View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Performance Profiling of Virtual Machines

Jiaqing Du
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
jiaqing.du@epfl.ch

Nipun Sehrawat
University of Illinois at Urbana Champaign, USA
sehrawa2@illinois.edu

Willy Zwaenepoel
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
willy.zwaenepoel@epfl.ch

Abstract

Profilers based on hardware performance counters are indispensable for performance debugging of complex software systems. All modern processors feature hardware performance counters, but current virtual machine monitors (VMMs) do not properly expose them to the guest operating systems. Existing profiling tools require privileged access to the VMM to profile the guest and are only available for VMMs based on paravirtualization. Diagnosing performance problems of software running in a virtualized environment is therefore quite difficult.

This paper describes how to extend VMMs to support performance profiling. We present two types of profiling in a virtualized environment: *guest-wide profiling* and *system-wide profiling*. Guest-wide profiling shows the runtime behavior of a guest. The profiler runs in the guest and does not require privileged access to the VMM. System-wide profiling exposes the runtime behavior of both the VMM and any number of guests. It requires profilers both in the VMM and in those guests.

Not every VMM has the right architecture to support both types of profiling. We determine the requirements for each of them, and explore the possibilities for their implementation in virtual machines using hardware assistance, paravirtualization, and binary translation.

We implement both guest-wide and system-wide profiling for a VMM based on the x86 hardware virtualization extensions and system-wide profiling for a VMM based on binary translation. We demonstrate that these profilers provide good accuracy with only limited overhead.

Categories and Subject Descriptors D.4 [Operating Systems]: Performance; C.4 [Performance of Systems]: Performance Attributes

General Terms Performance, Design, Experimentation

Keywords Performance Profiling, Virtual Machine, Hardware-assisted Virtualization, Binary Translation, Paravirtualization

1. Introduction

Profilers based on the hardware performance counters of modern processors are indispensable for performance debugging of complex software systems [21, 4, 23]. Developers rely on profilers to understand the runtime behavior, identify potential bottlenecks, and tune the performance of a program.

Performance counters are part of the processor's performance monitoring unit (PMU). The PMU consists of a set of performance counters, a set of event selectors, and the digital logic to increase a counter after a hardware event specified by the event selector occurs. Typical events include clock cycles, instruction retirements, cache misses, TLB misses, etc. When a performance counter reaches a pre-defined threshold, a counter overflow interrupt is generated.

The profiler selects the event(s) to be monitored, and registers itself as the PMU counter overflow interrupt handler. When an interrupt occurs, it records the saved program counter (PC) and other relevant information. After the program is finished, it converts the sampled PC values to function names in the profiled program, and it generates a histogram that shows the frequency with which each function triggers the monitored hardware event. For instance, Table 1 shows a typical output of the widely used OProfile profiler [17] for Linux. The table presents the eight functions that consume the most cycles in a run of the profiled program.

PMU-based performance profiling in a native computing environment has been well studied. Mature profiling tools built upon PMUs exist in almost every popular operating system [17, 13]. They are extensively used by developers to tune software performance. This is, however, not the case in a virtualized environment, for the following two reasons.

On the one hand, running an existing PMU-based profiler in a guest does not result in useful output, because, as far as we know, none of the current VMMs properly expose the PMU programming interfaces to guests. Most VMMs simply filter out guest accesses to the PMU. It is possible to run a guest profiler in restricted timer interrupt mode, but doing so results in limited profiling results. As more and more applications run in a virtualized environment, it is necessary to provide full-featured profiling for virtual machines. In particular, as applications are moved to virtualization-based public clouds, the ability to profile applications in a virtual machine without the need for privileged access to the VMM allows users of public clouds to identify performance bottlenecks and to fully exploit the hardware resources they pay for.

On the other hand, while running a profiler in the VMM is possible, without the cooperation of the guest its sampled PC values cannot be converted to function names meaningful to the guest application developer. The data in Table 1 result from running a profiler in the VMM during the execution of a computation-intensive application in a guest. 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 the guest. As the design of the profiler does not consider virtualization, all the CPU cycles consumed by the guest are accounted to this function in the VMM . Therefore, we cannot obtain detailed profiling data on the guest. Currently, only XenOprof [18] supports detailed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'11 March 9–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0501-3/11/03…$10.00.

profiling of virtual machines running in Xen [6], 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 and developers of virtualization solutions with a full-scale view of the whole software stack and its interactions with the hardware, helping them to tune the performance of the VMM, the guest, and the applications running in the guest.

% 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
0.0318	x86_decode_insn	kvm.ko
0.0318	vga_update_display	qemu
0.0318	get_call_destination	vmlinux

Table 1. A typical profiler output: the eight functions that consume the most cycles in a run of the profiled program.

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

The main contributions of this paper are:

1. We generalize the problem of performance profiling in a virtualized environment and propose two types of profiling: guest-wide profiling and system-wide profiling.
2. We analyze the challenges of achieving 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 convert samples belonging to a guest context into meaningful function names is required for system-wide profiling.
3. We present profiling solutions for virtualization based on hardware assistance and binary translation.
4. We demonstrate the feasibility and usefulness of virtual machine profiling by implementing both guest-wide and system-wide profiling for a VMM based on the x86 virtualization extensions and system-wide profiling for a VMM based on binary translation.

The rest of the paper is organized as follows. In Section 2 we review the structure and working principles of a profiler in a native environment. In Section 3 we analyze the challenges of supporting guest-wide and system-wide profiling for each of the three aforementioned virtualization techniques. In Section 4 we present the implementation of guest-wide and system-wide profiling in two VMMs, KVM and QEMU. We evaluate the accuracy, usefulness and performance of the resulting profilers in Section 5. In Section 6 we discuss some practical issues related to supporting virtual machine profiling in production environments. We describe related work in Section 7 and conclude in Section 8.

2. Native Profiling

Profiling is a widely used technique for dynamic program analysis. A profiler investigates the runtime behavior of a program as it executes. It determines how much of a hardware resource each function in a program consumes. A PMU-based profiler relies on performance counters to sample system states and figure out approximately how the profiled program behaves. Compared with other profiling techniques, such as code instrumentation [22, 12], PMU-based profiling provides a more accurate picture of the target program's execution as it is less intrusive and introduces fewer side effects.

The programming interface of a PMU is a set of performance counters and event selectors. When a performance counter reaches the pre-defined threshold, a counter overflow interrupt is generated by the interrupt controller and received by the CPU. Exploiting this hardware component for performance profiling, a PMU-based profiler generally 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*. The profiler records the saved PC, the event type causing the interrupt, and the identifier of the interrupted process under the counter overflow interrupt context. The interrupt is handled by the profiler synchronously.
- *Sample interpretation*. The profiler converts the sampled PC values into function names of the profiled process by consulting its virtual memory layout and its binary file compiled with debugging information.

