©ACM, 2011. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '11). ACM, New York, NY, USA, 15-26 http://doi.acm.org/10.1145/1952682.1952687.

Perfctr-Xen: A Framework for Performance Counter Virtualization

Ruslan Nikolaev Godmar Back
Virginia Polytechnic Institute
Blacksburg
{rnikola,gback}@cs.vt.edu

Abstract

Virtualization is a powerful technique used for variety of application domains, including emerging cloud environments that provide access to virtual machines as a service. Because of the interaction of virtual machines with multiple underlying software and hardware layers, the analysis of the performance of applications running in virtualized environments has been difficult. Moreover, performance analysis tools commonly used in native environments were not available in virtualized environments, a gap which our work closes.

This paper discusses the challenges of performance monitoring inherent to virtualized environments and introduces a technique to virtualize access to low-level performance counters on a per-thread basis. The technique was implemented in perfctr-xen, a framework for the Xen hypervisor that provides an infrastructure for higher-level profilers. This framework supports both accumulative event counts and interrupt-driven event sampling. It is light-weight, providing direct user mode access to logical counter values. perfctr-xen supports multiple modes of virtualization, including paravirtualization and hardware-assisted virtualization. perfctr-xen applies guest kernel-hypervisor coordination techniques to reduce virtualization overhead. We present experimental results based on microbenchmarks and SPEC CPU2006 macrobenchmarks that show the accuracy and usability of the obtained measurements when compared to native execution.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Performance measures; D.4.8 [*Operating Systems*]: Performance—Measurements

General Terms Performance, Measurement

Keywords Profilers, virtual machine monitors, perfctr, PAPI, HPCToolkit, Xen

1. Introduction

Virtualization allows multiple instances of an operating system to run on a single computer. The idea was first introduced for VM/370 [13] and has later been reincarnated for modern platforms as described in [11] and [24]. A *Hypervisor* or *VMM (virtual machine monitor)* is a software layer that separates the virtual hardware an OS sees from the actual hardware and arbitrates access to physical resources such as CPU or memory. Among widely known VMMs are Xen [8] and KVM [18], [20]. Virtualization improves *isolation* and *reliability* because each OS runs independently from the others on its own virtual processor, increases *resource utilization* as the same hardware can be used for multiple purposes, and leads to better *productivity* as large pieces of software can be preconfigured and installed very easily.

Virtualization is widely used in several application domains. It allows the creation of "virtual appliances" [25], which are software bundles that contain their own specialized OS and run along with other virtual appliances and general purpose OS on a single machine. Commercial providers of infrastructure as a service (IaaS) solutions rely on virtualization to provide business solutions for server consolidation. Virtualization has also been proposed as a means of utilizing manycore platforms efficiently [7].

However, running performance-critical applications in virtualized systems is challenging because of virtualization overhead and the difficulty of making appropriate resource allocation and scheduling decisions. Commonly used performance evaluation frameworks extensively exploit profiling to allow systems and application developers to understand the performance of their applications. Such profiling frameworks rely heavily on hardware performance counters provided by modern CPUs. These counters provide information about hardware-related events such as cache misses, branch mispredictions, and many others.

Like any other resource, access to these hardware performance counters must be managed by both the hypervisor and the guest operating system kernel. However, current hypervisors are unable to provide efficient and virtualized access to performance counters. Our work closes this gap.

This paper presents perfctr-xen, an infrastructure to provide direct access to hardware performance counters in virtualized environments using the Xen hypervisor. Perfctr-xen relies on the cooperation of guest kernel and underlying hypervisor to provide profiling tools running in the guest with access to performance counters that is compatible with the APIs used in native, unvirtualized environments, notably PAPI [10]. Consequently, frameworks and libraries that rely on PAPI can now be used inside Xen, such as HPCToolkit [6] or TAU [26]. To accomplish this compatibility, we modified both the Xen hypervisor as well as the guest kernel running inside each virtual machine. Perfctr-xen supports both paravirtualized mode as well as hardware virtualization mode and exploits optimizations that avoid trap-and-emulate overhead. Although our implementation focuses on Xen, the techniques we use are applicable to other hypervisors.

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, to 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

Library/Framework	Type	Monitoring	Direct access	Interfaces Used
perf_events	low level	Per thread	Yes	ioctl, mmap, sysctl, prctl
perfctr	low level	Per thread	Yes	ioctl, mmap, dev
perfmon	high and low level	Per thread	No	syscalls, mmap, signals
PAPI	high level	Per thread	Yes (w/ perfctr)	perfctr, perf_events, perfmon
OProfile	profiler	System wide	N/A	oprofilefs
XenoProf	profiler	System wide	N/A	oprofilefs
TAU PerfExplorer	profiler	Per thread	N/A	PAPI
HPCToolkit	profiler	Per thread	N/A	PAPI

Table 1. Characteristics of Performance Monitoring Libraries and Frameworks

2. Background

This section introduces hardware event counters and existing libraries and frameworks that provide access to them. We further discuss the state of support for such counters in existing virtual machines.

2.1 Hardware Event Counters

Modern CPUs provide access to hardware event counters through programmable performance monitoring registers. Such registers can be programmed to count events of interest, such as cache accesses or misses or branch mispredictions. The registers may be read-only or read-write. System software controls whether registers are directly accessible to non-privileged user applications or whether accesses must be done in privileged mode from system code. The number of registers is typically smaller than the set of event types that can be counted, requiring that the user select a subset of events of interest. The set of event types is specific to a given microarchitecture and frequently changes as the microarchitecture evolves. Performance monitoring registers can be set up to trigger interrupts when they overflow. This mechanism is useful to perform statistical profiling using sampling intervals that contain a constant number of the events of interest in each interval.

2.2 Performance Monitoring Frameworks and Libraries

A cornucopia of performance monitoring frameworks and libraries exist. A representative set of examples is shown in Table 1. These frameworks and libraries differ in their functionality, level of abstraction, granularity of monitoring, and the interfaces upon which they rely.

At the bottom level, *low-level performance counter libraries* provide a thin layer over the facilities provided by the hardware, which does not hide architecture-specific event types from the user. These systems consist of kernel extensions and a corresponding user library. The kernel extensions implement operations that require privileged access, such as reprogramming counters or setting up interrupt handling and forwarding interrupt notifications to processes. The user library provides an API for accessing event counters.

Events may be counted globally (system-wide), or per thread. Most libraries support both modes, although some (e.g., OProfile [12]) provide only global counting. Global recording of events has the advantage that it can account for vertical interactions at all levels of the software stack as well as during horizontal interactions with other programs, such as local servers. On the other hand, global profiling makes it difficult to separate events of interest from unrelated system activities or noise.

Per-thread counting provides each thread with its own logical set of performance counters, just like each thread has its own logical set of machine registers. To implement per-thread accounting, the performance counter framework needs to maintain per-thread state which is updated on each context switch.

Some libraries (e.g., perfctr [22] and perf_events [2] [1]), allow a thread to directly read the physical performance monitoring register in user mode in order to obtain fine-grained and precise information about events during its execution, while other libraries (e.g., perfmon [17]) require a system call to obtain access to this information. Those systems that provide direct access must either maintain the thread's logical value in the physical register while the thread is scheduled, or they must place a correction (offset) value in an agreed-upon location (such as a memory-mapped area in the thread's address space) that allows a thread to compute its logical value based on the value read from the physical register.

High-level performance counter libraries such as PAPI [10] provide a layer that hides microarchitecture-specific event types behind a uniform, higher-level API. Performance profilers such as TAU [26] and HPCToolkit [6] in turn are built on top of higher-level performance counter APIs. These profilers statistically sample events and present cumulative statistics to the user that relates these events to instructions and functions in the user codes, with appropriate references to the source code if available.

The choice of framework influences the accuracy of measurements [27]. Bypassing high-level APIs in favor of low-level APIs typically reduces the measurement error, but requires architecture-specific code. The accuracy depends also on which events should be counted (user mode only vs. user and kernel mode events [27]).

