Diagnosing Performance Overheads in the Xen Virtual Machine Environment

Aravind Menon*
EPFL, Lausanne
aravind.menon@epfl.ch

Jose Renato Santos
HP Labs, Palo Alto
joserenato.santos@hp.com

Yoshio Turner
HP Labs, Palo Alto
yoshio.turner@hp.com

G. (John) Janakiraman
HP Labs, Palo Alto
john.janakiraman@hp.com

Willy Zwaenepoel
EPFL, Lausanne
willy.zwaenepoel@epfl.ch

ABSTRACT

Virtual Machine (VM) environments (e.g., VMware and Xen) are experiencing a resurgence of interest for diverse uses including server consolidation and shared hosting. An application's performance in a virtual machine environment can differ markedly from its performance in a non-virtualized environment because of interactions with the underlying virtual machine monitor and other virtual machines. However, few tools are currently available to help debug performance problems in virtual machine environments.

In this paper, we present Xenoprof, a system-wide statistical profiling toolkit implemented for the Xen virtual machine environment. The toolkit enables coordinated profiling of multiple VMs in a system to obtain the distribution of hardware events such as clock cycles and cache and TLB misses.

We use our toolkit to analyze performance overheads incurred by networking applications running in Xen VMs. We focus on networking applications since virtualizing network I/O devices is relatively expensive. Our experimental results quantify Xen's performance overheads for network I/O device virtualization in uni- and multi-processor systems. Our results identify the main sources of this overhead which should be the focus of Xen optimization efforts. We also show how our profiling toolkit was used to uncover and resolve performance bugs that we encountered in our experiments which caused unexpected application behavior.

Categories and Subject Descriptors

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

General Terms

Measurement, Performance

Keywords

Virtual machine monitors, performance analysis, statistical profiling

*Work done during internship at HP Labs and then at EPFL.

VEE'05, June 11-12, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

Virtual Machine (VM) environments are experiencing a resurgence of interest for diverse uses including server consolidation and shared hosting. The emergence of commercial virtual machines for commodity platforms [7], paravirtualizing open-source virtual machine monitors (VMMs) such as Xen [4], and widespread support for virtualization in microprocessors [13] will boost this trend toward using virtual machines in production systems for server applications.

An application's performance in a virtual machine environment can differ markedly from its performance in a non-virtualized environment because of interactions with the underlying VMM and other virtual machines. The growing popularity of virtual machine environments motivates deeper investigation of the performance implications of virtualization. However, few tools are currently available to analyze performance problems in these environments.

In this paper, we present Xenoprof, a system-wide statistical profiling toolkit implemented for the Xen virtual machine environment. The Xenoprof toolkit supports system-wide coordinated profiling in a Xen environment to obtain the distribution of hardware events such as clock cycles, instruction execution, TLB and cache misses, etc. Xenoprof allows profiling of concurrently executing domains (Xen uses the term "domain" to refer to a virtual machine) and the Xen VMM itself at the fine granularity of individual processes and routines executing in either the domain or in the Xen VMM. Xenoprof will facilitate a better understanding of the performance characteristics of Xen's mechanisms and thereby help to advance efforts to optimize the implementation of Xen. Xenoprof is modeled on the OProfile [1] profiling tool available on Linux systems.

We report on the use of Xenoprof to analyze performance overheads incurred by networking applications running in Xen domains. We focus on networking applications because of the prevalence of important network intensive applica-

tions, such as web servers and because virtualizing network I/O devices is relatively expensive. Previously published papers evaluating Xen's performance have used networking benchmarks in systems with limited network bandwidth and high CPU capacity [4, 10]. Since this results in the network, rather than the CPU, becoming the bottleneck resource, in these studies the overhead of network device virtualization does not manifest in reduced throughput. In contrast, our analysis examines cases where throughput degrades because CPU processing is the bottleneck instead of the network. Our experimental results quantify Xen's performance overheads for network I/O device virtualization in uni- and multi-processor systems. Our results identify the main sources of this overhead which should be the focus of Xen optimization efforts. For example, Xen's current I/O virtualization implementation can degrade performance by preventing the use of network hardware offload capabilities such as TCP segmentation and checksum.

We additionally show a small case study that demonstrates the utility of our profiling toolkit for debugging real performance bugs we encountered during some of our experiments. The examples illustrate how unforeseen interactions between an application and the VMM can lead to strange performance anomalies.

The rest of the paper is organized as follows. We begin in Section 2 with our example case study motivating the need for performance analysis tools in virtual machine environments. Following this, in Section 3 we describe aspects of Xen and OProfile as background for our work. Section 4 describes the design of Xenoprof. Section 5 shows how the toolkit can be used for performance debugging of the motivating example of Section 2. In Section 6, we describe our analysis of the performance overheads for network I/O device virtualization. Section 7 describes related work, and we conclude with discussion in Section 8.

2. MOTIVATING EXAMPLE

Early on in our experimentation with Xen, we encountered a situation in which a simple networking application which we wrote exhibited puzzling behavior when running in a Xen domain. We present this example in part to illustrate how the behavior of an application running in a virtual machine environment can differ markedly and in surprising ways from its behavior in a non-virtualized system. We also show the example to demonstrate the utility of Xenoprof, which as we describe later in Section 5 we used to help us diagnose and resolve this problem.

The example application consists of two processes, a sender and a receiver, running on two Pentium III machines connected through a gigabit network switch. The two processes are connected by a single TCP connection, with the sender continuously sending a 64 KB file (cached in the sender's memory) to the receiver using the zero-copy `sendfile()` system call. The receiver simply issues receives for the data as fast as possible.

In our experiment, we run the sender on Linux (non-virtualized), and we run the receiver either on Linux or in a Xen domain. We measure the receiver throughput as we vary the size of the user-level buffer which the application passes to the socket receive system call. Note that we are not varying the receiver socket buffer size which limits kernel memory usage. Figure 1 shows that when the receiver is running on Linux, its throughput is nearly independent of the application receive buffer size. In contrast, throughput is highly sensitive to the application buffer size when the receiver runs in the Xen environment.

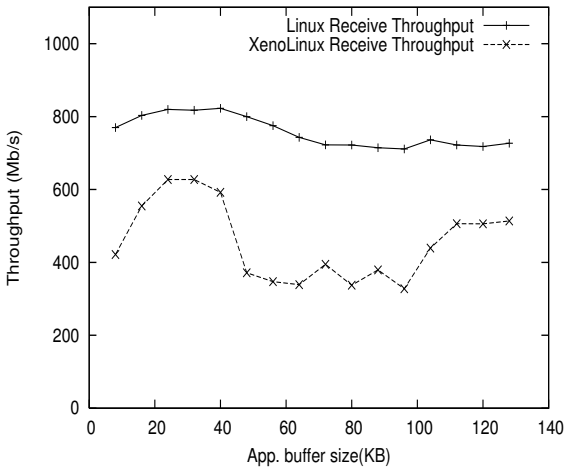

Figure 1: XenoLinux performance anomaly: Network receive throughput in XenoLinux shows erratic behavior with varying application buffer size

This is one prime example of a scenario where we would like to have a performance analysis tool which can pinpoint the source of unexpected behavior in Xen. Although system-wide profiling tools such as OProfile exist for single OSes like Linux, no such tools are available for a virtual machine environment. Xenoprof is intended to fill this gap.