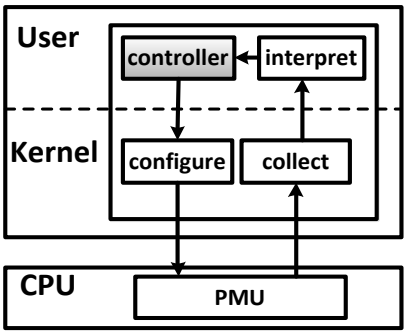

Figure 1. Block diagram of a native PMU-based profiler.

In a native environment, all three profiling components and their data structures reside in the operating system. They interact with each other through facilities provided by the operating system. Figure 1 shows a block diagram of a PMU-based profiler in a native environment.

In a virtualized environment, the VMM sits between the PMU hardware and the guests. The profiler's three components may be spread among the VMM and the guests. Their interactions may require communications between the VMM and the guests. In addition, the conditions for implementing these three components may not be satisfied in a virtualized environment. For instance, the sampling configuration component of a guest profiler may not be able to program the PMU hardware because of the interposition of the VMM. In the next section, we present a detailed discussion of the requirements for guest-wide and system-wide profiling for

virtualization based on hardware extensions, paravirtualization and virtualization based on binary translation.

3. Virtual Machine Profiling

3.1 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 in a virtualized environment. As we explained before, users of a public cloud service are normally not granted the privilege to run a profiler in the VMM, which is necessary for conducting system-wide profiling. To achieve guest-wide profiling, the VMM needs to provide PMU multiplexing, i.e., saving and restoring PMU registers, and enable the implementation of the three profiling components in the guest. Since sample interpretation in guest-wide profiling is the same as in native profiling, we only present here the required facilities for sampling configuration and sample collection. We return to the topic of PMU multiplexing in Section 3.3.

To implement sampling configuration, the guest must be able to program the PMU registers, either directly or with the assistance of the VMM.

To achieve sample collection, the guest must be able to collect correct samples under interrupt contexts, which requires that the VMM supports synchronous interrupt delivery to the guest. This means that, if the VMM injects an interrupt into a guest, that injected interrupt is handled first when the guest resumes its execution. For performance profiling, when a performance counter overflows, an interrupt is generated and first handled by the VMM. If the interrupt is generated when the guest code is being executed, 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 take a sample. If the guest handles the injected interrupt synchronously when it resumes execution, it collects correct samples 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 wrong sampling information.

Hardware assistance The x86 virtualization extensions provide facilities that help implement guest-wide profiling. First, the guest can be configured with direct access to the PMU registers, which are model-specific registers (MSRs) in the x86. The save and restore of the relevant MSRs can also be done automatically by the CPU. Second, the guest can be configured to exit when interrupts occur. The hardware guarantees that event delivery to a guest is synchronous, so the VMM can forward to the guest all counter overflow interrupts contributed by it, and the guest profiler samples correct system states. We present our implementation of guest-wide profiling for virtualization based on the x86 hardware extensions in Section 4.1.

Paravirtualization The major obstacle of implementing guest-wide profiling for VMMs based on paravirtualization is synchronous interrupt delivery to the guest. At least for Xen, this facility 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. For sampling configuration, as a paravirtualized guest runs in a deprivileged mode and cannot access the PMU hardware, the VMM must implement the necessary programming interfaces to allow the guest to program the PMU indirectly.

Binary translation For VMMs based on binary translation, sampling configuration can be achieved with the assistance of the VMM, which is able to identify instructions that access the PMU and rewrite them appropriately. Similar to paravirtualization, synchronous interrupt delivery to the guest is an engineering challenge. As far as we know, no VMMs based on binary translation support this feature.

3.2 System-wide Profiling

Challenges System-wide profiling reveals the runtime characteristics of both the VMM and the guests. It first requires that all three components of the 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 challenges for system-wide profiling come from the other two profiling components.

The first challenge for system-wide profiling comes from sample collection. If the counter overflow is triggered by a guest user process, the VMM profiler cannot obtain the identity of this process without the assistance of the guest operating system. This information is described by a global variable in the guest kernel, and the VMM does not know the internal memory layout of the guest. To solve this problem, the guest must share this piece of information with the VMM profiler.

The second challenge is interpreting samples belonging to the guests. Even if the VMM profiler is able to sample all the required information, sample interpretation is not possible because the VMM does not know the virtual memory layout of the guest processes or the guest kernel. This requires the guest to interpret its samples, which means that at least the sample interpretation component of a profiler should run in the guest.

One approach that addresses both the sample collection and the sample interpretation problem is to not let the VMM record samples corresponding to a guest, but to delegate this task to the guest. We call this approach *full-delegation*. It requires guest-wide profiling to be supported by the VMM. With this approach, during the profiling process, one profiler runs in the VMM and one runs in each guest. The VMM profiler is responsible for handling all counter overflow interrupts, but it only collects and interprets samples contributed by the VMM. For a sample not belonging to the VMM, a counter overflow interrupt is injected into the corresponding guest. The guest profiler is driven by the injected interrupts to collect and interpret samples contributed by the guest. Overall system-wide profiling results are obtained by merging the outputs of the VMM profiler and the guests.

An alternative solution is to let the VMM profiler collect all samples and to delegate the interpretation of guest samples to the corresponding guest [18]. We call this approach *interpretation-delegation.* With this solution, the guest makes the identity of the process to be run available to the VMM profiler. When a counter overflows, the VMM records the saved PC, the event type, and the identifier of the interrupted guest process, and sends it to the guest. After the guest receives the sample, it notifies the sample interpretation component of its profiler to convert the sample to a function name, in the same manner as a native profiler. After the profiling finishes, the results recorded in the guests are merged with those produced by the VMM profiler to obtain a system-wide view.

The interpretation-delegation approach for system-wide profiling requires explicit communication between the VMM and the guest. Their interaction rate approaches the rate of counter overflow interrupts, which can go up to thousands of times per second with a normal profiling configuration. Efficient communication methods should be used to avoid distortions in the profiling re-

sults. We choose to use a buffer shared among all the profiling participants for exchanging information. In addition, a guest should process samples collected for it in time. Otherwise, if a process terminates before the samples contributed by it are interpreted, there will be no way to interpret these samples because the sample interpretation component needs to consult the virtual memory layout of this process.

Similar to full-delegation, interpretation-delegation can also be implemented by running one profiler in the VMM and one in each guest. However, the guest profiler does not need to access the PMU hardware. It only processes samples in the shared buffer, which are collected for it by the VMM profiler, by running its sample interpretation component when handling a virtual interrupt injected by the VMM profiler.