To compensate for the limited number of performance counter registers, some frameworks (perfmon, PAPI) support event multiplexing. This technique applies only a subset of the desired event sets during subsections of a program's execution, then scales the results to extrapolate their values for the entire program.

2.3 Virtualization Approaches

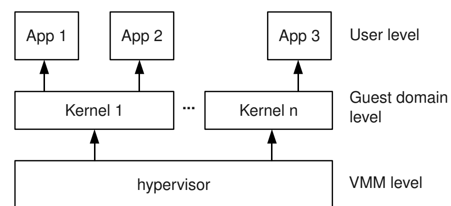

Figure 1. Architecture of a Type-I Virtual Machine Monitor

In this work, we consider Type I virtual machines [19] (see Figure 1) in which the hypervisor forms the lowest layer with direct access to the hardware and guest operating systems run on top of the hypervisor in separated domains.

Fully virtualized systems leave the guest OS entirely unaware that it is not running on physical hardware. As a result, an unchanged guest kernel (such as an off-the-shelf image of a commercial OS) can be executed. Such full virtualization can be accom-

[1] perf_events was previously known as perf_counter

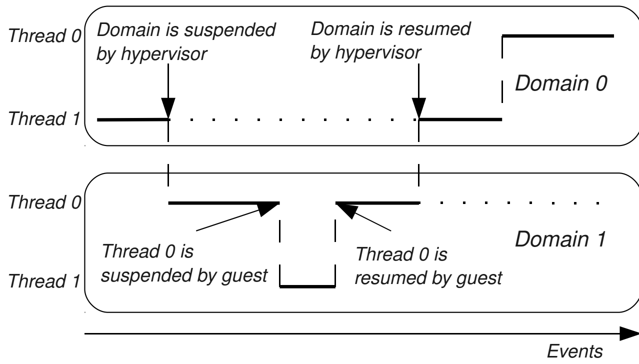

Figure 2. Context switching in a virtualized environment. The guest domains and the hypervisor are both unaware of when a domain or thread switch takes place.

plished either using hardware assist, or using binary translation. Hardware-assisted systems such as Intel VT or AMD-V run the kernel in a deprivileged mode, allowing the hypervisor to trap and emulate any instructions whose effect must be local to a given domain. Prior to the introduction of hardware assist, full virtualization of IA32-based systems required binary translation of the guest kernel code because the architecture lacked the ability to intercept all necessary instructions when executed in deprivileged mode [23].

For this reason, the Xen virtual machine monitor [8] introduced adaptations into the guest OS kernel code (a process known as paravirtualization), to reduce emulation and management costs and yield better performance. Beyond avoiding binary translation, the adaption of guest kernel code either in their core or through the use of special drivers has become common today for any virtual machine monitor.

2.4 Performance Counters in Virtual Machines

Support for performance event monitoring depends on the type of virtualization being used. There is limited support in Xen for selected microarchitectures when hardware-assisted virtualization (e.g., using Intel VT or AMD-V) is used. In this approach, accesses to performance monitoring registers are intercepted via traps. However, the set of architectures supported is far smaller than that supported by PAPI, and hardware-assisted virtualization is not always used.

The XenoProf [21] framework extends the OProfile [12] system-wide profiler to allow per-domain (e.g., per guest) profiling in Xen, even when hardware-assisted virtualization is not used. However, XenoProf does not allow independent and simultaneous profiling of different domains.

Aside from these approaches, most current virtual machines disallow access to the performance monitor registers by the guest operating systems, thus preventing widely used low-level libraries such as perfctr and perf_events from being used. Consequently, users cannot benefit from high-level performance profilers such as TAU PerfExplorer and HPCToolkit to diagnose the performance of their applications when executing on top of virtual machines.

3. Virtualizing Hardware Event Counters

The encapsulation of guest domains from the underlying hypervisor poses a difficulty for virtualizing performance counters, because these two components are mutually unaware of their scheduling policies. As shown in Figure 2, the guest kernel remains unaware if the hypervisor suspends its domain on the physical CPU on which it runs. Likewise, the hypervisor is unaware of when a guest kernel switches to a different user task on its domain's virtual CPUs.

Both inter- and intra-domain context switches involve the performance monitoring framework and may require updating machine-specific (MSR) registers. First, PMU (Performance Monitoring Unit) configuration registers (e.g., event selectors) need to be re-programmed to reflect the desired event configuration of the thread to be resumed. Second, if the performance counter register contains the logical value of the thread to be resumed, it must be restored (and the value of the outgoing thread must be saved). Otherwise, its value must be sampled and recorded in the corresponding data structure for the thread or domain.

Since the guest domain kernel runs in a deprivileged environment, its access to the registers during intra-domain context switches must be managed by the hypervisor. For fully-virtualized domains, a trap-and-emulate approach can be used to intercept and emulate the corresponding privileged instructions that write to these registers. Although read accesses could be trapped as well, current architectures allow the hypervisor to grant direct read access to the MSR registers to guest domains. If the guest is adapted to use paravirtualization, the cost of trapping and emulating individual privileged instruction can be reduced by using hypercalls instead, which allows the batching of multiple updates. The guest can also cache previously activated configurations to avoid these hypercalls if possible, thus avoiding unnecessary writes to the MSR configuration registers.

Whereas the counter-related configuration information must be saved and restored on both inter- and intra-domain switches, the values of the registers containing the actual event counts require saving and restoring only if the register physically contains the thread's logical value during execution. Consequently, the cost of writing to these registers can be avoided when this is not required, which holds true in two cases. First, if a counter is used to obtain accumulated event counts (in 'a-mode'), a virtualization-aware guest domain can apply the necessary correction offsets to obtain the logical accumulated value from the physical value. Second, in the case of the timestamp counter (TSC) register, hardware-assisted virtualization via Intel VT or AMD-V allows for transparent, per-domain offsetting, which also avoids the cost of physically updating these registers on a context switch. In addition, on some architecture implementations, it is impossible to safely update the TSC register since it may be shared across cores. Avoiding the save-and-restore cost is beneficial because it can be expensive (66-93 CPU cycles per register; for Pentium 4 as much as 18 registers must be restored).

On the other hand, if a counter is used in interrupt mode ('i-mode'), the physical register contains a small negative value that will overflow, thus triggering an interrupt, after a desired number of events occurs. In this case, saving and restoring cannot be avoided. The interrupt is handled by the hypervisor, who must forward this interrupt to the affected domain via a virtual interrupt mechanism. Upon receipt of the virtual interrupt, the guest kernel notifies the user-level profiling components via a Unix signal. Previous work [21] claimed that such delivery needs to be synchronous, an assertion repeated in [15]. In Xen, the delivery of virtual interrupts to guest domains is not synchronous, but uses a software interrupt mechanism, thus making it possible that their delivery could be delayed by higher-priority interrupts. In Section 4.2.2, we argue that support for synchronous interrupt delivery is not required if the performance counter registers are restored before the guest domain is resumed, and if the guest domain verifies that a register in fact overflowed before delivering the notification to the user level.

Perfctr-xen supports performance counter virtualization in Xen in three configurations, which required different and separate implementations: (1) for paravirtualized guest kernels, which use

hypercalls to communicate performance counter configuration changes from guest to hypervisor, and in which the guest and hypervisor cooperate to maintain information about the current thread context; (2) for fully-virtualized guest kernels, which use the save-and-restore approach for all registers; and (3) a hybrid approach in which a guest can run in a hardware-assisted, fully-virtualized domain but still enjoy the generality of and the optimizations developed for the paravirtualization case.

4. Implementation

Our implementation is based on, and compatible with, the existing *perfctr* [22] implementation. In this section, we describe *perfctr* in detail and outline how we adapted it to enable performance counter virtualization in Xen for paravirtualized and hybrid modes. Section 4.2.5 describes our virtualization strategy for fully virtualized domains.