3. BACKGROUND

In this section, we briefly describe the Xen virtual machine monitor and the OProfile tool for statistical profiling on Linux. Our description of their architecture is limited to the aspects that are relevant to the discussion of our statistical profiling tool for Xen environments and the discussion of Xen's performance characteristics. We refer the reader to [4] and [1] for more comprehensive descriptions of Xen and OProfile, respectively.

3.1 Xen

Xen is an open-source x86 virtual machine monitor (VMM) that allows multiple operating system (OS) instances to run concurrently on a single physical machine. Xen uses paravirtualization [25], where the VMM exposes a virtual machine abstraction that is slightly different from the underlying hardware. In particular, Xen exposes a hypercall mechanism that virtual machines must use to perform privileged operations (e.g., installing a page table), an event notification mechanism to deliver virtual interrupts and other notifications to virtual machines, and a shared-memory based device channel for transferring I/O messages among virtual machines. While the OS must be ported to this virtual machine interface, this approach leads to better performance than an approach based on pure virtualization.

The latest version of the Xen architecture [10] introduces a new I/O model, where special privileged virtual machines called *driver domains* own specific hardware devices and run their I/O device drivers. All other domains (*guest domains* in Xen terminology) run a simple device driver that com-

municates via the device channel mechanism with the driver domain to access the real hardware devices. Driver domains directly access hardware devices that they own; however, interrupts from these devices are first handled by the VMM which then notifies the corresponding driver domain through *virtual interrupts* delivered over the event mechanism. The guest domain exchanges service requests and responses with the driver domain over an I/O descriptor ring in the device channel. An asynchronous inter-domain event mechanism is used to send notification of queued messages. To support high-performance devices, references to page-sized buffers are transferred over the I/O descriptor ring rather than actual I/O data (the latter would require copying). When data is sent by the guest domain, Xen uses a sharing mechanism where the guest domain permits the driver domain to map the page with the data and pin it for DMA by the device. When data is sent by the driver domain to the guest domain, Xen uses a page-remapping mechanism which maps the page with the data into the guest domain in exchange for an unused page provided by the guest domain.

3.2 OProfile

OProfile [1] is a system-wide statistical profiling tool for Linux systems. OProfile can be used to profile code executing at any privilege level, including kernel code, kernel modules, user level applications and user level libraries.

OProfile uses performance monitoring hardware on the processor to collect periodic samples of various performance data. Performance monitoring hardware on modern processor architectures include counters that track various processor events including clock cycles, instruction retirements, TLB misses, cache misses, branch mispredictions, etc. The performance monitoring hardware can be configured to notify the operating system when these counters reach specified values. OProfile collects the program counter (PC) value at each notification to obtain a statistical sampling. For example, OProfile can collect the PC value whenever the clock cycle counter reaches a specific count to generate the distribution of time spent in various routines in the system. Likewise, OProfile can collect the PC value whenever the TLB miss counter reaches a specific count to generate the distribution of TLB misses across various routines in the system. The accuracy and overhead of the profiling can be controlled by adjusting the sampling interval (i.e., the count when notifications are generated).

On x86 systems, OProfile configures the performance monitoring hardware to raise a Non-Maskable Interrupt (NMI) whenever a hardware performance counter reaches its threshold. The use of the NMI mechanism for notification is key to accurate performance profiling. Since the NMI cannot be masked, the counter overflow will be serviced by the NMI handler without any delay allowing the program counter to be read immediately and associated with the sample. If a maskable interrupt is used to notify counter overflow, the interrupt can be masked which will delay the recording of the program counter causing the sample to be associated with a wrong process or routine (e.g., if a context switch occurs between the overflow of the counter, and the actual collection of the sample). Furthermore, since the NMI will be serviced even in the middle of other interrupt handlers, use of the NMI allows the profiling of interrupt handlers.

Profiling with OProfile operates as follows (simplified):

1. User provides input to OProfile about the performance events to be monitored and the periodic count.
2. OProfile programs hardware counters to count the user-specified performance event and to generate an NMI when the counter has counted to the user-specified count.
3. The performance monitoring hardware generates an NMI upon counter overflow.
4. OProfile's NMI handler catches the NMI and records the program counter value in a kernel buffer.
5. OProfile processes the buffer periodically to determine the routine and executable corresponding to the program counter on each sample in the buffer. This is determined by consulting the virtual memory layout of the process and the kernel.

4. STATISTICAL PROFILING IN XEN

In this section, we describe Xenoprof, a system-wide statistical profiling toolkit we have developed for Xen virtual machine environments. The Xenoprof toolkit provides capabilities similar to OProfile for the Xen environment (i.e., using performance monitoring hardware to collect periodic samples of performance data). The performance of applications running on Xen depend on interactions among the application's processes, the operating system it is running on, the Xen VMM, and potentially other virtual virtual machines (e.g., driver domain) running on the same system. In order to study the costs of virtualization and the interactions among multiple domains, the performance profiling tool must be able to determine the distribution of performance events across routines in the Xen VMM and all the domains running on it.

The Xenoprof toolkit consists of a VMM-level layer (we hereon refer to this layer as Xenoprof) responsible for servicing counter overflow interrupts from the performance monitoring hardware and a domain-level layer derived from OProfile responsible for attributing samples to specific routines within the domain. The OProfile layer drives the performance profiling through hypercalls supported by Xenoprof and Xenoprof delivers samples to the OProfile layer using Xen's virtual interrupt mechanism. System-wide profiling is realized through the coordination of multiple domain-level profilers. While our current implementation is based on OProfile, other statistical profilers (e.g., VTune [3]) can be ported to use the Xenoprof interface.

We first describe the issues that guided our design choices for Xenoprof, followed by a description of the Xenoprof framework itself. We then describe the modifications to standard domain level profilers such as OProfile that are necessary to enable them to be used with Xenoprof for providing system-wide profiling in Xen.