The choice between full-delegation and interpretation-delegation to implement system-wide profiling depends on whether synchronous interrupt delivery is supported by the VMM. If so, full-delegation is the preferred approach. If not, one should either choose the interpretation-approach or add the support of synchronous interrupt delivery to the VMM. Full delegation requires less engineering effort and is transparent to the profiled guest. Implementing synchronous interrupt delivery to the guest in software is, however, not trivial, and current VMMs based on paravirtualization and binary translation do not support this feature. Therefore, we choose the full-delegation approach to implement system-wide profiling for a VMM based on hardware assistance and the interpretation-delegation approach for a VMM based on binary translation (see Section 4).

Hardware assistance For VMMs based on hardware extensions, since they have all the facilities to implement 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 4.1 shows. System-wide profiling can also be achieved by the interpretation-delegation approach. An efficient communication path between the guest and the VMM and extensions to the profilers running both in the VMM and in the guest require more engineering work than the full-delegation approach.

Paravirtualization For VMMs based on paravirtualization, system-wide profiling can be implemented by interpretation-delegation. XenOprof uses this approach to perform system-wide profiling in Xen. Its implementation requires less engineering effort than in VMMs based on hardware assistance or binary translation, because Xen provides the hypercall mechanism that facilitates interactions between the VMM and the guest. The full-delegation approach may also work if the VMM supports guest-wide profiling.

Binary Translation For VMMs based on binary translation, system-wide profiling can be achieved through the interpretation-delegation approach. If the VMM supports synchronous interrupt delivery to the guest, the full-delegation approach also works.

When using interpretation-delegation, VMMs based on binary translation need to solve the following additional problem. If the execution of a guest triggers a counter overflow, the PC value sampled by the VMM profiler points to a translated instruction in the translation cache, not to the original instruction. Additional work is required to map the sampled PC back to the guest address where the original instruction is located. This address translation problem can be solved by the following idea. During the translation of guest instructions, we save the mapping from the address(es) of one or more translated instructions to the address of the original guest instruction in the *address mapping cache*, a counterpart of the translation cache. For each memory address of the translation cache, there is an entry in the address mapping cache, which points to the address holding the original guest instruction. For samples from a guest context, rather than storing the PC value itself, the VMM looks up the original instruction address in the address mapping cache and stores that address as part of the sample. This rewriting of the sampled PC value is transparent to the sample interpretation component in the guest.

3.3 PMU Multiplexing

Besides the requirements stated previously, another important question for both guest-wide and system-wide profiling is: what is the right time to save and restore PMU registers?

The first option is to save and restore these registers when the CPU switches between running guest code and running VMM code. We call this type of profiling *CPU switch*. Profiling results with CPU switch reflect the execution of the guest on the virtualized CPU, but not the guest's use of devices emulated by software. 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 effectively being contributed by the guest, they are not accounted to it. In the case of guest-wide profiling, the PMU is turned off, and in the case of system-wide profiling the events are accounted to the VMM.

The second option is to save and restore relevant registers when the VMM switches execution from one guest to another. We call this *domain switch*. This method accounts to a guest all the hardware events triggered by its execution and by the execution of the VMM while emulating devices on its behalf. In other words, domain switch PMU multiplexing reflects the characteristics of the entire virtual environment, including both the virtualized hardware and the virtualization software.

Guest-wide and system-wide profiling can choose either of the two approaches for PMU multiplexing. Generally, domain switch provides a more realistic picture of the underlying virtual environment. We compare the profiling results of these two methods in Section 5.

4. Implementation

We describe the implementation of both guest-wide and system-wide profiling for the kernel-based virtual machine (KVM) [16], a VMM based on hardware assistance. We also present how system-wide profiling is implemented in QEMU [7], a VMM based on binary translation. As both KVM and QEMU are considered as hosted VMMs, we use the terms "VMM" and "host" interchangeably in this section.

Our implementation follows two principles. First, performance profiling should introduce as little as possible overhead to the execution of the guest. Otherwise, the monitoring results would be perturbed by monitoring overhead. Second, performance profiling should generate as little as possible performance overhead for the VMM. It should not slow down the whole system too much. To achieve these goals, we only introduce additional switches between the VMM and the guest when absolutely necessary. For all our implementations, except during the profiling initialization phase, only virtual interrupt injection into the guest causes additional context switches, but these are inevitable for PMU-based performance profiling.

4.1 Hardware Assistance

KVM is a Linux kernel infrastructure which leverages hardware virtualization extensions to add a virtual machine monitor capability to Linux. With KVM, the VMM is a kernel module in the host Linux, 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 it has the most mature code.

The virtualization extensions augment 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. A virtual machine control structure (VMCS) is introduced to control various behaviors of a virtual machine. Two transitions are also 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 filed in the 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. This approach assumes all NMIs are caused by counter overflows in a profiling session. To be more precise, KVM could also check the content of all performance counters to make sure that NMIs are really caused by counter overflows.

Our guest-wide profiling implementation requires no modifications to the guest and its profiler. The guest profiler reads and writes the physical PMU registers directly as it does in native profiling. KVM is responsible for virtualizing the PMU hardware and forwarding NMIs due to performance counter overflows to the guest. A user can launch the profiler from the guest and do performance profiling exactly as in a native environment.

We implement system-wide profiling by the full-delegation approach, since KVM is built upon 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 3.2. The only changes to KVM are clearing the bit in an APIC register after each VM-exit (see below) and injecting NMIs into a guest that causes a performance counter overflow.

When CPU switch is enabled, KVM saves all the relevant MSRs when a VM-exit happens and restores them when the corresponding VM-resume occurs. By configuring certain fields in the VMCS, this is done automatically in hardware. When domain switch is enabled, we tag all (Linux kernel) threads belonging to a guest and group them into one domain. When the Linux kernel switches to a thread not belonging to the current domain, it saves and restores the relevant registers (in software).

In the process of implementing these two profiling techniques in KVM, we also observe the following two noteworthy facts. First, in the x86 architecture, there is one bit of a register in the Advanced Programmable Interrupt Controller (APIC) that specifies the delivery of NMIs due to performance counter overflows. Clearing this mask bit enables interrupt delivery and setting it inhibits delivery. After the APIC sends a counter overflow NMI to the CPU, it automatically sets this bit. To allow subsequent NMIs to be delivered, a profiler should clear this bit after it handles each NMI. In theory, exposing the register containing this bit to the guest would require the virtualization of the APIC. However, the current implementation of KVM does not virtualize the APIC, but emulates it in software. To bypass this problem, we simply clear the bit after each VM-exit, no matter whether the exit is caused by a performance counter overflow or not.