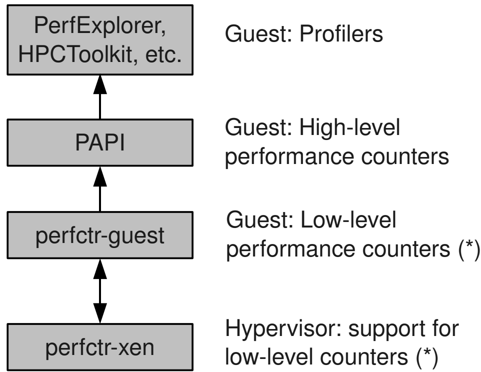

Figure 3. Software layers in perfctr-xen. Components marked with an asterisk (*) were adapted from *perfctr*.

4.1 The *perfctr* library

We chose *perfctr* because it is widely used and provides the foundation for higher-level libraries and frameworks such as PAPI, HPC-Toolkit, or PerfExplorer, as shown in Figure 3. It is efficient, lightweight and allows direct access to performance counters in user mode. *Perfctr* supports a wide range of x86 implementations spanning multiple generations and different vendors, whose hardware event counter implementation can differ significantly. In addition, *perfctr* works on non-x86 platforms such as PowerPC and ARM and can easily be integrated in any Linux distribution.

Perfctr consists of a kernel driver and a user-level library. The kernel driver maintains performance counter-related per-thread data structures, updates them on each context switch, and makes them available to the user-level library via a read-only mapping. Besides miscellaneous architecture-specific information, this per-thread data structure contains the following information:

- *Control State.* Information about which PMU data registers a thread is actively using, which events these registers count, and to which physical register address they are mapped. Similar information is kept with respect to the use of the time-stamp counter, which is also virtualized.
- *Counter State.* For each PMU data register, as well as the TSC register, two values are kept: Sum_{thread}, which reflects the thread's accumulated logical event count up to including the last suspension point; and $Start_{thread}$, which reflects the sampled value of the counter at the last resumption point.

Perfctr supports two types of counters: *a-mode* and *i-mode* counters. A-mode counters are used by threads to measure the number of events occurring in some region of a program. User code explicitly reads the counter's value when needed. When a thread wants to access the logical value of a counter at time t, a user library function issues a `RDTSC` or `RDPMC` instruction to obtain the register's physical value $Phys(t)$ and computes the logical value $Log_{thread}(t)$ as

$$Log_{thread}(t) = Sum_{thread} + (Phys(t) - Start_{thread}) \quad (1)$$

On each context switch, the *perfctr* kernel driver updates the accumulated value of the outgoing thread as $Sum_{thread} \leftarrow Sum_{thread} + (Phys - Start_{thread})$ to account for the events during the last scheduling period. In addition, the $Start_{thread}$ value of the thread to be resumed is reset as $Start_{thread} \leftarrow Phys$. Note that the actual physical register value is not changed on a context switch for a-mode counters.

I-mode counters, which are used for sampling, trigger interrupts after a certain number of events has occurred, which represents the sampling period. Since the value at which an overflow interrupt is triggered is fixed at 0 and cannot be programmed, the physical register must be set to a small negative value whose absolute value represents the desired length of the sampling period. *I-mode* counters are treated differently during a context switch: their physical value is saved on suspend and restored on resume. The Sum_{thread} field maintains the counter's accumulated logical value as for a-mode counters. The $Start_{thread}$ field is used to record the physical value when a thread is suspended. Consequently, the logical value of an i-mode counter can also be obtained using equation (1).

When an overflow occurs, *perfctr* handles this interrupt, identifies the register(s) that have overflowed and updates Sum_{thread}, then disables further event counting for these registers. Using the OS's signal delivery mechanism, a signal is sent to the user process. The signal handler is then responsible for recording the sample based on the provided user process's state and it must re-enable event counting. Once re-enabled, the physical value of the register is reset to the sampling period, which is also recorded in the data structure maintained by *perfctr*.

4.2 The perfctr-xen framework

Perfctr-xen includes a hypervisor driver, a guest kernel driver, and a modified user-level library, whose functionality we describe in this section.

4.2.1 A-mode counters

The virtualization technique described in Section 4.1 requires that the underlying system perform two actions during a context switch: (1) update the counter state of the threads being suspended and resumed, and (2) activate the resumed thread's control state. As discussed in Section 3, in a virtualized environment, both intra-domain context switches between threads in a domain and inter-domain context switches between domains can occur. During intra-domain switches, the guest kernel can perform the state updates similar to the native implementation. For inter-domain context switches, the hypervisor must perform these updates.

We first considered having the hypervisor update each thread's counter state directly on the guest kernel's behalf. This approach has the advantage that no changes to the *perfctr* user library are required. However, it would create undesirable coupling between the hypervisor and the guest kernel implementations, because the hypervisor would need to traverse guest kernel data structures. Instead, we decided to split the control and counter state in two parts. At the guest kernel level, a per-thread data structure is maintained. At the hypervisor level, a per-VCPU data structure is maintained for each virtual CPU that is assigned to a guest domain. The hypervisor provides read-only access to this data structure to the guest

kernel, who in turn maps it into the address space of each thread using performance counters.

The per-VCPU data structure is modeled after the per-thread data structure used in the native version of *perfctr* (in fact, our implementation uses the same data structure declarations, as discussed in Section 4.2.4). For each PMU data register, as well as for the TSC register, the hypervisor maintains two values per VCPU: $Start_{vcpu}$ and Sum_{vcpu}. $Start_{vcpu}$ represents the sampled value of the counter at the most recent resumption point of the domain or thread (whichever happened last). If the hypervisor resumes a domain, it directly updates $Start_{vcpu}$ after sampling the counter. If the guest kernel resumes a thread, it requests via a hypercall that the hypervisor record the sampled value in $Start_{vcpu}$. The same hypercall is also used to activate this thread's counter-related control state.

The field Sum_{vcpu} represents the cumulative number of events incurred by this domain since the last intra-domain thread resumption point until the most recent domain suspension point. It is set to zero on each intra-domain switch during the hypercall that notifies the hypervisor that the guest kernel resumed a thread. On each inter-domain context switch, the perfctr-xen hypervisor driver updates the accumulated value of the outgoing VCPU as $Sum_{vcpu} \leftarrow Sum_{vcpu} + (Phys - Start_{vcpu})$ to account for the events incurred since the last intra- or inter-domain resumption point.

The perfctr-xen guest kernel driver maintains the value Sum_{thread} for each thread as in the native case, which represents the cumulative number of events up to the last thread suspension point. A counter's logical value at time t is computed as

$$Log_{thread}(t) = Sum_{thread} + (Phys^*(t) - Start^*_{thread}) \quad (2)$$

$Phys^*(t)$ represents the adjusted physical value that accounts for possible VCPU preemption, which is computed as

$$Phys^*(t) = Sum_{vcpu} + (Phys(t) - Start_{vcpu}) \quad (3)$$

Thus, the logical value represents the sum of the cumulative number of events until the last thread suspension point, plus the number of events encountered from there until the last domain resumption point while the domain was active, plus the events encountered since then until t, reduced by an adjusted start value $Start^*_{thread}$.

The adjusted thread start value $Start^*_{thread}$ compensates for the requirement that each intra-domain context switch includes a hypercall. Since this hypercall is introduced by our framework, we wish to exclude any events occurring during its execution. Right before resuming a guest thread, the guest kernel driver computes $Start^*_{thread} = Phys^*(t_r)$ after returning from the hypercall at time t_r

$$Start^*_{thread} = Sum_{vcpu} + (Phys(t_r) - Start_{vcpu}) \quad (4)$$

This adjustment excludes any events incurred between when the hypervisor sampled the counter during the hypercall and t_r. The inclusion of the term Sum_{vcpu} ensures that all such events are excluded, even if the domain was suspended and resumed during the hypercall by the preemptive scheduler. Since $Start^*_{thread}$ takes the place of $Start_{thread}$ in equation (1), we store its value in the $Start_{thread}$ field of the per-thread structure.