4.1 Design choice for profiling in Xen

The main issue with profiling virtual machine environments is that profiling cannot be centralized. System-wide profiling in any environment requires the knowledge of detailed system level information. As an example, we saw in Section 3.2 that in order to account a PC sample to the correct routine and process, OProfile needs to consult the virtual memory layout of the current process, or the kernel. In a single-OS case like OProfile, all the required system level information is available in one centralized place, the kernel. Unfortunately, in a virtual machine environment like Xen,

the information required for system-wide profiling is spread across multiple domains, and this domain-level information is not usually accessible from the hypervisor. For instance, Xen cannot determine the current process running in a domain, or determine its memory layout in order to find the routine corresponding to a PC sample collected for profiling. Thus, system-wide profiling in Xen is not possible with the profiler executing solely inside Xen.

Even if the domain's kernel symbol map is available to Xen, it would still be difficult for Xen to determine the symbol map for processes running in the domain.

Thus, system-wide profiling in Xen must be "distributed" in nature, with help required from domain specific profilers for handling the profiling of the individual domains. Most of the domain specific profiling responsibilities are delegated to domain specific profilers, and a thin paravirtualized interface, Xenoprof, is provided for coordinating their execution, and providing them access to the hardware counters.

4.2 Xenoprof Framework

Xenoprof helps realize system-wide profiling in Xen by the coordination of domain specific profilers executing in each domain. Xenoprof uses the same hardware counter based sampling mechanism as used in OProfile. It programs the hardware performance counters to generate sampling interrupts at regular event count intervals. Xenoprof hands over the PC samples collected on counter overflow to the profiler running in the current domain for further processing. For system-wide profiling, all domains run a Xenoprof compliant profiler, thus no PC samples are lost. Xenoprof also allows the selective profiling of a subset of domains.

Xenoprof allows domain level profilers to collect PC samples for performance counter overflows which occur in the context of that domain. Since these PC samples may also include samples from the Xen address space, the domain profilers must be extended to recognize and correctly attribute Xen's PC samples. Xenoprof also allows domain level profilers to optionally collect PC samples for other "passive" domains. A possible use of this feature is when the passive domain does not have a Xenoprof-compliant profiler, but we are still interested in estimating the aggregate execution cost accruing to the domain.

Domain level profilers in Xenoprof operate in a manner mostly similar to their operation in a non-virtualized setup. For low level operations, such as accessing and programming performance counters, and collecting PC samples, the domain profilers are modified to interface with the Xenoprof framework. The high level operations of the profiler remain mostly unchanged. After collecting PC samples from Xenoprof, the profiler accounts the sample to the appropriate user or kernel routine by consulting the required memory layout maps. At the end of profiling, each domain profiler has a distribution of hardware event counts across routines in its applications, the kernel or Xen for its domain. The global system profile can be obtained by simply merging the individual profiles. The detailed profile for Xen can be obtained by merging the Xen components of the profiles from the individual domains.

The Xenoprof framework defines a paravirtualized interface to support domain level profilers in the following respects:

1. It defines a performance event interface which maps performance events to the physical hardware counters, and defines a set of functions for domain level profilers to specify profiling parameters, to start and to stop profiling.

2. It allows domain-level profilers to register virtual interrupts that will be triggered whenever a new PC sample is available. Unlike sampling in OProfile, the domain level profilers in Xenoprof do not collect PC samples themselves. Instead, Xenoprof collects it on their behalf. This is because it may not be possible to call the domain level virtual interrupt handler synchronously from any context in Xen, whenever a hardware counter overflows. Since PC samples must be collected synchronously to be correct, Xenoprof collects the samples on behalf of the domain and logs them into the per-domain sample buffer of the appropriate domain. The domain profiler is notified of additional samples through a virtual interrupt.

3. It coordinates the profiling of multiple domains. In every profiling session there is a distinguished domain, the initiator, which is responsible for configuring and starting a profiling session. The initiator domain has the privileges to decide the profiling setup, including hardware events to profile, domains to profile, etc. After the initiator selects the set of domains to be profiled, Xenoprof waits until all participant domains perform a hypercall indicating that they are ready to process samples. The profiling session starts only after this condition is satisfied and after a start command is issued by the initiator. Profiling stops when requested by the initiator. The sequencing of profiling operations across the initiator and other domains is orchestrated by the user (e.g., using a user-level script).

In the current architecture, only the privileged domain, domain0, can act as the initiator for a profiling session involving more than one domain. Thus, only the privileged domain 0 can control system-wide profiling involving multiple domains. This restriction is necessary because allowing unprivileged domains to do system-wide profiling would violate the security and isolation guarantees provided by Xen.

Xenoprof allows unprivileged domains to do single-domain profiling of the domain itself. In this scenario, the domain can start profiling immediately, and no coordination between domains is involved.

One restriction in the current architecture of Xenoprof is that it does not allow concurrent independent profiling of multiple domains. Concurrent independent profiling of different domains would require support for per-domain virtual counters in Xenoprof. This is currently not supported.

4.3 Porting OProfile to Xenoprof

Domain level profilers must be ported to the Xenoprof interface so that they can be used in the Xen environment. Porting a profiler to Xenoprof is fairly straightforward, and entails the following steps:

1. The profiler code for accessing and programming the hardware counters must be modified to use the Xenoprof virtual event interface.

2. Before starting profiling, the profiler queries Xenoprof to determine whether it is to take on the role of the initiator. Only the initiator domain performs the global profiling setup, such as deciding the events and profiling duration. Both the initiator and the other profilers register their callback functions for collecting PC samples.

3. The profiler is extended with access to the Xen virtual memory layout, so that it can identify Xen routines corresponding to PC samples in Xen's virtual address range.

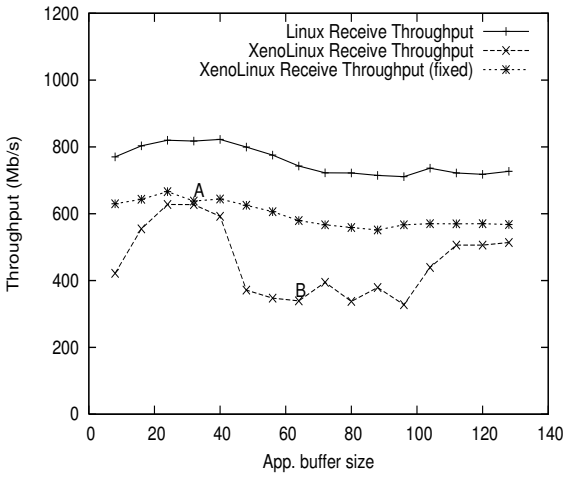

Figure 2: Xenoprof network performance anomaly

We ported OProfile to use the Xenoprof interface in the above respects.