Second, for guest-wide profiling with CPU switch, we find that the CPU receives NMIs due to counter overflows in host mode, typically right after a VM-exit. For guest-wide profiling, however, performance monitoring is only enabled in guest mode, and NMIs due to performance counter overflows are not supposed to happen in host mode. We could not with 100% certainty determine the reason for this problem because of the lack of a hardware debugger. One plausible explanation is that the VM-exit operation is not "atomic". It consists of a number of sub-operations, including saving and restoring MSRs. A counter may overflow during the execution of VM-exit, but before performance monitoring is disabled. The corresponding NMI is not generated immediately, because the instruction executing when an NMI is received is completed before the NMI is generated [15]. The NMI due to a performance counter overflow in the middle of a VM-exit is thus generated after the VM-exit operation finishes, when the processor is in host mode. We solve this problem by registering an NMI handler in the host to catch those host counter overflows and inject the corresponding virtual NMIs into the guest.

4.2 Binary Translation

We present the implementation of system-wide profiling by the interpretation-delegation approach in QEMU, a VMM based on binary translation. In this environment, the guest profiler runs in the guest kernel space; the guest runs in QEMU; QEMU runs in the user space of the host; and the host profiler runs in the host kernel space. The implementation takes more engineering effort than that of the full-delegation approach in KVM. The implementation touches a number of major components in the whole software stack, including the host, the host profiler, the guest, and the guest profiler.

The conventional x86 is not virtualizable because some instructions do not trap when executed in the unprivileged mode. Dynamic binary translation solves this problem by rewriting those problematic instructions on the fly. A binary translator processes a basic block of instructions at a time. The translated code is stored in a buffer called the translation cache, which holds a recently used subset of all the translated code. Instead of the original guest code, the CPU actually executes the code in the translation cache.

Virtual interrupts injected to a guest are delivered asynchronously in QEMU. Once a virtual interrupt injection request is received, QEMU first unchains the translated basic block being executed and forces the control back to itself after this basic block finishes execution. It then sets a flag of the guest virtual CPU to indicate the reception of an interrupt. The injected interrupt is handled when the guest resumes execution.

To reduce the VM exit rate due to information exchange among the guest, QEMU, and the host, we design an efficient communication mechanism for interpretation-delegation. This is important because in a typical profiling session interactions among all these participants can occur at the rate of thousands of times per second. If each interaction involves one VM exit, the profiling results would be polluted and far from being accurate. The key data structure underlying this communication mechanism is a buffer shared among the three participants. All the critical information in a profiling session, such as the PCs and pointers to process descriptors, is exchanged through this buffer. Each profiling participant reads the buffer directly whenever it needs any information and no VM exits are triggered.

The shared buffer is allocated in the guest and shared through the following control channel. The guest exchanges information with QEMU through a customized virtual device added to QEMU and the corresponding device driver in the guest kernel. QEMU and the host kernel talk with each other through common us-

er/kernel communication facilities provided by the host. After the address of the buffer is passed from the guest profiler to the host profiler, the guest profiler accesses the shared buffer by an address in the guest kernel space; QEMU uses this buffer through an address in its own address space; the host profiler accesses it with an address in the host kernel space.

In our implementation, QEMU is responsible for rewriting the PC value sampled by the host profiler into an address of the guest pointing to the original guest code. To reduce the overall overhead of PC value rewriting, the address mapping cache proposed in Section 3.2 does not map the host address of each instruction in the translation cache to its corresponding guest address. Instead, the cache only maintains one entry for one basic block. All the addresses of the instructions in a translated basic block are mapped to the starting address of the corresponding original basic block in the guest. This does not hurt the accuracy of performance profiling with functions as the interpretation granularity because of two reasons. First, a profiler always interprets any address pointing to the body of a function to the name of that function. Second, a basic block does not span across more than one function, because it terminates right after the first branch instruction.

Putting all the pieces together, the process of system-wide profiling for binary translation can be described as follows.

1. In the initialization phase, the host profiler is loaded to the host kernel, and the guest profiler is loaded to the guest kernel. A communication channel across all the profiling participants is established and a buffer is shared among them.
2. The user starts a profiling session by launching the host profiler. The host profiler sends a message to the guest to start the guest profiler.
3. When profiling is being conducted, the guest records the address pointing to the descriptor of each process right before it schedules the process to run. There is only one entry in the shared buffer for this address. The guest keeps overwriting this entry because it is only useful when the execution of the corresponding process triggers a performance counter overflow. When a counter overflows and if it is contributed by the guest, the host profiler copies the sampled PC value, the event type, and the address to the descriptor of the corresponding guest process to a sampling slot in the shared buffer. It then sends a signal to QEMU running in user space. After the counter overflow NMI is handled and QEMU is scheduled to run again, the signal from the host profiler is delivered first. The signal handler rewrites the sampled PC value, records the current privilege mode of the virtual CPU in the same sampling slot, and injects an NMI into the guest. Upon handling the injected NMI, the guest profiler processes all the available sampling slots one by one.
4. The user finishes the profiling session by stopping the host profiler. The host profiler sends a message to the guest to stop the guest profiler. The output of the host profiler and the guest profiler is merged together as the final profiling results.

Because the host knows little about the internals of a guest and the guest code is dynamically translated, the host profiler can only obtain limited runtime information about the guest under an NMI context. Both the guest and QEMU are required to help record or process sampling information on behalf of the host profiler. This leads to changes to all the participants involved in system-wide profiling based on interpretation-delegation.

5. Evaluation

We first verify the accuracy of our profilers by comparing the results of native profiling with profiling in various virtualized environments. We then show how guest-wide profiling can be used to profile two guests simultaneously. Next, by comparing the results of CPU switch and domain switch for guest-wide profiling, we show that domain switch sometimes provides considerably more information about the guest's execution. We also demonstrate the power of guest-wide profiling by using it on a couple of examples to explain why one virtualization technique performs better than the other. Finally, we quantify the overhead of our profilers by comparing the execution time of a computation-intensive program with and without profiling.

Our experiments involve two machines. The first one is a Dell OptiPlex 745 desktop with one dual-core Intel Core2 E6400 processor, 2GB DDR2 memory, and one Gigabit Ethernet NIC. The second machine is a Sun Fire x4600 M2 server with four quad-core AMD Opteron 8356 processors, 32GB DDR2 memory, and a dozen of Gigabit Ethernet NICs. Unless explicitly stated, all the experiments are conducted on the Intel machine.

For hardware-assisted virtualization, the VMM consists of the 2.6.32 Linux kernel with the KVM kernel module loaded and QEMU 0.11 in user space [7]. For virtualization based on binary translation, the VMM is QEMU 0.10.5 in user space, which runs on top of the 2.6.30 Linux kernel with virtualization extensions disabled. All guests are configured with one virtual CPU and run Linux with the 2.6.32 kernel. The profiler is OProfile 0.9.5.

For both guest-wide and system-wide profiling, CPU switch for KVM adds 170 lines of C code to the Linux kernel, while domain switch consists of 272 lines of C code. QEMU system-wide profiling with CPU switch introduces 1115 lines of C code to QEMU, the host kernel, and the guest kernel.