The use of $Start^*_{thread}$ enables an additional optimization. In applications in which multiple threads count the same types of events, the hypercall accompanying the intra-domain context switch does not change any counter's control state. In this case, we skip this hypercall. Consequently, $Start_{vcpu}$ is not reset to the counter's current physical value and Sum_{vcpu} is not reset to 0. Since we still initialize $Start^*_{thread}$ using equation (4), we allow the thread being resumed to subtract events incurred by other threads within the same domain since $Start_{vcpu}$ was last initialized. This optimization reduces the frequency with which $Start_{vcpu}$ is updated, which in turn increases the risk that an integer wrap-around leads to incorrect results when computing a thread's logical value. To reduce this risk, we expanded the width of the counters. Whereas *perfctr* uses 32 bits to represent only the lower 32 bits of all counters, our implementation uses their actual width (64 bits for the TSC register, and 48 bits for PMU data registers; 40 bits on older CPUs). We sign-extend based on the physical register width and store the extended values in 64-bit variables.

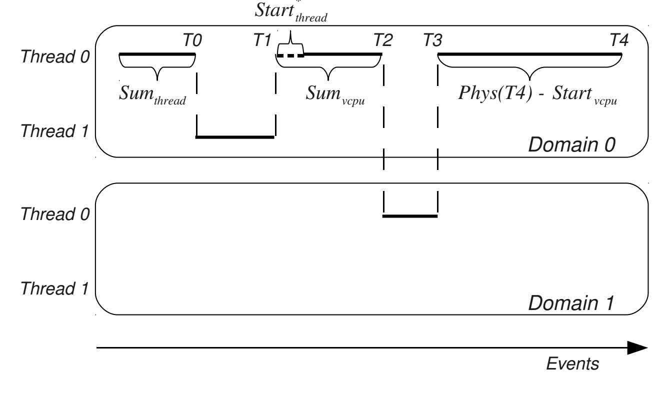

Figure 4. Example scenario for virtualized counters

Example Scenario. Figure 4 shows an example scenario to illustrate equations (2) and (3). Initially, thread 0 in domain 0 is running. At point T0, thread 0 is suspended by the guest kernel and its accumulated event count is recorded in Sum_{thread}. At T1, thread 0 is resumed. The hypervisor sets $Sum_{vcpu} \leftarrow 0$; upon return from the hypercall, the guest records $Start^*_{thread}$. At point T2, the domain is suspended; the hypervisor records the number of events elapsed in Sum_{vcpu} and later resumes the domain at point T3. At this point, the hypervisor samples $Start_{vcpu}$ as $Phys(T3)$. Finally, the logical value computed at time T4 reflects the sum of the three segments during which the thread was active, while excluding those time periods during which the thread or domain was suspended.

4.2.2 I-mode counters

As in the native *perfctr* implementation, i-mode counters require saving and restoring the physical register value on both intra- and inter-domain context switches. In addition, when a thread using i-mode counters is suspended by the guest, the PMU must be reprogrammed to stop triggering interrupts for this counter. Since writes to PMU registers can be performed only by the hypervisor, an additional hypercall is necessary when a guest thread is suspended. Our implementation uses the $Start_{vcpu}$ and Sum_{vcpu} fields in the VCPU structure to maintain the currently active thread's $Start_{thread}$ and Sum_{thread} values at the time the thread is resumed. While a thread is active, we set its per-thread $Start_{thread} \leftarrow 0$ and $Sum_{thread} \leftarrow 0$ in order to be able to use equation (2) to compute the logical value (if desired).

When an intra-domain context switch occurs, a guest invokes a suspension hypercall which will update Sum_{vcpu} and store the current physical value in $Start_{vcpu}$. These values are then preserved in the Sum_{thread} and $Start_{thread}$ fields of the outgoing thread. The guest then invokes a resumption hypercall which will restore Sum_{vcpu} and $Start_{vcpu}$ based on the previously saved Sum_{thread} and $Start_{thread}$ fields of the thread to be resumed. The $Start_{vcpu}$ value will be written to the corresponding physical register. When an inter-domain context switch occurs, the hypervisor updates Sum_{vcpu} to account for the events incurred by the

domain and preserves the outgoing VCPU's value in its $Start_{vcpu}$ field before restoring the physical register value from the saved $Start_{vcpu}$ field of the VCPU to be scheduled.

When an overflow interrupt occurs, the hypervisor forwards this interrupt to the guest domain using a virtual interrupt we added for this purpose (`VIRQ_PERFCTR`). When the guest receives the virtual interrupt, it performs the same actions as in the native *perfctr* implementation, with three slight nuances: (1) When the guest receives the virtual interrupt, it suspends counting for the interrupted thread, and Sum_{vcpu} will be updated via the suspension hypercall. To prepare for the next sampling period, $Start_{thread}$ is reset with the negative sampling period. (2) When the user thread resumes counting, the resumption hypercall is executed, which restores $Start_{vcpu}$ from $Start_{thread}$ and sets the physical register value from $Start_{vcpu}$. (3) The guest does not need to re-program APIC controller, as it has been done already by the hypervisor.

The virtual interrupt is not delivered synchronously because a software interrupt mechanism is used. As such, it is possible that the delivery of the interrupt is delayed, for instance, because other, higher-priority interrupts are being handled first. This could have two consequences. First, it could affect the accuracy because the events incurred during those interrupts will be counted as being part of that thread's activity. However, it is already the case that interrupt-related guest kernel activities can perturb the currently running thread's event count. Second, it is possible that a higher-priority interrupt triggers a context switch in the guest. When the virtual interrupt is eventually handled, a different thread may be running on the VCPU. Since we save and restore each thread's counter state on intra-domain interrupts, we can check if the currently running thread indeed encountered an overflow (i.e., if any of its counters have a non-negative value), and prevent the delivery of the overflow notification if this is not the case. When a thread whose counters have overflowed is suspended before the interrupt has been delivered, we mark it by setting an 'interrupt pending' flag, which is checked when the thread is resumed so that the overflow notification is delivered in the correct context.

4.2.3 Memory Management

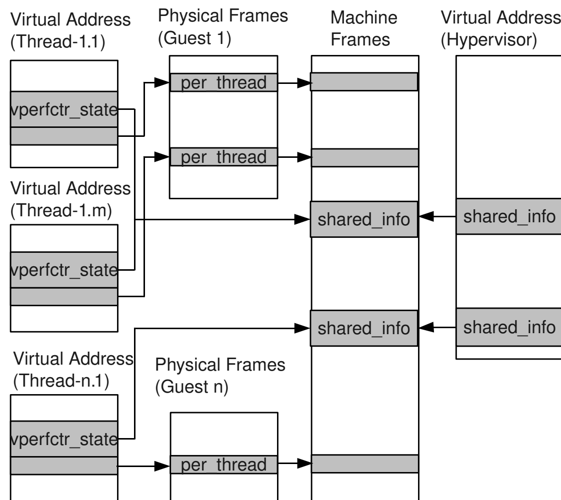

Figure 5. Page mappings in paravirtualized mode

The virtualization approach described in Sections 4.2.1 and 4.2.2 relies on sharing data structures between hypervisor and guest threads, as well as between guest kernel and guest threads. To expose the hypervisor's per-VCPU data structures, we extended the existing `shared_info` data structure in Xen. This addition increases the structure's size from 1 page to 8 pages; as a result, we needed to modify those places in the code where a single page size was assumed. The additional information, which is kept in 7 adjacent pages, is also made visible to user threads via a read-only mapping to facilitate the computation of the logical counter value. User threads also have read-only access to per-thread information which is mapped into their user space as in the original implementation.

The way in which the per-VCPU data mapping is established differs between paravirtualized and hardware-assisted mode. In paravirtualized mode, shown in Figure 5, the `shared_info` structure does not appear as physical memory to the guest kernel. Instead, it is allocated by the hypervisor in machine memory and appears at a fixed virtual address in the guest kernel's address space. The corresponding machine address is communicated to the guest kernel through the `xen_start_info` data structure. The guest kernel uses a Xen Guest API (`xen_remap_domain_mfn_range`) to create an additional mapping to these machine frames.