5. PERFORMANCE DEBUGGING USING XENOPROF

In this section, we analyze the performance anomaly described in Section 2 using our Xenoprof based profiling tool. To recall briefly, the performance bug consists of the erratic variation in TCP receive throughput as a function of the application level buffer size, when the application runs in a Xen driver domain on a Pentium III machine. We reproduce the graph of the receiver throughput versus application buffer size in Figure 2.

To understand the variation in receiver throughput running in Xen, we profile the receiver at two points that have widely different performance (points A and B in Figure 2 with high and low throughput, respectively). Table 1 shows the distribution of execution time for points A and B across the XenoLinux kernel (Linux ported to the Xen environment), the network driver and Xen. The table shows that when the throughput is low at point B, a significantly greater fraction of the execution time is spent in the XenoLinux kernel, compared to point A, which has higher throughput.

	Point A	Point B
XenoLinux kernel	60	**84**
network driver	10	5
xen	30	11

Table 1: Xen domain0 Profile

The additional kernel execution overhead at B has a function-wise breakdown as shown in table 2. Table 2 shows that point B suffers significantly lower throughput compared to A, because of the additional time (roughly 40%) spent in the kernel routines `skb_copy_bits`, `skbuff_ctor` and `tcp_collapse`. The corresponding overheads for these functions at point A are nearly zero.

The biggest overhead at point B comes from the function `skb_copy_bits`. `skb_copy_bits` is a simple data copying function in the Linux kernel used to copy data from a socket buffer to another region in kernel memory.

	Point A	Point B
skb_copy_bits	0.15	**28**
skbuff_ctor	absent	**9**
tcp_collapse	0.05	**3**
other routines	99.8	60

Table 2: XenoLinux Profile

The second function, `skbuff_ctor`, is a XenoLinux specific routine which is called whenever new pages are added to the socket buffer slab cache. This function zeroes out new pages before adding them to the slab cache, which can be an expensive operation. XenoLinux clears out pages added to its slab cache to make sure that there are no security breaches when these pages are exchanged with other domains.

`tcp_collapse` is a routine in Linux which is called for reducing the per-socket TCP memory pressure in the kernel. Whenever a socket's receive queue memory usage exceeds the per-socket memory limits, `tcp_collapse` is invoked to free up fragmented memory in the socket's receive queue. (Note that this deals with per-socket memory limits, not the entire kernel's TCP memory usage). The `tcp_collapse` function tries to reduce internal fragmentation in the socket's receive queue by copying all the socket buffers in the receive queue into new compact buffers. It makes use of the `skb_copy_bits` function for compacting fragmented socket buffers. Frequent compaction of socket buffers can be an expensive operation.

The high execution cost of `tcp_collapse` and `skb_copy_bits` at point B indicates that the kernel spends a significant fraction of the time compacting fragmented socket buffers for reducing TCP memory pressure. This is suspicious, because in the normal scenario, most of the socket buffers in a socket's receive queue have a size equal to the Maximum Transfer Unit (MTU) size and do not suffer much internal fragmentation.

To check if socket buffer fragmentation is indeed the issue here, we instrument the XenoLinux kernel to determine the average internal fragmentation in the socket buffers. Kernel instrumentation reveals that on an average, each socket buffer is allocated 4 KB (i.e. one page size), out of which it uses only 1500 bytes (MTU) for receiving data. This obviously leads to internal fragmentation, and causes the per-socket memory limit to be frequently exceeded, leading to significant time being wasted in defragmenting the buffers.

It turns out that for every 1500 byte socket buffer requested by the network driver, XenoLinux actually allocates an entire page, i.e. 4 KB. This is done to facilitate the transfer of a network packet between an I/O domain and a guest domain. A page sized socket buffer facilitates easy transfer of ownership of the page between the domains. Unfortunately, this also leads to significant wastage of per-socket memory, and as we see in the analysis above, this can result in unexpected performance problems.

The performance problem resulting from socket memory exhaustion can be artificially fixed by using the kernel parameter `tcp_adv_window_scale`. This kernel parameter determines the fraction of the per-socket receive memory which is set aside for user-space buffering. Thus, by reserving a large fraction of the socket's memory for user buffering, the effective TCP window size of the socket can be reduced, and thus the kernel starts dropping packets before it leads to socket memory exhaustion.

With this configuration in place, the performance problem arising due to socket buffer compaction is eliminated and we get more consistent receiver performance. This improved scenario is shown in Figure 2 as the line corresponding to the "fixed" Xenoprof configuration. The overhead of virtualization is approximately constant across the range of buffer sizes, leading to similar throughput behavior for this configuration and the base Linux configuration.

6. PERFORMANCE OVERHEADS IN XEN

In this section, we compare the performance of three networking applications running in the Xen virtual environment with their performance in a non-virtualized Linux environment. The three applications provide a thorough coverage of various performance overheads arising in a virtualized network environment.

The three benchmarks we use stress different aspects of the network stack. The first two are simple microbenchmarks, similar to the TTCP [2] benchmark, and measure the throughput of TCP send and receive over a small number of connections (one per network interface card). The third benchmark consists of a full fledged web server running a simple synthetic workload. These benchmarks stress the TCP send half, TCP receive half, and concurrent TCP connections respectively.

We use Xenoprof to analyze and evaluate the performance of the three benchmarks in non-virtualized Linux (single-CPU) and also in three different Xen configurations:

1) Xen-domain0: Application runs in a single privileged driver domain.

2) Xen-guest0: A guest domain configuration in which an unprivileged guest domain and a privileged driver domain run on the same CPU. The application and a virtual device driver run in the guest domain, and the physical device driver (serving as the backend for the guest) runs in the driver domain and interacts directly with the I/O device.

3) Xen-guest1: Similar to Xen-guest0 configuration except the guest and driver domains each run on a separate CPU.

Our experimental setup is the following. For experiments that do not include the Xen-guest1 configuration (Sections 6.1.1 and 6.2), we use a Dell PowerEdge 1600SC, 2393 MHz Intel Xeon server, with 1 GB of RAM and four Intel Pro-1000 gigabit ethernet cards. The server is connected over a gigabit switch to multiple client machines having similar configurations and with one gigabit ethernet card each. For experiments that include Xen-guest1 (Sections 6.1.2 and 6.3, and for the web server benchmark, we use a Compaq Proliant DL360 dual-CPU 2800 MHz Intel Xeon server, with 4 GB of RAM, and one Broadcom 'Tigon 3' gigabit ethernet card. The server is connected to one client machine with similar configuration over a gigabit switch.

Xen version 2.0.3 is used for all experiments, and the domains run XenoLinux derived from stock Linux version 2.6.10. All domains are configured to use 256 MB of RAM.

In the following subsections, we describe our evaluation and analysis of Xen for the three benchmarks.

6.1 Receiver workload

The TCP *receiver* microbenchmark measures the TCP receive throughput achievable over a small number of TCP connections. This benchmark stresses the receive half of the TCP stack for large data transfers over a small number of connections.