5.1 Computation-intensive Workload

To verify the accuracy of our profilers for computation-intensive workloads, we use the code given in Figure 2 as the profiled application, and we compare the output of native profiling with virtualized profiling. The program consists of an infinite loop executing two computation-intensive functions `compute_a()` and `compute_b()`, which perform floating point arithmetic and consume different number of CPU cycles. We run this program in two different processes, `process1` and `process2`. We launch both processes at the same time and pin them to one CPU core.

```
int main(int argc, char *argv[])
{
    while (1) {
        compute_a();
        compute_b();
    }
}
```

Figure 2. Code used to verify the accuracy of VM profilers for computation-intensive programs.

Table 2 to Table 5 present the output of profiling runs of this program in which we measure the number of CPU cycles consumed, for native profiling (Table 2), guest-wide profiling in KVM (Table 3), system-wide profiling in KVM (Table 4), and system-wide profiling in QEMU (Table 5)[1]. The results for the

[1] For system-wide profiling, only CPU cycles consumed in the guest are counted in the percentages.

VM profilers are roughly the same as those for the native profiler. As expected, the two processes, `process1` and `process2`, consume roughly the same number of cycles, and the ratio between cycles consumed in `compute_a()` and in `compute_b()` is also roughly similar.

% CYCLE	Function	Module
40.3463	compute_a	process2
38.2010	compute_a	process1
10.6135	compute_b	process2
10.2371	compute_b	process1
0.1505	vsnprintf	vmlinux
0.1129	(no symbols)	bash
0.0376	(no symbols)	libc.so
0.0376	mem_cgroup_read	vmlinux

Table 2. % of cycles consumed in two processes running the program given in Figure 2**,** native profiling.

% CYCLE	Function	Module
38.8114	compute_a	process1
38.5913	compute_a	process2
10.3815	compute_b	process2
10.0880	compute_b	process1
0.5503	native_apic_mem_write	vmlinux
0.2201	(no symbols)	libc.so
0.2201	schedule	vmlinux
0.1101	(no symbols)	bash

Table 3. % of cycles consumed in two processes running the program given in Figure 2**,** KVM guest-wide profiling.

% CYCLE	Function	Module
39.9220	compute_a	process1
39.4209	compute_a	process2
10.3563	compute_b	process2
10.0223	compute_b	process1
0.0557	__switch_to	vmlinux
0.0557	ata_sff_check_status	vmlinux
0.0557	run_time_softirq	vmlinux
0.0557	update_wall_time	vmlinux

Table 4. % of cycles consumed in two processes running the program given in Figure 2, KVM system-wide profiling.

% CYCLE	Function	Module
40.0000	compute_a	process2
36.2963	compute_a	process1
9.2593	compute_b	process1
8.5185	compute_b	process2
0.7407	update_wall_time	vmlinux
0.3704	__schedule	vmlinux
0.3704	__tasklet_hi_schedule	vmlinux
0.3704	cleanup_workqueue_thread	vmlinux

Table 5. % of cycles consumed in two processes running the program given in Figure 2, QEMU system-wide profiling**.**

These results are further confirmed by Figure 3, which shows the average and the standard deviation of the percentage of CPU cycles consumed by `compute_a()` and `compute_b()` over 10 runs with all four profilers. Our profilers provide stable results, with standard deviations ranging from 0.44% to 2.87%. Native profiling has the smallest variance and system-wide profiling for QEMU has the largest.

Figure 4 shows the results of simultaneous guest-wide profiling of two KVM guests running the program described in Figure 2. The percentage of CPU cycles consumed by each function is the same in both guests, and similar to the percentage for each function for native profiling, indicating the accuracy of our profiler when used with multiple virtual machines.

Although the data in this experiment do not constitute "proof of correctness", they give us reasonable confidence that our design and implementation work reasonably well in terms of CPU cycles. We obtain similar results for instruction retirements.

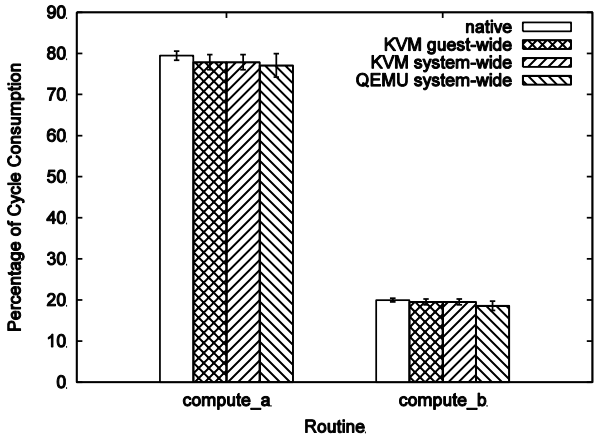

Figure 3. Average and standard deviation of the percentage of cycles consumed by `compute_a()` and `compute_b()`.

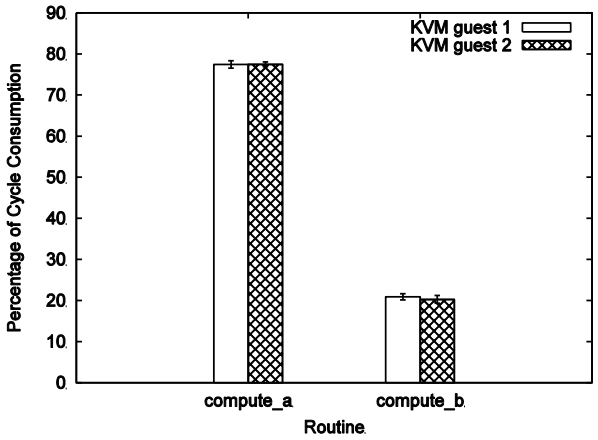

Figure 4. Average and standard deviation of the percentage of cycles consumed by `compute_a()` and `compute_b()` in two KVM guests, KVM guest-wide profiling.

5.2 Memory-intensive Workload

We use the program described in Figure 5 to demonstrate the operation of the KVM guest-wide profiling with memory-intensive programs. This program makes uniformly distributed random accesses to a fixed-size region of memory. We run this program with a working set of 512KB for which the entire execution fits in the L2 cache, and with a working set of 2048KB that causes many misses in the L2 cache.

Table 6 presents the profiling results for L2 cache misses for the 512KB working set, and Table 7 presents the results for the 2048KB working set. The results clearly reflect the higher number of L2 misses with the larger working set.

```
struct item {
    struct item *next;
    long pad[NUM_PAD];
}

void chase_pointer()
{
    struct item *p = NULL;
    p = &randomly_connected_items;
    while (p != null) p = p->next;
}
```

Figure 5. Code used to verify the accuracy of VM profilers for memory-intensive programs.