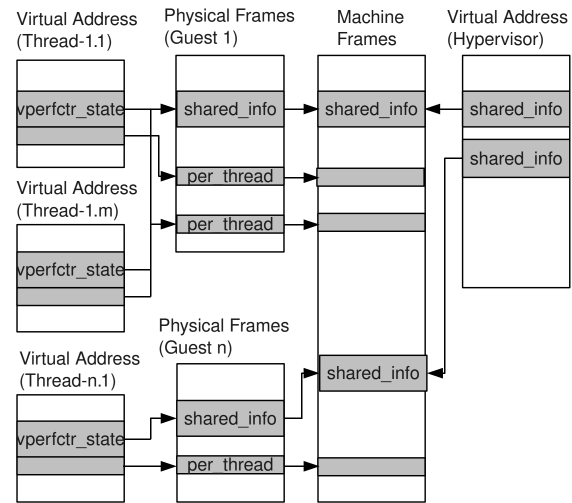

Figure 6. Page mappings in hybrid mode

In hardware-assisted mode, shown in Figure 6, the guest has full control over its physical address space. It can allocate the `shared_info` in any of its physical page frames. The chosen physical address is communicated from the guest kernel to the hypervisor. Since `shared_info` appears in the guest's physical memory map, Linux's standard mapping API (`vm_insert_page`) can be used to add read-only mappings into the user threads' address spaces.

A domain may use multiple VCPUs, and the guest kernel scheduler may migrate threads between VCPUs based on its scheduling policy. Consequently, we expose the per-VCPU data structures of *all* VCPUs to every user thread. We also added an additional field `smp_id` to the per-thread structure to record the VCPU on which the thread is resumed. The user-level library uses this field as an index to access the correct per-VCPU structure.

Care must be taken to handle thread or domain migrations that may occur while accessing those counters. We implemented an optimistic approach in which we check if the values of the $Start_{thread}$ and $Start_{vcpu}$ fields corresponding to the TSC counter changed between before and after the attempted access. Such a change indicates a domain and/or thread migration, in which case we retry the access until we succeed.

Component	Number of Lines	Details
perfctr	563	VCPU support, hypervisor communication, etc.
Linux	36	shared_info management, VIRQ_PERFCTR
Xen	3488	perfctr-xen, shared_info management, VIRQ_PERFCTR

Table 2. Added or modified code

4.2.4 Software Engineering Considerations

In addition to providing compatibility with *perfctr*, we aimed to reuse as much of its codebase as possible. We were able to reuse the architecture-dependent code portions almost entirely, which will allow us to add support for newer CPU families as soon as they are supported by *perfctr*. Both the guest kernel driver and the hypervisor driver are based on *perfctr*. For the guest kernel driver, we replaced the functions that assumed direct access to the hardware with the appropriate hypercalls. For the hypervisor driver, we needed to provide glue code so that it could function within the Xen hypervisor rather than the Linux kernel for which it was designed. This glue code was written in the form of preprocessor macros and inlined functions contained in a separate header file, allowing us to avoid changes to most of the *perfctr* code. Table 2 summarizes the total amount of added or modified lines of code in the Xen hypervisor, the Linux guest kernel and the imported *perfctr* code. We note that more than half of the number of lines of code added to Xen stems from the addition of the *perfctr* driver.

4.2.5 Counter Virtualization in Fully-Virtualized Domains

The implementation described in Sections 4.2.1 and 4.2.2 requires that the guest kernel includes our perfctr-xen implementation so that it can benefit from the optimizations we made to enable direct access to counter values, which avoids the cost associated with save-and-restore for a-mode counters. Fully-virtualized domains do not require any guest kernel changes.

For fully virtualized domains, Xen's VPMU driver already supports counter virtualization for PMU registers on some recent CPUs. This virtualization is achieved by using a save-and-restore mechanism for PMU registers on inter-domain context switches as well as a hardware-assisted trap-and-emulate mechanism for PMU configuration registers. (A similar approach was implemented for KVM in [15].) The hypervisor intercepts the privileged instructions a domain uses when writing to configuration registers. A hardware-supported access bitmap allows the hypervisor to provide exclusive access to dedicated PMU registers for a domain. These mechanisms allow the use of the native *perfctr* implementation in fully-virtualized environments for the PMU registers, but they fail to provide per-VCPU virtualization for the TSC register, which cannot be reliably written to.

Xen exploits the TSC offsetting feature provided in hardware-assisted virtualization, so that each domain can set its own virtual initial value. However, this per-VCPU offset δ is not adjusted during inter-domain switches, hence does not reflect just the cycles during which a particular VCPU was active. To address this problem, we modified the implementation in the following way. When a domain is suspended, we take a sample of the TSC value ($TSC_{last} \leftarrow Phys(t)$). When the domain resumes, we obtain the current value $TSC_{cur} \leftarrow Phys(t)$, and re-calculate the TSC offset as $\delta \leftarrow \delta - (TSC_{cur} - TSC_{last})$. The updated offset is recorded by the CPU and will be reflected when a guest executes the `RDTSC` instruction.

5. Experimental Results

Our perfctr-xen implementation was able to pass all *perfctr* and PAPI built-in tests. We verified that it functioned correctly with higher-level tools such as the HPCToolkit profiler. In this section, we discuss our experiments to validate the correctness of our implementation using microbenchmarks, we discuss test results obtained for the SPEC CPU2006 benchmarks, and show profiling results obtained using the HPCToolkit running on top of perfctr-xen. All experimental results were obtained for Xen 4.0.1, Linux 2.6.32.21-PvOps, perfctr 2.6.41 run on a Intel Xeon E5520 with 2x4 cores and 12 GB of RAM.

5.1 Validation of Correctness

To validate the correctness of our implementation, we compared the perfctr-xen implementation running in Xen with the original *perfctr* implementation running in native mode on the same hardware. For most counters, we expect to obtain the same value. For some counters (e.g., cache misses) we expect to see slight deviations because different domains running in parallel may compete for the same resource.

5.1.1 A-mode Counters

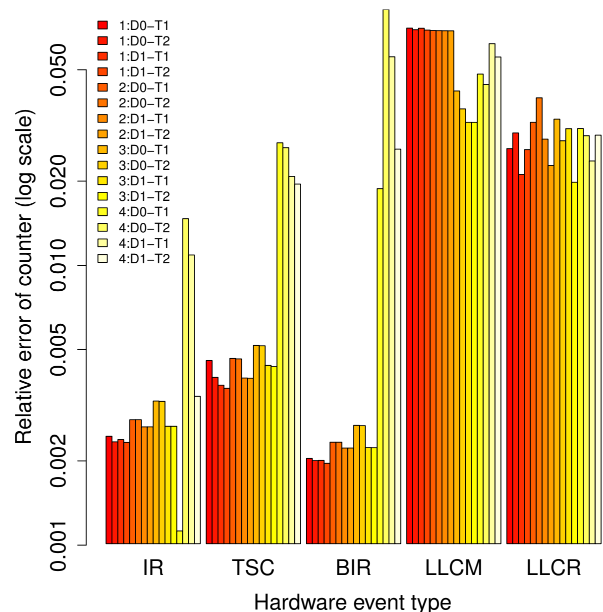

Figure 7. Microbenchmark result for a-mode counters

We first ran a specially developed 1-minute long microbenchmark. This synthetic microbenchmark heavily exercises branch instructions, memory accesses, and floating-point operations without performing a useful task. In Figure 7, the error relative to the native execution environment is shown for several test scenarios and event types. We considered two domains (Dom0 and Dom1) and two threads (Thread 1 and Thread 2) in each domain, running in parallel. Each test result is denoted as $N : D_x - T_y$, where N is the test case scenario, x is a guest domain and y is a thread. We considered the following test scenarios N, which represent different arrangements of CPU multiplexing: (1) Each domain runs on two dedicated physical cores (PCPUs), and each thread in every domain runs on a dedicated VCPU. (2) Each domain runs on a dedicated PCPU and all threads in every domain run on a shared VCPU. (3) Domains run on a shared PCPU and all threads in each domain run on a shared VCPU. (4) Like (1), except that threads

are randomly migrated across VCPUs, and VCPUs are randomly migrated across different PCPUs. We used the Xen `xm` command to pin VCPUs to PCPUs, and we used the Linux `taskset` command to pin threads to VCPUs.