The benchmark consists of a number of *receiver* processes, one per NIC (network interface card), running on the target machine. Each *receiver* is connected to a *sender* process running on a different machine over a separate NIC. *Sender* processes use `sendfile()` to send a cached 64 KB file to the corresponding *receiver* process continuously in a loop. We measure the maximum aggregate throughput achievable by the *receivers*. The *sender* processes each run on a different machine to make sure that they do not become a bottleneck.

6.1.1 Xen-domain0 configuration

We first compare the performance of the *receiver* in the Xen-domain0 configuration with its performance in a baseline Linux system. In the next subsection, we will evaluate its performance in Xen guest domain configurations. Figure 3 compares the Xen-domain0 and Linux performance for different number of NICs.

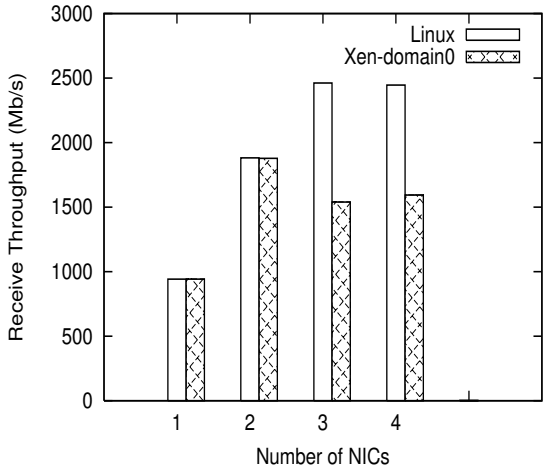

Figure 3: Receive throughput on Xen domain0 and Linux

In Linux, the *receivers* reaches an aggregate throughput of 2462 Mb/s with 3 NICs, whereas in Xen-domain0 reaches 1878 Mb/s with 2 NICs.

This result contrasts with previous research [4], which claims that Xen-domain0 achieves performance nearly equal to the performance of a baseline Linux system. Those experiments used only a single NIC which became the bottleneck resource. Since the system was not CPU-limited, higher CPU utilization with Xen was not reflected in reduced throughput. With more NICs, as the system becomes CPU-limited, performance differences crop up.

We profile the Xen-domain0 configuration and Linux for different hardware events in the 3 NIC run. We first compare the aggregate hardware event counts in the two configurations for the following hardware events: instruction counts, L2 cache misses, data TLB misses, and instruction TLB misses. Figure 4 shows the normalized values for these events relative to the Xen-domain0 numbers.

Figure 4 shows that a primary cause for reduced throughput in Xen-domain0 is the significantly higher data and instruction TLB miss rates compared to Linux. We can compare configurations in terms of the ratio of their cache or

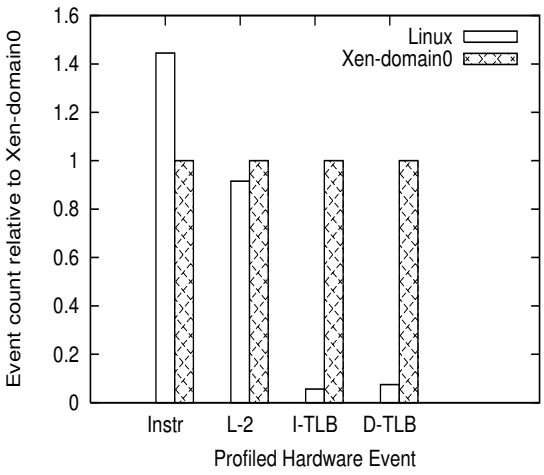

Figure 4: Relative hardware event counts in Xen-domain0 and Linux for receive (3 NICs)

TLB *miss rates* (we define miss rate as the ratio of number of misses to number of instructions executed). Compared to Linux, Xen-domain0 has a data TLB miss count roughly 13 times higher (which works out to 19 times higher data TLB miss rate), and instruction TLB miss count roughly 17 times higher (or 25 times higher miss rate). Higher TLB misses leads to instruction stalling, and at full utilization this results in the instruction count for Xen-domain0 being only about 70% of the instruction count for Linux, leading to a similar decrease in achieved throughput.

TLB misses in Xen-domain0 are not concentrated in a few hotspot functions, but are spread across the entire TCP receive code path, as shown in Table 3. From preliminary investigations based on Xen code instrumentation, we suspect the increase in misses is caused by increases in working set size as opposed to TLB flushes, but further investigation is needed.

% D-TLB miss	Function	Module
9.48	e1000_intr	network driver
7.69	e1000_clean_rx_irq	network driver
5.81	alloc_skb_from_cache	XenoLinux
4.45	ip_rcv	XenoLinux
3.66	free_block	XenoLinux
3.49	kfree	XenoLinux
3.32	tcp_preque_process	XenoLinux
3.01	tcp_rcv_established	XenoLinux

Table 3: Data TLB miss distribution in Xen-domain0 (3 NICs)

In addition to comparing two configurations in terms of miss rates, which determine the time required to execute each instruction, we can also evaluate the efficiency of two configurations in terms of their *instruction cost*, which we define as the number of instructions that must be executed to transfer and process a byte of data to or from the network. By dividing the ratio of instructions executed in the two configurations by the ratio of throughput achieved in the two configurations, we get the *instruction cost ratio* for the two configurations. Comparing Xen-domain0 to Linux, 0.6919 (the ratio of instructions executed, Figure 4) divided by 0.6251 (throughput ratio, Figure 3) yields instruction cost ratio 1.11, indicating that Xen-domain0 requires 11% more instructions to process each byte compared to Linux. The *receiver* in the Xen-domain0 configuration must execute additional instructions within Xen and also Xen specific routines in XenoLinux.

The execution of Xen is roughly 14% of CPU cycles. Table 4 shows the cost of some specific routines in Xen and XenoLinux which contribute to execution overhead.

% Execution time	Function	Module
9.15	skbuff_ctor	XenoLinux
6.01	mask_and_ack_irq	Xen
2.81	unmask_irq	Xen

Table 4: Expensive routines in Xen-domain0 (3 NICs)

The `skbuff_ctor` routine, discussed in Section 5, is a XenoLinux specific routine which can be quite expensive depending on the frequency of socket buffer allocation (9% of CPU time in this case). Masking and unmasking of interrupts in Xen contributes an additional 9% of CPU overhead not present in Linux. For each packet received, the network device raises a physical interrupt which is first handled in Xen, and then the appropriate "event" is delivered to the correct driver domain through a virtual interrupt mechanism. Xen masks level triggered interrupts to avoid unbounded reentrancy since the driver domain interrupt handler resets the interrupt source outside the scope of the physical interrupt handler [10]. The domain interrupt handler has to make an additional hypercall to Xen to unmask the interrupt line after servicing the interrupt.