L2 Miss	Function	Module
1250	chase_pointer	cache_test
100	(no symbols)	bash
100	(no symbols)	ld.so
100	(no symbols)	libc.so
50	sync_buffer	oprofile.ko
50	do_notify_resume	vmlinux
50	do_wp_page	vmlinux
50	find_first_bit	vmlinux

Table 6. L2 cache misses (in thousands) for the program given in Figure 5 with a working set of 512KB, KVM guest-wide profiling.

L2 Miss	Function	Module
150750	chase_pointer	cache_test
2050	native_apic_mem_write	vmlinux
250	idle_cpu	vmlinux
250	run_posix_cpu_timers	vmlinux
200	account_user_time	vmlinux
200	unmap_vmas	vmlinux
200	update_curr	vmlinux
150	do_timer	vmlinux

Table 7. L2 cache misses (in thousands) for the program given in Figure 5 with a working set of 2048KB, KVM guest-wide profiling.

Figure 6 shows the average number of L2 cache misses triggered by one pointer access of our memory-intensive benchmark. We run the benchmark with different working set sizes in four different computing environments. For system-wide profiling of both KVM and QEMU, we only count the cache misses reported in the guest profiler. The number of cache misses per pointer access for native Linux, the KVM guests, and the QEMU guest follow a similar pattern. After the size of the working set exceeds a certain value, the amount of L2 cache available for the benchmark, the miss rate increases dramatically. For native Linux and KVM, the available L2 cache is about 1024 KB. For QEMU, it is 512 KB, because QEMU involves the execution of more software components, such as the binary translator and the MMU emulation code. The cache miss rate after these points grows linearly with the working set size.

5.3 CPU Switch vs. Domain Switch

For guest-wide profiling, there are two possible places to save and restore the registers related to profiling. CPU switch saves and restores the relevant registers when the CPU switches from running guest code to VMM code, or vice versa. Domain switch does this when the VMM switches execution from one guest to another one.

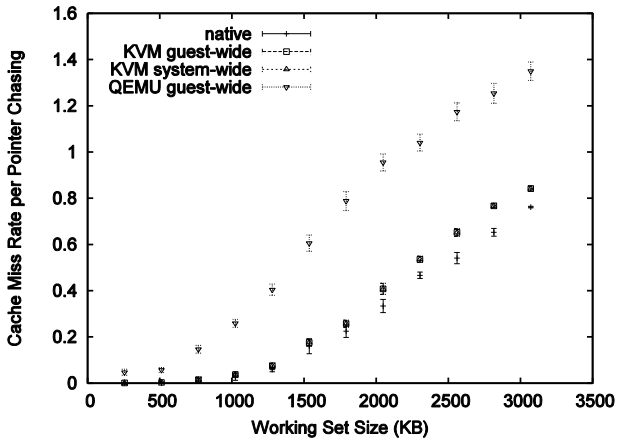

Figure 6. The number of L2 cache misses per pointer access for different working set sizes in four computing environments.

To show the difference between CPU switch and domain switch, we use guest-wide profiling for KVM on a guest that receives TCP packets. In the experiment, as much TCP traffic as possible is pushed to the guest from a Gigabit NIC on a different machine. The virtual NIC used by the guest is RTL8139.

% 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_rc	vmlinux
3.2095	(no symbols)	libc.so
2.8716	ip_route_input	vmlinux
2.7027	tcp_rcv_established	vmlinux

Table 8. Instruction retirements for TCP receive in a KVM guest, guest-wide profiling with CPU switch.

% 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 9. Instruction retirements for TCP receive in a KVM guest, guest-wide profiling with domain switch.

Table 8 presents the eight functions with the largest number of instruction retirements with CPU switch, and Table 9 with domain switch. The total number of samples with CPU switch is 1184 vs. 7286 with domain switch. In other words, more than 80% of the retired instructions involved in receiving packets in the guest are spent outside the guest, inside the device emulation code of the VMM. The VMM spends a large number of instructions emulating the effects of the I/O operations in the virtual RTL8139 NIC and the virtual APIC. The top three functions in Table 9 are from the RTL8139 NIC driver, and the next two program the APIC.

Only below those five appear the three guest functions appearing at the top in Table 8.

This example clearly shows that domain switch can provide more complete information than CPU switch for I/O intensive programs.

5.4 The Power of Guest-wide Profiling

One of the advantages of guest-wide profiling is that it does not require access to the VMM. Nevertheless, it allows advanced performance debugging, as we demonstrate next with the following two examples.

We first profile the benchmark described in Figure 7 to show why hardware-supported nested paging [8] provides more efficient memory virtualization than shadow page tables [19]. This program is a familiar UNIX kernel micro-benchmark that stresses process creation and destruction.[2]

```
int main(int argc, char *argv[])
{
    for (int i = 0; i < 32000; i++) {
        int pid = fork();
        if (pid < 0) return -1;
        if (pid == 0) return 0;
        waitpid(pid);
    }
    return 0;
}
```

Figure 7. A micro-benchmark that stresses process creation and destruction [3].

CYCLE	Function	Module
1300	do_wp_page	vmlinux
1100	do_wait	vmlinux
750	page_fault	vmlinux
400	get_page_from_freelist	vmlinux
400	wait_consider_task	vmlinux
350	unmap_vmas	vmlinux
200	flush_tlb_page	vmlinux
200	native_flush_tlb_single	vmlinux

Table 10. Cycles (in millions) consumed in the program given in Figure 7 in a KVM guest with nested paging, KVM guest-wide profiling.

CYCLE	Function	Module
5450	native_set_pmd	vmlinux
5350	do_wp_pge	vmlinux
3500	native_flush_tlb_single	vmlinux
3050	get_page_from_freelist	vmlinux
2650	schedule	vmlinux
1100	native_flush_tlb	vmlinux
1050	do_wait	vmlinux
950	page_fault	vmlinux

Table 11. Cycles (in millions) consumed in the program given in Figure 7 in a KVM guest with shadow page tables, KVM guest-wide profiling.

Running natively, we measure 4.97 seconds to create and destroy 32000 processes. With nested paging, the guest takes 5.52 seconds, slightly slower than at native speed. When using shadow page tables, the execution time grows to 20.06 seconds. By profiling the benchmark in the guest, a developer can easily figure out which operations involved in process creation and destroying are expensive.

Table 10 presents the eight functions that consume the most CPU cycles with nested paging, and Table 11 presents the same results for shadow page tables. By comparing the profiling results presented in these two tables, we observe that operations related to page table manipulation, such as `native_set_pmd()` and `do_wp_page()`, become more expensive with shadow page tables. The shadow page table mechanism causes a large number of VM exits, including when loading and updating a page table in the guest, when accessing protected pages, and when performing privileged operations like TLB flushing. With nested paging, most of these operations do not cause a VM exit.