We considered the following counters: TSC (Time Stamp Counter), IR (Instructions Retired), BIR (Branch Instructions Retired), LLCM (L2 cache misses), and LLCR (L2 cache references). The results show some deviations, but the overall relative error (compared to native) remains very small. As expected, the per-thread values for the number of cycles spent, instructions and branch instructions retired match more closely the values obtained in native execution than the values corresponding to cache misses because those values are less affected by resource sharing. The results shown were obtained for paravirtualized domains; we obtained comparable results for hardware-assisted domains using our hybrid mode implementation.

5.1.2 I-mode Counters

To verify the functioning of our i-mode counter implementation, we used PAPI's included tests. We present the results for the *overflow_pthreads* test. The test is a synthetic benchmark that performs a set of floating point operations. We ran the test for 300 sampling periods, and recorded the logical counter values afterwards. The benchmark runs 4 threads for which it uses a random CPU (VCPU in our case) assignment. We present results for two scenarios: (1) Dom0 and Dom1 run on separate PCPUs. (2) Dom0 and Dom1 run on a shared PCPU. As Figure 8 shows, the error relative to the native mode for the number of retired floating-point instructions (PAPI_FP_INS) is negligible. Figure 9 shows the number of cycles as measured using the PAPI_TOT_CYC event type, which exhibit a larger relative error. This result is expected because we do not compensate for events occurring in the hypercall events at resumption points when using i-mode counters. The hypercalls consume cycles, but do not perform any floating point operations.

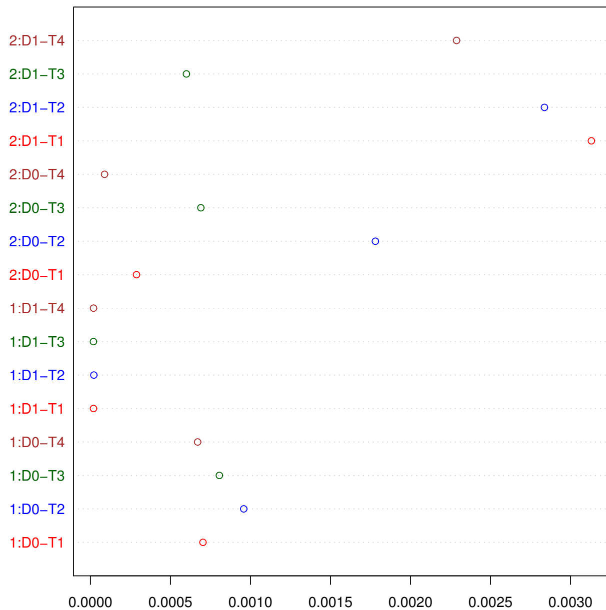

Figure 8. Relative error for i-mode counters (PAPI_FP_INS) for PAPI overflow test

5.2 SPEC CPU2006 benchmarks

We used the SPEC CPU2006 benchmarks as macrobenchmarks to show the correctness of our implementation and provide error estimates for CPU and memory bounded workloads. Native mode execution is again used as reference point. Since Dom0 is a paravirtualized domain in Xen, we used the Dom1 and Dom2 domains for tests that include fully-virtualized domains. (To exclude any possible effect of Dom0, we pinned it to a dedicated core.) We considered 5 scenarios: (1) Native mode execution. (2) Fully-virtualized domains Dom1 and Dom2, each running on a dedicated core (DC). (3) Fully-virtualized domains Dom1 and Dom2 running on the same core (SC). (4) Paravirtualized domains Dom0 and Dom1, each running on a dedicated core (DC). (5) Paravirtualized domains Dom0 and Dom1 running on the same core (SC).

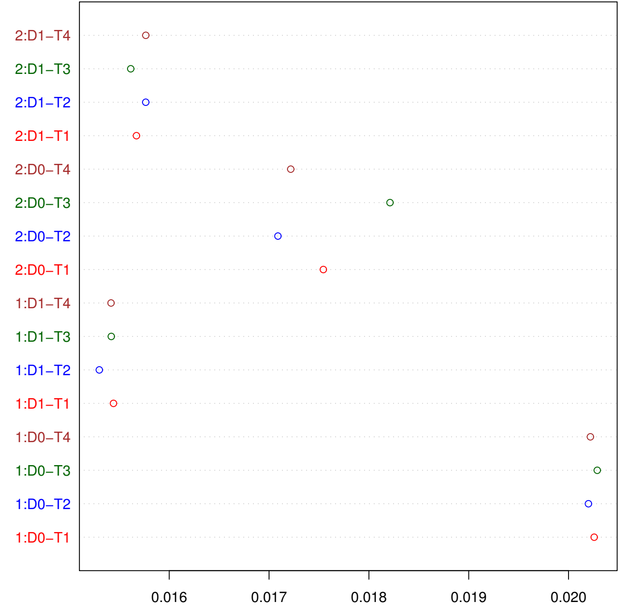

Figure 9. Relative error for i-mode counters (PAPI_TOT_CYC) for PAPI overflow test

The official SPEC distribution contains a large set of different benchmarks. We ran all of them using the 'train' problem size and recorded the total number of events counted during their execution. Since some benchmarks were executed under different data sets, we calculated the cumulative event counter values for all data sets. We present results for a subset of benchmarks only, choosing those for which both a non-negligible number of events was counted and for which the difference between the scenarios was largest; these represent the relative weakest performance of our framework.

In Figure 10, the results for the cycle counts reported by the virtualized TSC are shown. If the benchmarks execution were unaffected by virtualization, and if our framework achieved the same accuracy as *perfctr* running natively, we would expect to obtain the same results for all test scenarios for a given benchmark. This is true for most benchmarks, although 3 benchmarks (*mcf*, *astar*, and *lbm*) show significant deviations for the fully virtualized configuration. When counting the number of instructions retired (Figure 11), we did not observe any significant differences.

Figures 12 and 13 display the number of L2 cache references and misses, respectively. Since these events are more strongly influenced by environmental factors inherent to the virtualized environment, they show slightly larger deviations, particularly for the number of cache misses. For example, *libquantum* shows a significant drop in the number of cache misses observed, although the number of cache references is roughly the same. These effect warrant further investigation to ascertain if they indeed reflect environmen-

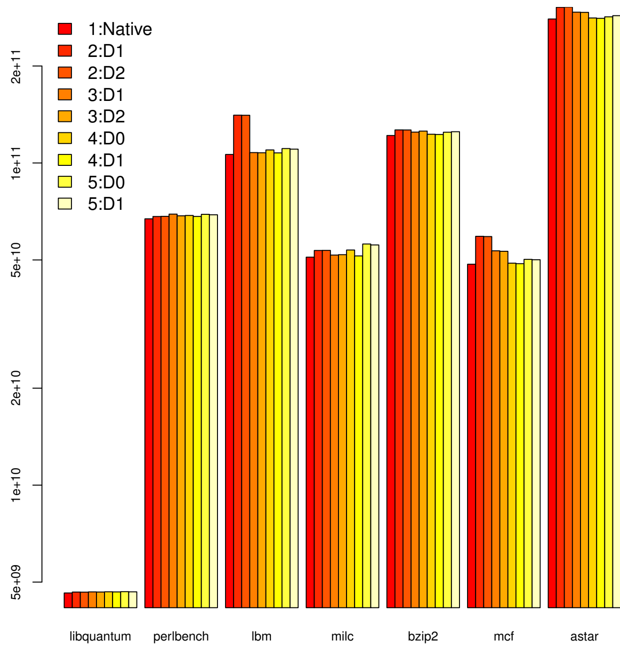

Figure 10. SPEC CPU2006, Time Stamp Counter (TSC)