In summary,

1. The maximum throughput achieved in the Xen-domain0 configuration for the *receiver* benchmark is roughly 75% of the maximum throughput in Linux.

2. Xen-domain0 suffers a significantly higher TLB miss rate compared to Linux, which is the primary cause for throughput degradation.

3. Other overheads arise from interrupt handling overheads in Xen, and Xen specific routines in XenoLinux.

6.1.2 Guest domain configurations

We next evaluate the TCP *receiver* benchmark in the guest domain configurations Xen-guest0 and Xen-guest1, and compare performance with the Xen-domain0 configuration. Our experiments show that *receiver* performance in Xen-guest0 is significantly lower than in the Xen-domain0 configuration. Whereas the Xen-domain0 *receiver* achieves 1878 Mb/s, and Linux *receiver* achieves 2462 Mb/s, the *receiver* running in Xen-guest0 achieves 849 Mb/s, less than the capacity of a single NIC. Thus, for the remainder of this section, we focus our analysis for the guest domains only for the single NIC configuration, where Xen-domain0 and Xen-guest1 both deliver 941 Mb/s and Xen-guest0 achieves only 849 Mb/s.

The primary cause of throughput degradation in the guest domain configurations is the significantly higher instruction counts relative to Xen-domain0. Figure 5 shows normalized aggregate counts for various performance events rela-

tive to the Xen-domain0 values. We can see that compared to Xen-domain0, the Xen-guest0 and Xen-guest1 configurations have higher instruction cost ratios (2.25 and 2.52, respectively), higher L2 cache miss rates (1.97 times higher and 1.52 times higher), lower instruction TLB miss rates (ratio 0.31 and 0.72), and lower data TLB miss rates (ratio 0.54 and 0.87).

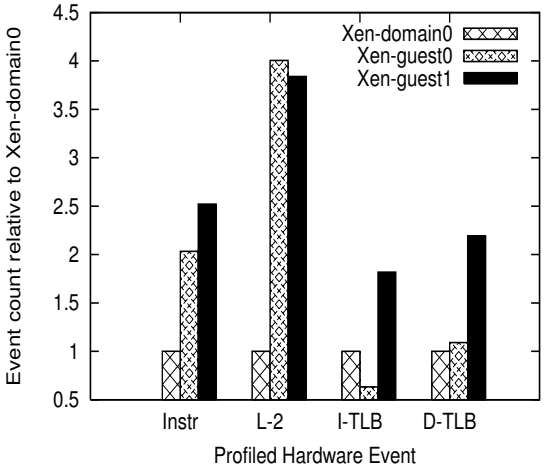

Figure 5: Relative hardware event counts in Xen-domain0, Xen-guest0 (1CPU) and Xen-guest1 (2CPUs)

Table 5 gives the breakdown of instruction counts across the guest and driver domains and Xen for the three configurations. The table shows that compared to Xen-domain0, the guest domain configurations execute considerably more instructions in Xen and a large number of instructions in the driver domain even though it only hosts the physical driver.

	Xen-domain0	Xen-guest0	Xen-guest1
Guest domain	N/A	16453	20145
Driver domain	19104	15327	18845
Xen	1939	11733	14975

Table 5: Instruction execution samples in Xen configurations

Detailed profiling of Xen execution overhead, shown in Table 6, shows that most of the instruction overhead of Xen comes from the page remapping and page transfer between the driver and guest domains for network data received in the driver domain. For each page of network data received, the page is remapped into the guest domain and the driver domain acquires a replacement page from the guest.

The second source of overhead is within the driver domain. Most of the overhead of the driver domain, as shown in Table 7, comes from the network filtering and bridging code. Xen creates a number of virtual network interfaces for guest domains, which are bridged to the physical network interface in the driver domain using the standard Linux bridging code[1]

[1]As an alternative to bridging, Xen recently added a routing-based solution. We have not yet evaluated this approach.

% Instructions	Xen function
3.94	__copy_from_user_ll
3.02	do_mmu_update
2.59	__copy_to_user_ll
2.50	mod_l1_entry
1.67	alloc_domheap_pages
1.47	do_extended_command
1.43	free_domheap_pages

Table 6: Instruction count distribution in Xen (Xen-guest0 configuration)

% Instructions	Driver domain function
5.14	nf_hook_slow
3.59	nf_iterate
1.76	br_nf_pre_routing
1.17	net_rx_action
1.31	fdb_insert
1.00	br_handle_frame

Table 7: Instruction count distribution in driver domain (Xen-guest0 configuration)

Apart from these two major sources of overhead, Figure 5 shows that the guest configurations also suffer significantly higher L2 cache misses, roughly 4 times the misses in Xen-domain0. This may be explained by the fact that executing two domains concurrently increases the working set size and reduces locality, leading to more L2 misses. One surprising result from Figure 5 is that Xen-guest0 suffers lower TLB misses compared to Xen-guest1, even though Xen-guest0 involves switching between the guest and driver domains on the same CPU.

In summary,

1. The *receiver* benchmark in the guest domain configurations achieves less than half the throughput achieved in the Xen-domain0 configuration.

2. The main reasons for reduced throughput are the computational overheads of the hypervisor and the driver domain, which cause the instruction cost to increase by a factor of 2.2 to 2.5.

3. Increased working set size and reduced locality lead to higher L2 misses in the guest configurations.

6.2 Sender workload

We now evaluate the different Xen configurations using our second microbenchmark, the TCP *sender* benchmark. This is similar to the *receiver* benchmark, except that we measure the throughput on the *sender* side of the TCP connections. We run multiple *sender* processes, one per NIC, on the target machine, and run each *receiver* process on a separate machine. This benchmark stresses the send control path of the TCP stack for large data transfers.

We first compare the performance of the *sender* running in Xen-domain0 with its performance in Linux. We observe that, for our system configuration, the *sender* workload is very light on the CPU, and both Linux and Xen-domain0 are able to saturate up to 4 NICs, achieving an aggregate throughput of 3764 Mb/s, without saturating the CPU. Thus, we observe no throughput differences up to 4 NICs. However as in the *receiver* benchmark, the Xen-

domain0 configuration suffers significantly higher TLB miss rates compared to Linux, and shows greater CPU utilization for the same workload.

We next evaluate the TCP *sender* benchmark in the guest domain configuration, Xen-guest0. As TCP send is cheaper compared to TCP receive, we expected the *sender* in the guest configuration to perform better than the *receiver*. However, we found that maximum throughput achievable by the *sender* in the Xen-guest0 configuration is only 706 Mb/s, whereas in the Xen-domain0 configuration, the *sender* is able to achieve up to 3764 Mb/s.