Our second example is again TCP receive, similar to the experiment described in Section 5.3. The difference in this experiment is that, instead of RTL8139, we use the E1000 virtual NIC and a virtual NIC based on VirtIO [20]. VirtIO is a paravirtualized I/O framework that provides good I/O performance for virtual machines.

% INSTR	Function	Module
25.2399	e1000_intr	e1000.ko
16.8906	e1000_irq_enable	e1000.ko
12.1881	e1000_xmit_frame	e1000.ko
4.6065	native_apic_mem_write	vmlinux
4.4146	csum_partial	vmlinux
3.3589	e1000_alloc_rx_buffers	e1000.ko
3.2630	native_apic_mem_read	vmlinux
3.0710	__copy_user_intel	vmlinux

Table 12. Instruction retirements for TCP receive in KVM guest with E1000 virtual NIC, guest-wide profiling with domain switch.

% INSTR	Function	Module
52.3312	native_safe_halt	vmlinux
7.7244	native_apic_mem_write	vmlinux
6.6806	csum_partial_copy_generic	vmlinux
1.8903	native_write_cr0	vmlinux
1.4614	ipt_do_table	ip_tables.ko
0.9047	(no symbols)	libc.so
0.9047	get_page_from_freelist	vmlinux
0.9047	schedule	vmlinux

Table 13. Instruction retirements for TCP receive in KVM guest with VirtIO virtual NIC, guest-wide profiling with domain switch.

Table 12 presents the profiling results of packet receive through the E1000 virtual NIC in a KVM guest. Similar to the results of RTL8139, interrupt handling functions retire more than 40% of all instructions, because of the high network I/O interrupt rate. Table 13 presents the results of the VirtIO-based NIC. The function `native_safe_halt()` retires more than half of all instructions, but this function actually executes the `HLT` instruction, which halts the CPU until the next external interrupt occurs. The frequent execution of this instruction in the guest shows that the guest is not saturated while handling 1Gbps TCP traffic. Compared with the data in Table 12, we do not find a single function related to interrupt handling, which indicates that the interrupt rate due to network I/O is low. Our profiling results validate the design of VirtIO, which improves virtualized I/O performance by batching I/O operations to reduce the number of VM exits.

Therefore, as these two experiments demonstrate, guest-wide profiling with domain switch helps developers understand the

[2] This experiment is conducted on our AMD machine, because KVM's modular implementation on AMD CPUs can easily be switched between shadow page tables and hardware-supported nested paging.

underlying virtualized environment without the need for accessing the VMM.

5.5 Profiling QEMU

With our system-wide profiling extensions for QEMU, we profile TCP receive of both the host and the guest. The experiment configuration is similar to the one described in Section 5.4. Instead of KVM, we use QEMU running in user space as the VMM. The virtual NIC is based on VirtIO. The observed TCP receive throughput is around 50MB/s. This amount of traffic saturates the physical CPU but does not keep the virtual CPU of the guest busy.

Table 14 presents the profiling results of the host part. Function `cpu_x86_exec()` retires a large portion of all the instructions. Its functionality is similar to `vmx_vcpu_run()` in KVM, which switches the CPU from the host context to the guest context.

Table 15 shows the results of the guest part, which is obtained by running a customized OProfile in the guest. The appearance of function `__schedule()` indicates that the virtual CPU is not saturated. The reason why function `strcat()` retires most instructions in the guest may be that the corresponding translated native code of this operation is expensive.

% INSTR	Function	Module
68.9548	cpu_x86_exec	qemu
6.0842	__ldl_mmu	qemu
4.2902	helper_cc_compute_c	qemu
1.7161	cpu_x86_handle_mmu_fault	qemu
1.7161	phys_page_find_alloc	qemu
1.4041	ld_phys	qemu
1.2480	tlb_set_page_exec	qemu
0.6240	helper_cc_compute_all	qemu

Table 14. Instruction retirements for TCP receive in QEMU host with VirtIO virtual NIC, system-wide profiling with CPU switch.

% INSTR	Function	Module
10.5178	strcat	vmlinux
3.8835	ipt_do_table	vmlinux
2.7508	olpc_ec_cmd	vmlinux
2.4272	__schedule	vmlinux
2.4272	__slab_alloc	vmlinux
2.2654	ip_route_input	vmlinux
2.2654	skb_gro_receive	vmlinux
1.9417	vring_add_buf	virtio_ring.ko

Table 15. Instruction retirements for TCP receive in QEMU guest with VirtIO virtual NIC, system-wide profiling with CPU switch.

5.6 Profiling Overhead

Profiling based on CPU performance counters inevitably slows down the profiled program, even in a native environment, because of the overhead of handling the counter overflow interrupts. In a virtualized environment, these interrupts need to be forwarded to the guest, adding more context switches between the host and the guest and therefore more overhead. In addition, the VMM needs to save and restore the performance counters on a VM switch.

We evaluate the overhead of our profiling extensions by comparing the execution time, with and without profiling, of the program in Figure 2, which is modified to execute a fixed number of iterations. The program runs in the guest, and we take a sample every 5 million CPU cycles (or about 400 times per second).

Table 16 presents the results for profiling overhead. In the native environment, the overhead of profiling is about 0.048%. For KVM guest-wide profiling, the overhead is about 0.386%. We further breakdown the overhead into two parts: additional context switches due to interrupt injection account for about 80% of overall overhead and interrupt handling in the guest takes the remaining 20%. For KVM system-wide profiling, the overhead is 0.441%. This is roughly the sum of the overhead of native and KVM guest-wide profiling, because KVM system-wide profiling also runs a profiler in the host Linux. System-wide profiling for QEMU incurs more overhead, around 0.942%. The overhead comes from multiple sources. First, QEMU runs in user space, and forwarding an interrupt to the guest requires a change in CPU privilege level and a signal to the user space process. Second, QEMU needs to query the address mapping cache and rewrite the sampled address. Third, frequent context switches also hurt the performance of QEMU's binary translation engine.

Profiling environment	Execution time overhead
Native	0.048% ± 0.0042%
KVM guest-wide	0.386% ± 0.0450%
KVM system-wide	0.441% ± 0.0435%
QEMU system-wide	0.942% ± 0.0441%

Table 16. Profiling overhead. The sample rate of the profiler is about 400 times per second.

6. Discussion

Although both guest-wide and system-wide profiling are feasible and useful for diagnosing performance problems in a virtualized computing environment, there are still a number of issues that need to be considered before these techniques can be deployed in production use, as discussed next.

Virtual PMU interface Since the PMU is not a standardized hardware component of the x86 architecture, the programming interfaces for PMUs differ between hardware vendors and even between different models from the same hardware vendor. In addition, different processors may also support different profiling events. Therefore, for guest-wide profiling to be portable to different processors, a proper interface between the guest profiler and the virtualized PMU must be defined. There are two ways to expose PMU interfaces to the guests.