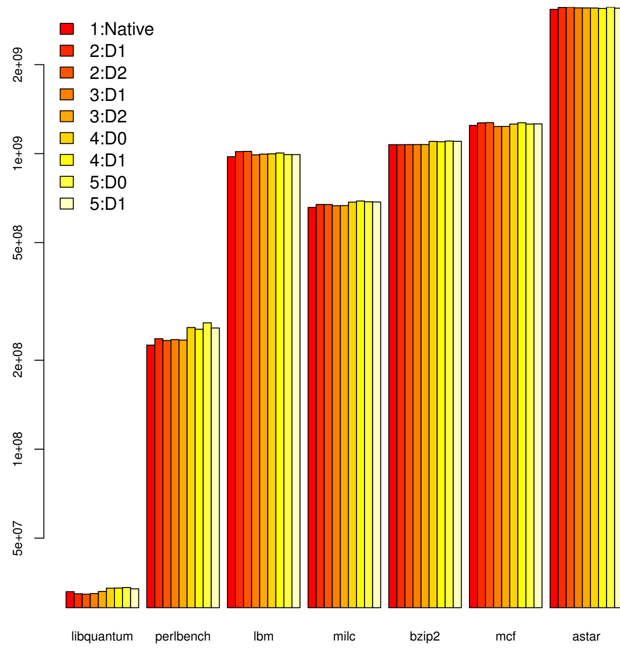

Figure 12. SPEC CPU2006, L2 Cache References

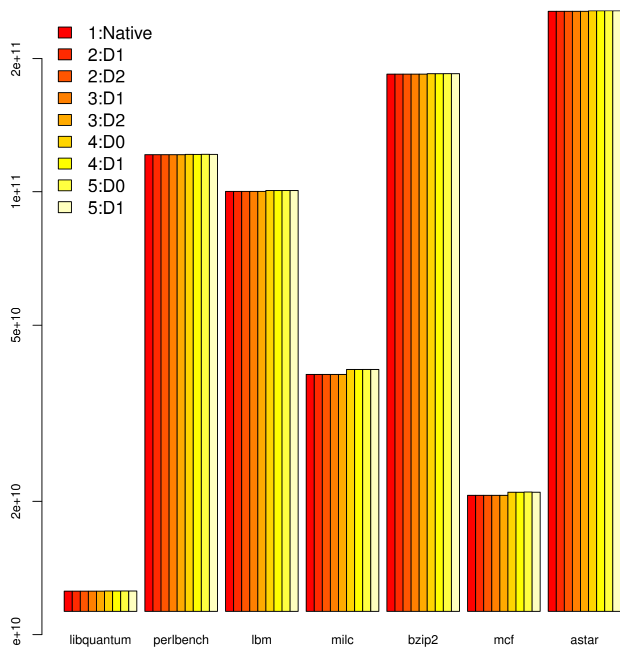

Figure 11. SPEC CPU2006, Instructions Retired

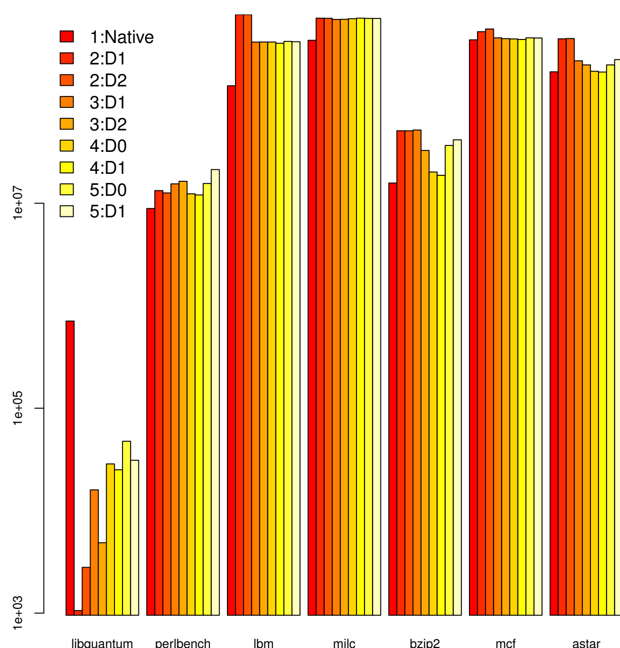

Figure 13. SPEC CPU2006, L2 Cache Misses

tal circumstances or are caused by inadvertent interactions with the measurement framework.

5.3 HPCToolkit Profiling

HPCToolkit's sampling mechanism is based on PAPI, which exploits the i-mode counter capabilities of our framework. As a profiler, HPCToolkit maps sample counts to individual functions. We tested HPCToolkit on the SPEC CPU2006 benchmarks. As an example, we selected the *429.mcf* benchmark, which performs combinatorial analysis. We considered PAPI_TOT_CYC (number of cycles), PAPI_L2_TCM (L2 cache misses), and PAPI_BR_INS (number of branch instructions) events. We considered sampling periods of 40000, 1000, and 500 events, the overall event counts ranged from 10s to 100s of millions of events. Similar to our previous setup, we run 2 concurrent instances of HPCToolkit in two separate domains using the following scenarios that correspond to labels in Tables 3, 4, and 5. (1) Domains Dom0 and Dom1 run on the same PCPU. (2) Domains Dom0 and Dom1 run on different PCPUs. We present results for all top-level functions that accounted for at least 1% of the total number of samples, sorted by decreasing

Function	1:D0	1:D1	2:D0	2:D1
main	0.98	0.99	0.95	0.98
global_opt	0.98	1	0.95	0.98
price_out_impl	0.98	1.01	0.95	0.99
primal_net_simplex	0.98	0.98	0.94	0.96
primal_bea_mpp	0.99	0.99	0.97	0.98
replace_weaker_arc	0.9	0.97	0.88	0.94
refresh_potential	0.96	0.98	0.88	0.9
update_tree	0.96	0.97	0.9	0.93
primal_iminus	1.56	1.56	1.36	1.51
insert_new_arc	1.2	1.13	1.02	1.13
flow_cost	0.94	0.93	0.94	0.94
dual_feasible	0.95	0.95	0.95	0.93
suspend_impl	0.91	0.9	0.9	0.89

Table 3. HPCToolkit profiling results for 429.mcf, sample count ratio virtualized/native (PAPI_TOT_CYC)

Function	1:D0	1:D1	2:D0	2:D1
main	1.01	1.01	0.99	1.01
global_opt	1.01	1.01	0.99	1.01
price_out_impl	1	1.01	0.98	1.01
primal_net_simplex	1.03	1.02	1.01	1.01
primal_bea_mpp	1.04	1.04	1.07	1.05
replace_weaker_arc	0.97	0.97	1	0.96
refresh_potential	1.04	1.01	0.9	1.01
update_tree	0.83	0.72	0.53	0.61
primal_iminus	4.5	4.75	3.5	4.83
insert_new_arc	0.27	0.46	0.3	0.22
flow_cost	0.97	1.03	0.91	0.94
dual_feasible	0.95	0.95	1.05	1
suspend_impl	1	0.94	1	1.06

Table 4. HPCToolkit profiling results for 429.mcf, sample count ratio virtualized/native (PAPI_L2_TCM)

Function	1:D0	1:D1	2:D0	2:D1
main	1.01	1.01	0.98	1.01
global_opt	1.01	1.02	0.98	1.02
price_out_impl	1.02	1.04	0.99	1.03
primal_net_simplex	1	1	0.97	1
primal_bea_mpp	1	0.98	0.97	1
replace_weaker_arc	0.99	1.05	1.04	1.09
refresh_potential	1.02	1.28	1.03	1.04
update_tree	0.86	0.75	0.84	0.86
primal_iminus	1.53	1.47	1.23	1.27
insert_new_arc	0.99	0.87	0.95	0.96
flow_cost	0.97	0.97	0.97	0.97
dual_feasible	1.03	1	1	1
suspend_impl	1.07	1	1	1.07

Table 5. HPCToolkit profiling results for 429.mcf, sample count ratio virtualized/native (PAPI_BR_INS)

number of samples. The tables show the ratio of the sample counts reported under virtualized vs. native execution. For most functions, similar counts were reported, although one (primal_iminus) shows significant differences which will warrant further investigation. We note that the same set of functions was identified in both execution modes, making the use of HPCToolkit in a virtualized environment a viable tool for identifying bottleneck functions that account for the largest proportion of events.