Figure 6 compares the hardware event counts for Xen-guest0 and the event counts for the *sender* running in Xen-domain0 with one NIC. The figure demonstrates that the primary cause for throughput degradation in Xen-guest0 is higher instruction cost. The Xen-guest0 configuration incurs a factor of 6 times higher instruction count (8.15 times higher instruction cost) than the Xen-domain0 configuration. The L2 miss rate is also a factor of 4 higher in the Xen-guest0 configuration than the Xen-domain0 configuration, which can be attributed to contention for the L2 cache between the guest domain and the driver domain.

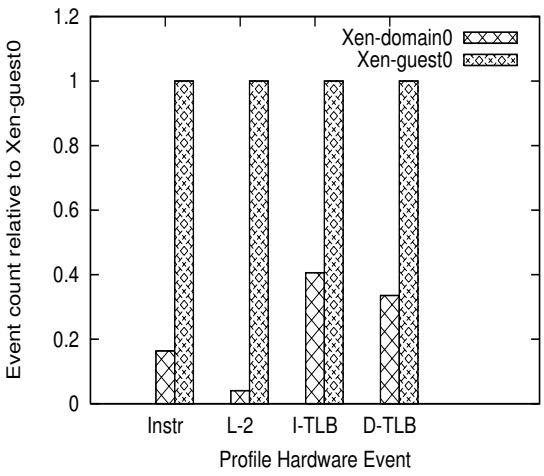

Figure 6: Relative hardware event counts in Xen-domain0 and Xen-guest0 for sender benchmark

We study the distribution of instructions among the routines in the domain hosting the sender application, i.e., the guest domain of the Xen-guest0 configuration and the driver domain of the Xen-domain0 configuration. The instruction counts for these domains differs by a factor of 2.8 (as compared to factor of 6 overall between the two configurations).

Function	Xen-guest0 guest	Xen-domain0
tcp_write_xmit	212	72
tcp_transmit_skb	**512**	**92**
ip_queue_xmit	577	114

Table 8: Instruction execution samples in Xen-guest0 guest and Xen-domain0

Table 8 shows the cost of high-level TCP/IP protocol processing functions. The Xen-guest0 guest has significantly higher instruction counts for these functions, and although not shown, this leads to similar increases for all descendant functions. It appears as if the TCP stack in the guest configuration processes a larger number of packets compared to Xen-domain0 to transfer the same amount of data. Further analysis (from kernel instrumentation) reveals that this is indeed the case, and this in fact results from the absence of TCP segmentation offload (TSO) support in the guest domain's virtual NIC.

TSO support in a NIC allows the networking stack to bunch together a large number of socket buffers into one big packet. The networking stack can then do TCP send processing on this single packet, and can offload the segmentation of this packet into MTU sized packets to the NIC. For large data transfers, TSO support can significantly reduce computation required in the kernel.

For the *sender* workload, Xen-domain0 can directly make use of TSO support in the NIC to offload its computation. However, the guest domain is forced to perform more TCP processing for MTU sized packets, because the virtual NIC visible in the guest domain does not provide TSO support. This is because virtual I/O devices in Xen have to be generic across a wide range of physical devices, and so they provide a simple, consistent interface for each device class, which may not reflect the full capabilities of the physical device.

Profiling results show that in addition to performing large TCP packet segmentation in software, the guest domain uses the function `csum_partial_copy_generic` to perform TCP checksum computation in software. This is again because the virtual NIC does not support the checksum offload capabilities of the physical NIC. However, we have determined that disabling TSO support in the Xen-domain0 configuration increases instructions executed by a factor of 2.7. Recall that the ratio of instructions executed between the guest domain in Xen-guest0 and the driver domain in Xen-domain0 is 2.8. Therefore, we conclude that the inability to use TSO in the guest domain is the primary cause for the higher instruction cost for the guest domain in Xen-guest0.

In summary,

1. The *sender* workload in the guest configuration achieves a throughput less than one-fifth of its throughput in Xen-domain0 or Linux.

2. Xen's I/O driver domain model prevents the guest domain from offloading TCP processing functions (e.g., TCP segmentation offload, checksum offload) to the physical interface.

3. Increased working set size and reduced locality in the guest configuration also lead to higher L2 miss rates.

6.3 Web server workload

Our third benchmark for evaluating network performance in Xen is the macro-level web-server benchmark running a simple synthetic workload. The web-server benchmark stresses the overall networking system, including data transfer paths and connection establishment and closing for a large number of connections.

We use the multithreaded *knot* web server from the Capriccio project [22], and compare the performance of the different domain configurations with a baseline Linux configuration. We run the server using the LinuxThreads threading library in all configurations, because the NPTL library currently cannot be used with Xen.

To generate the workload, we run multiple httperf [17] clients on different machines which generate requests for a

single 1 KB file at a specified rate (open loop). We measure the aggregate throughput of the web server for different request rates. For our workload, we deliberately choose a small file size to force the server to become CPU-bound [6], in order to emphasize the performance difference between different configurations.

Figure 7 shows the performance of the web server for the configurations, Linux, Xen-domain0, Xen-guest0 and Xen-guest1. As we can observe, the maximum throughput achieved in the Xen-domain0 and Xen-guest1 configurations is less than 80% of the throughput achieved in Linux. *knot* running on Xen-guest0, achieves a significantly lower peak throughput. Its peak throughput is only around 34% of the peak throughput of Linux.

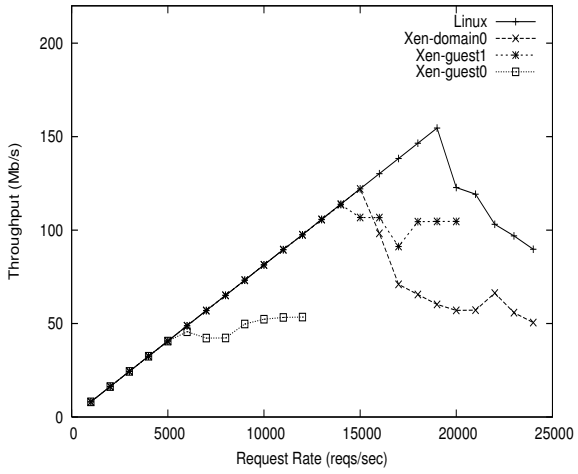

Figure 7: httperf throughput for different configurations

Figure 8 compares the aggregate hardware event counts for all four configurations at a request rate of 5000 reqs/s. At this request rate, all configurations deliver the same throughput.