The first method is to rely on the `CPUID` instruction to return the physical CPU family and model identifier to the guest. This information tells the guest profiler how to program the PMU hardware directly. The burden on the VMM is minimal, but the solution breaks one fundamental principle of virtualization: decoupling software from hardware.

The second approach is to expose to the guest a standardized virtual PMU with a limited number of broadly used profiling events, such as CPU cycles, TLB and cache misses, etc. The guest profiler is extended to support this virtual PMU, and the VMM provides the mapping of operations between the virtual PMU and the underlying physical PMU. This approach decouples software from hardware, but imposes additional work on the VMM.

Profiling accuracy In addition to the statistical nature of sampling-based profiling, there are other factors that potentially affect profiling accuracy in virtualized environments.

The first problem is that the multiplexing of some hardware resources inevitably introduces noise into profiling results. For instance, TLBs are flushed when switching between the VMM and the guests. If TLB misses are being monitored, the profiling results in a guest are perturbed by the execution of the VMM and/or other guests. This problem also exists in native profiling. Although it can be mitigated by cache/TLB entry tagging, profil-

ing results for these events are still not guaranteed to be entirely accurate.

The second problem is specific to profilers based on domain switch. If the VMM is interrupted to perform some action on behalf of another guest, the handling of this interrupt is incorrectly charged to the currently executing guest. The issue is similar to the resource accounting problem in a conventional operating system [5], and can possibly be solved by techniques such as early demultiplexing [10].

PMU emulation In addition to virtualizing the PMU, for VMMs based on binary translation and pure emulators, it is also possible to emulate the PMU hardware in software. In this case, the PMU of the physical CPU is not involved during the profiling process, and the entire functionality of the PMU is emulated in software. By emulating the PMU the VMM can support some events that are not implemented by the physical PMU. For instance, if the energy consumption of each CPU instruction is known, one could build an energy profiler in this way.

We use PMU emulation to count instruction retirements. When a basic block is translated, we count the number of guest instructions in the block and insert a few instructions at the beginning of the translated basic block. When the translated block is executed, these instructions increase the emulated performance counter. If the emulated counter reaches the predefined threshold, an NMI is injected into the virtual CPU. The difficulties of PMU emulation lie in supporting a large number of hardware events. Emulating these events may incur high overhead and emulating some of them may not even be possible for a binary translator or an instruction-level CPU emulator.

7. Related Work

The XenOprof profiler [18] is the first profiler for virtual machines. According to our definition, it does system-wide profiling. It is specifically designed for Xen, a VMM based on paravirtualization. A newer version of Xen, Xen HVM, supports hardware-assisted full virtualization. Xen HVM saves and restores MSRs when it performs domain switches, but it does not attribute samples from domain 0, in which all I/O device emulation is done, to any guest. VM exits in a domain that do not require the intervention of domain 0 are handled by Xen HVM under the context of that domain. As a result, guest-wide profiling in Xen HVM reflects neither the characteristic of the physical CPU nor that of the CPU plus the VMM.

Linux perf [2] is a new implementation of performance counter support for Linux. It runs in the Linux host and can profile a Linux guest running in KVM [1]. It obtains samples of the guest by observing its virtual CPU state. Because this is done outside the guest, only PC and CPU privilege mode can be recorded. The address of the descriptor of the current process is not known. As a result, Linux perf can only interpret samples that belong to the kernel of the Linux guest, and cannot handle samples contributed by user space applications. The binary image and the virtual memory layout of the guest kernel, necessary for sample interpretation, are obtained through an explicit communication channel.

VMware vmkperf [14] is a performance monitoring utility for VMware ESX. It runs in the VMM and only records how many hardware events happen in a given time interval. It does not handle counter overflow interrupts, and it does not attribute them to functions. It does not support the profiling mechanisms presented in this paper.

VTSS++ [9] demonstrates a profiling technique similar to guest-wide profiling. It requires the cooperation of a profiler running in the guest and a PMU sampling tool running in the VMM. It relies on sampling timestamps to attribute hardware events sampled in the host to the corresponding threads in the guest. Although it does not require modifications to the VMM, VTSS++ requires access to the VMM to run a sampling tool, and the accuracy of the profiling results is affected by the estimation algorithm it uses.

The work in this paper builds on our earlier work [11], which proposes some basic ideas of virtual machine profiling and only concentrates on guest-wide profiling for VMMs based on hardware-assisted virtualization. This paper extends the earlier work in several ways. We implement system-wide profiling for a VMM based on binary translation. We also evaluate our implementations through extensive experiments to demonstrate the feasibility and usefulness of virtual machine profiling.

8. Conclusions

Profilers based on CPU performance counters help developers debug performance problems in complex software systems, but they are not well supported in virtual machine environments, making performance debugging in such environments hard.

We define guest-wide profiling, which allows profiling of a guest without VMM access, and system-wide profiling, which allows profiling of the VMM and any number of guests. We study the requirements for each type of profiling. Guest-wide profiling requires synchronous interrupt delivery to the guest. System-wide profiling requires cooperation between the VMM and the guest to interpret samples belonging to the guest. We describe two approaches to implement this cooperation, full-delegation and interpretation-delegation.

We develop a guest-wide and a system-wide profiler for a VMM based on hardware-assisted virtualization (KVM), and a system-wide profiler for a VMM based on binary translation (QEMU). We demonstrate the accuracy and the power of these profilers, and show that their performance overhead is very small.

As more and more computing is migrated to virtualization-based cloud infrastructures, better profiling tools for virtual machines will facilitate performance debugging and improve resource utilization in the cloud.

Acknowledgements

We would like to thank Mihai Dobrescu, Simon Schubert and the anonymous reviewers for their valuable comments and help in improving this paper.

References

[1] Enhance perf to collect KVM guest os statistics from host side. 2010. http://lwn.net/Articles/378778.

[2] Performance Counters for Linux. 2010. http://lwn.net/Articles/-310176.

[3] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: where have all the cycles gone? *Operating Systems Review*, 1997.

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

[6] 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 9th ACM Symposium on Operating Systems Principles*, 2003.

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

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[9] Stanislav Bratanov, Roman Belenov, and Nikita Manovich. Virtual machines: a whole new world for performance analysis. *Operating Systems Review*, 2009.

[10] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. *Operating Systems Review*, 1996.

[11] 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*, 2010.

[12] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Notices*, 1982.

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

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

[15] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*.

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

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

[18] 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*, 2005.

[19] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 2005.

[20] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review*, 2008.

[21] B. Sprunt. The basics of performance-monitoring hardware. *IEEE MICRO*, 2002.

[22] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

[23] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*,1996.