6. Related Work

There has been a number of previous efforts to add support for hardware event counters to virtualized environments. XenoProf [21], which is integrated in Xen, allows the use of event counters for system-wide monitoring and profiling. It is an extension of the OProfile Linux system-wide profiler. Each monitored domain runs an instance of OProfile with a Xen-specific driver, which communicates with XenoProf in the hypervisor. XenoProf collects PC samples and puts them into shared buffer. Then it notifies the corresponding domain via virtual interrupt, so that it can map the PC sample to a specific executable symbol. XenoProf does not support performance counter virtualization (i.e., the simultaneous monitoring of multiple domains), works only in paravirtualized mode, is specific to OProfile and cannot be easily adapted to work with other higher-level toolkits such as HPCToolkit.

Work concurrent with ours [15, 16] implemented performance counter virtualization for the hardware-assisted KVM virtual machine monitor that is included in recent versions of the Linux kernel. Like Xen's VPMU driver discussed in Section 4.2.5, their implementation uses a save-and-restore mechanism for PMU registers. During inter-domain context switches, the hypervisor saves and restores the PMU registers of a domain. The delivery of overflow interrupts to a domain relies on hardware support provided by architectural virtualization extensions. Such full virtualization approaches have the advantage that they do not require any accommodations to the guest kernel or user libraries, and thus allow the use of virtually any framework in the target domain, but they forgo the potential optimizations arising from guest kernel adaptation, such as the offsetting technique for a-mode counters discussed in Section 4.2.1. In addition, each instruction that changes a configuration register requires a separate trap to emulate its effect, whereas the use of hypercalls allows the batching of such changes by combining them into a single call. Finally, although hardware virtualization extensions are becoming increasingly common, some major IaaS cloud providers (e.g., Amazon EC2) still widely uses paravirtualized setups.

The VTSS++ system presented in [9] uses a system-wide global sampling mechanism that records time-stamped event counter values. The global TSC register is used to obtain these time stamps. In addition, the system records the time stamps of all intra- and inter-domain context switches. An off-line post-processing system then reconstructs the events produced by individual threads and domains. This method has the advantage that no guest kernel or user-level provisions are required, but its reliance on post-processing may make it unsuitable for some applications.

The vmkperf utility [4] used by VMWare ESX allows the counting of events occurring within given time intervals, similar to a-mode counters. vmkperf does not support the functionality of i-mode counters and therefore cannot easily be used to support high-level profiling toolkits.

The perf_events (previously known as perf_counter) framework [2] provides performance monitoring capabilities similar to those of *perfctr*. This framework has been integrated in recent versions of the Linux kernel. Like *perfctr*, it supports per-thread counters and direct user mode access. Higher-level frameworks such as PAPI include support for perf_events, although the recently added direct access feature [3] is not currently supported in PAPI. The techniques we presented in this paper are applicable to perf_events as well; a virtualization of perf_events is possible future work. The tight integration of perf_events with the Linux kernel may require a substantial amount of refactoring in order to create a hypervisor driver. This coupling may make it more difficult to exploit a single code base for the guest and hypervisor driver as done in our perfctr-xen implementation (see Section 4.2.4).

The perfmon framework [17] for Linux provides both low-level and high-level features. The framework supports per-thread monitoring and sampling, although it exclusively relies on system calls to access counter data. The Intel VTune performance analyzer [5] and AMD's Code Analyst [1, 14] are proprietary frameworks for precise, low-overhead event sampling. They consist of a kernel driver and high-level infrastructure that provides result analysis capabilities. Event counts can be viewed on per-thread or per-module basis. Our techniques could be applied to a possible virtualization of perfmon, VTune, and CodeAnalyst.

7. Conclusion

This paper presented perfctr-xen, a novel performance counter framework for the Xen hypervisor which we have developed. perfctr-xen extends the existing *perfctr* framework so it can be used in virtual machine environments running under the Xen hypervisor. perfctr-xen supports both paravirtualized guest and guests using hardware-based virtualization. It provides a hybrid mode in which paravirtualization techniques are applied to hardware-assisted guest virtual machine.

The technical contributions of this paper are the following: (1) application of an offsetting technique that allows direct access to logical per-thread counter values from user mode while avoiding the costs associated with saving and restoring physical PMU data registers; (2) the optimization of guest and hypervisor communication to minimize and amortize the costs associated with their coordination, while avoiding the costs of trapping and emulating counter-related instructions; (3) a technique for increasing the accuracy of performance monitoring by correcting for monitoring overhead.

Perfctr-xen enables the use of higher-level profiling frameworks such as PAPI or HPCToolkit in those environments, without requiring changes to them. As such, it addresses an urgent need in emerging IaaS cloud environments.

Acknowledgments

We thank the anonymous reviewers for their suggestions. This material is based upon work supported by the National Science Foundation (NSF) under Grant CSR–AES #0720673.

We released the code of our framework to the general public under an open source license. The latest version can be obtained at `http://people.cs.vt.edu/~rnikola/`.

References

[1] AMD CodeAnalyst. http://developer.amd.com/cpu/codeanalyst/, 2011.

[2] Performance counters for Linux. http://lwn.net/Articles/310176/, 2008.

[3] Perf_counter direct access support. http://lwn.net/Articles/323891/, 2009.

[4] Vmkperf utility for VMWare ESX 4.0, 2011.

[5] VTune amplifier profiler. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/, 2011.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[7] G. Back and D. S. Nikolopoulos. Application-specific system customization on many-core platforms: The VT-ASOS framework. In *STMCS: Second Workshop on Software Tools for Multi-Core Systems (STMCS)*, San Jose, CA, Mar. 2007.

[8] P. Barham, B. Dragovic, K. Fraser, and et al. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ISBN 1-58113-757-5.

[9] S. Bratanov, R. Belenov, and N. Manovich. Virtual machines: a whole new world for performance analysis. *SIGOPS Oper. Syst. Rev.*, 43: 46–55, April 2009.

[10] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[11] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997. ISSN 0734-2071. doi: 10.1145/265924.265930.

[12] W. Cohen. Multiple architecture characterization of the build process with OProfile, Red Hat. http://people.redhat.com/wcohen/wwc2003/, 2003.

[13] R. Creasy. The origin of the VM/370 time-sharing system. *Softw. World (UK)*, 13(1):4 – 10, 1982. ISSN 0038-0652.

[14] P. Drongowski, L. Yu, F. Swehosky, S. Suthikulpanit, and R. Richter. Incorporating instruction-based sampling into AMD CodeAnalyst. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 119 –120, Mar. 2010. doi: 10.1109/ISPASS.2010.5452049.

[15] J. Du, N. Sehrawat, and W. Zwaenepoel. Performance profiling in a virtualized environment. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, Berkeley, CA, USA, 2010. USENIX Association.

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

[17] S. Eranian. Perfmon2: A flexible performance monitoring interface for linux. In *Ottawa Linux Symposium*, pages 269–288, Ottawa, Canada, 2006.

[18] I. Habib. Virtualization with KVM. *Linux Journal*, 2008(166):8, 2008. ISSN 1075-3583.

[19] S. T. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 71–84, Berkeley, CA, USA, 2003. USENIX Association.

[20] A. Kivity. KVM: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[21] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 13–23, New York, NY, USA, 2005. ISBN 1-59593-047-7. doi: 10.1145/1064979.1064984.

[22] M. Pettersson. Perfctr library. http://user.it.uu.se/~mikpe/linux/perfctr/, 2011.

[23] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*, pages 129–144, 2000.

[24] M. Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5): 34–40, 2004.

[25] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.

[26] S. Shende and A. D. Malony. TAU: The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[27] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009).*, pages 23–32, Apr. 2009.