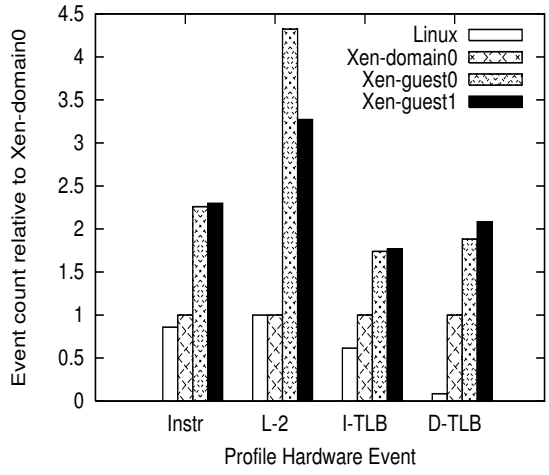

Figure 8: Relative hardware event counts in different configurations for web-server benchmark

Many of the trends seen in figure 8 are similar to the results seen for the *sender* and *receiver* microbenchmarks. In particular, DTLB misses per instruction is much higher in all Xen configurations than in Linux. The TLB miss rate is the primary cause for the lower throughput of Xen-domain0 compared to Linux.

Both the guest configurations suffer from higher instruction costs and L2 miss rates compared to the Xen-domain0 configuration. Overheads identified earlier such as those associated with bridging in the driver domain and page transfers in Xen contribute to the higher instruction count for the guest configurations. However, TSO offload does not contribute to significant overhead in this workload, because each data transfer is of a small 1 KB file, and large data transfers are not involved. The higher computational overhead does not degrade performance significantly in the Xen-guest1 configuration since it utilizes two CPUs. The Xen-guest0 configuration performs the worst, both because of its higher computational overhead on a single CPU, and also because of higher L2 miss rates.

7. RELATED WORK

Several previous papers which describe various VMM implementations include performance results that measure the impact of virtualization overhead on microbenchmark or macrobenchmark performance (e.g., [4, 15, 25, 23, 19]). Xenoprof allows a deeper investigation of such performance results to measure the impact of virtualization on microarchitectural components such as the cache and the CPU instruction pipelines, and the interaction between the application, the VMM, and other domains.

The use of techniques similar to Xen's I/O device driver domains seems to be a growing trend with many cited advantages including portability, reliability, and extensibility [10, 9, 24, 14]. It is important to gain a better understanding of the performance impact of using this new model [6]. Our work aims to contribute to this effort, and it is hoped that the performance issues that have been identified so far through the use of Xenoprof will be helpful in finding opportunities to improve performance with this model.

Performance monitoring techniques in general can be classified into two categories, software level monitoring and hardware level monitoring. Software level monitoring involves collection of data related to software events, such as number of calls to a function, amount of heap memory usage, and detection of memory leaks. Collection of software level data is done through code instrumentation, either at compile time or dynamically at run time. Tools that can use this approach include Caliper [12], PIN [21], and Paradyn [16]. Hardware level monitoring uses programmable performance monitoring counters in the processor or chipset hardware to collect information about hardware events. Several hardware monitoring interfaces [8, 18, 5] and tools [3, 1] have been defined for using hardware performance monitoring on different architectures and platforms.

Conventionally, both software and hardware performance monitoring have been defined in the scope of single OS systems. Xenoprof extends this approach to virtual machine environments which potentially run multiple OS instances simultaneously.

Hardware performance monitoring has been used to tune application performance in Java Virtual Machines, which

present a different but somewhat analogous hierarchy of interacting entities as in the Xen environment [20, 11].

Xen currently supports the notion of software performance counters, which can be used to count software level events occurring inside Xen, such as number of hypercalls, context switches between domains, etc. Xenoprof extends the profiling facilities of Xen in an orthogonal direction, allowing hardware level performance events to be counted across multiple domains.

8. CONCLUSIONS

In this paper, we presented Xenoprof, a system-wide statistical profiling toolkit for the Xen virtual machine environment. Xenoprof allows coordinated profiling of multiple VMs in a virtual machine environment to determine the distribution of hardware events such as clock cycles, cache and TLB misses.

We evaluated network virtualization overheads in the Xen environment using different workloads and under different Xen configurations. Our Xenoprof based profiling toolkit helped us analyze performance problems observed in the different Xen configurations. We identified key areas of network performance overheads in Xen, which should be the focus of further optimization efforts in Xen.

We also demonstrated the use of the profiling toolkit for resolving real performance bugs encountered in Xen. We believe that with the growing popularity of virtual machines for real world applications, profiling tools such as Xenoprof will be valuable in assisting in debugging performance problems in these environments.

9. REFERENCES

[1] Oprofile. `http://oprofile.sourceforge.net`.

[2] TTCP Benchmarking Tool. `http://www.pcusa.com/Utilities/pcattcp.htm`.

[3] The VTune^{TM} Performance Analyzers. `http://www.intel.com/software/products/vtune`.

[4] 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 *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[5] S. Browne et al. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proc. Supercomputing (SC)*, May 2000.

[6] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *USENIX Annual Technical Conference*, Apr 2005.

[7] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. Technical Report US Patent 6397242, vmware, Oct 1998.

[8] S. Eranian. The perfmon2 interface specification. Technical Report HPL-2004-200(R.1), HP Labs, Feb 2005.

[9] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, Cambridge University, Aug 2004.

[10] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Oct 2004.

[11] M. Hauswirth et al. Vertical profiling: Understanding the behavior of object-oriented applications. In *ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct 2004.

[12] Hewlett-Packard. The Caliper performance analyzer. `http://www.hp.com/go/caliper`.

[13] Intel. Vanderpool technology. `http://www.intel.com/technology/computing/vptech/`.

[14] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Operating Systems Design and Implementation (OSDI)*, Dec 2004.

[15] D. Magenheimer and T. Christian. vBlades: Optimized paravirtualization for the Itanium processor family. In *USENIX Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[16] B. Miller. Paradyn parallel performance tools. `http://www.paradyn.org`.

[17] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998.

[18] M. Pettersson. The Perfctr interface. `http://user.it.uu.se/mikpe/linux/perfctr`.

[19] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, Jun 2001.

[20] P. Sweeney et al. Using hardware performance monitors to understand the behavior of Java applications. In *USENIX Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[21] Vijay Janapa Reddi et al. Pin: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Workshop on Computer Architecture Education (WCAE)*, June 2004.

[22] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[23] C. Waldspurger. Memory resource management in VMware ESX server. In *Operating Systems Design and Implementation (OSDI)*, Dec 2002.

[24] A. Whitaker, R. Cox, M. Shaw, and S. Gribble. Constructing services with interposable virtual hardware. In *Networked Systems Design and Implementation (NSDI)*, Mar 2004.

[25] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali isolation kernel. In *Operating Systems Design and Implementation (OSDI)*, Dec 2002.