physical PMU registers directly as it does in native profiling. KVM forwards NMIs due to performance counter overflows to the guest. When CPU switch is enabled, KVM saves the relevant MSRs when a VM-exit happens and restores them when the corresponding VM resume occurs. This can be done automatically in hardware, by configuring certain fields in the VMCS. When domain switch is enabled, we tag all threads belonging to a guest and group them into one domain. When the Linux (host) kernel switches to a thread not belonging to the current domain, it saves and restores the relevant registers in software.

We implement system-wide profiling for KVM by the full-delegation approach since it is built on hardware virtualization extensions and supports synchronous virtual interrupt delivery in the guest. In a profiling session, we run one unmodified profiler instance in the host and one in each guest. These profiling instances work and cooperate as we discussed in Section 4. The only changes to KVM are clearing the bit in an advanced programmable interrupt controller (APIC) register after each VM-exit and injecting an NMI to a guest when it causes a performance counter overflow.

## 5.2 Experimental Results

We use our guest-wide profiling extensions to KVM to profile a guest in an experiment that measures the TCP receive throughput. The VMM consists of the 2.6.32 Linux kernel with KVM enabled and QEMU [4] 0.11. The guest runs Linux with the 2.6.32 kernel. The profiler is OProfile [9] 0.9.5. Both the guest kernel and the profiler remain unmodified. Our machine is equipped with one Intel Core2 quad-core processor and one Gigabit Ethernet NIC. As much as possible TCP traffic is pushed to the guest from a Gigabit NIC on a different machine.

| % INSTR | Function | Module |
|---|---|---|
| 14.1047 | csum_partial | vmlinux |
| 8.9527 | csum_partial_copy_generic | vmlinux |
| 6.2500 | copy_to_user | vmlinux |
| 3.9696 | ipt_do_table | ip_tables.ko |
| 3.6318 | tcp_v4_rcv | vmlinux |
| 3.2095 | (no symbols) | libc.so |
| 2.8716 | ip_route_input | vmlinux |
| 2.7027 | tcp_rcv_established | vmlinux |

Table 2: Eight functions with the highest number of instruction retirements, with CPU switch enabled.

Table 2 presents the eight functions with the largest number of instruction retirements. In this experiment CPU switch is used for PMU virtualization. The total number of samples is 1184. Most of the functions in the table are from the Linux network stack and carry out packet processing jobs. Table 3 gives the results for the same experiment, but with domain switch enabled for PMU virtualization. The total number of samples is 7286. This shows that more than 80% of the retired instructions involved in receiving packets in the guest are spent outside the guest, inside the device emulation code. This percentage is reasonable since the experiment involves an I/O-intensive application. The top five functions are all related to I/O. The first three are from the RTL8139 NIC driver, and the last two program the APIC. This result shows that the VMM spends a large amount of instructions emulating the effects of the I/O operations in the virtual RTL8139 NIC and the virtual APIC.

| % INSTR | Function | Module |
|---|---|---|
| 31.0321 | cp_interrupt | 8139cp.ko |
| 18.3365 | cp_rx_poll | 8139cp.ko |
| 14.1916 | cp_start_xmit | 8139cp.ko |
| 5.7782 | native_apic_mem_write | vmlinux |
| 5.1331 | native_apic_mem_read | vmlinux |
| 2.6215 | csum_partial | vmlinux |
| 1.4411 | csum_partial_copy_generic | vmlinux |
| 1.2901 | copy_to_user | vmlinux |

Table 3: Eight functions with the highest number of instruction retirements, with domain switch enabled.

The data in both tables demonstrate the feasibility of conducting guest-wide profiling in a virtualized environment based on hardware assistance. Both CPU switch and domain switch are useful, as they show developers different perspectives of how programs behave in a virtualized computing environment.

## 6 Related Work

XenOprof [10] is the first profiler that supports virtual machines. According to our definitions, it does system-wide profiling. It is specifically designed for Xen, a VMM based on paravirtualization.

Xen HVM is a version of Xen that supports hardware-assisted full virtualization. In its support for profiling, Xen HVM only saves and restores MSRs when it performs domain switches. In Xen's architecture, I/O device emulation for all the guests is done in domain 0. VM exits in a domain that do not require the intervention of domain 0 are handled in the context of that domain. As a result, guest-wide profiling in Xen HVM does not accurately reflect the behavior of a guest.

Linux perf [1] is a new implementation of performance counter support for Linux. It runs in the host Linux and can profile a Linux guest running in KVM. Linux perf is similar to system-wide profiling discussed in this paper, but it can only interpret samples belonging to the kernel of the Linux guest, not its user-space applications. This is because Linux perf is aware of the virtual memory layout of the guest Linux kernel, but not of the virtual memory layout of a user process.

VTSS++ [5] demonstrates a profiling technique similar to guest-wide profiling defined in this paper. It requires the cooperation of a profiler running in the guest and a PMU sampling tool running in the VMM. It relies on sampling time stamps to attribute counter overflows to the corresponding threads in the guest. Its profiling results are not completely precise because the limited resolution of the timer interval may result in an inaccurate attribution of a counter overflow to a thread. It also requires that the VMM does not virtualize the processor timestamp counter.

VMware vmkperf [7] is a performance monitoring utility for the VMware ESX server. It runs in the VMM and only records how many hardware events happen in a time interval. It does not handle counter overflow interrupts and attribute them to particular procedures. Therefore, it does not support any of the profiling mechanisms presented in this paper.

## 7 Conclusions

In this paper, we studied the problem of supporting performance profiling in a virtualized environment. We gave solutions that achieve both guest-wide and system-wide profiling for VMMs based on hardware assistance. We also described a solution that implements system-wide profiling for VMMs based on binary translation. In the end, we demonstrated the feasibility of guest-wide and system-wide profiling by implementing them in KVM.

## References

[1] Performance Counters for Linux. 2010. `http://lwn.net/Articles/310176/`.

[2] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. *Operating Systems Review*, 33:45–58, 1998.

[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 *Proceedings of the nineteenth ACM symposium on Operating systems principles*, page 177. ACM, 2003.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] Stanislav Bratanov, Roman Belenov, and Nikita Manovich. Virtual machines: a whole new world for performance analysis. *SIGOPS Oper. Syst. Rev.*, 43(2):46–55, 2009.

[6] Intel Inc. Intel VTune Performance Analyser, 2010. `http://software.intel.com/en-us/intel-vtune/`.

[7] VMware Inc. Vmkperf for VMware ESX 4.0, 2010.

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

[9] J. Levon and P. Elie. Oprofile: A system profiler for linux. 2010. `http://oprofile.sourceforge.net`.

[10] A. Menon, J.R. Santos, Y. Turner, G.J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *VEE*, volume 5, pages 13–23, 2005.

[11] J. Sugerman, G. Venkitachalam, and B.H. Lim. Virtualizing I/O devices on VMware workstations hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14, 2001.