NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-07-01
March 26, 2007

Blackbox No More: Reconstruction of Internal Virtual Machine State

Benjamin Prosnitz

Abstract

Virtual Machine Monitors (VMM) provide Virtual Machine software which runs on them with a virtual hardware interface. The Virtual Machine's activity is visible to the VMM through this interface but the VMM sees an amalgamation of the activity from all programs running within the Virtual Machine. As a result, it is difficult to partition the activity by program and understand what is really happening inside the Virtual Machine. In this paper, I present a mechanism to associate the activity done within the Virtual Machine with the different programs and threads running within it. I will show that each process and thread has certain characteristics visible from the hardware level to allow for the reconstruction of process and thread tables. I will then show that by using these reconstructed tables, system activity can be associated with the threads and processes which do it.

Keywords: Virtual Machines, monitoring

Blackbox No More: Reconstruction of Internal Virtual Machine State

Benjamin Prosnitz
Department of Electrical Engineering and Computer Science
Northwestern University
b-prosnitz@northwestern.edu

Abstract

Virtual Machine Monitors (VMM) provide Virtual Machine software which runs on them with a virtual hardware interface. The Virtual Machine's activity is visible to the VMM through this interface but the VMM sees an amalgamation of the activity from all programs running within the Virtual Machine. As a result, it is difficult to partition the activity by program and understand what is really happening inside the Virtual Machine.

In this paper, I present a mechanism to associate the activity done within the Virtual Machine with the different programs and threads running within it. I will show that each process and thread has certain characteristics visible from the hardware level to allow for the reconstruction of process and thread tables. I will then show that by using these reconstructed tables, system activity can be associated with the threads and processes which do it.

1. Introduction

Virtual Machine Monitors (VMM) emulate a complete computer hardware environment. Guest operating system software running within the Virtual Machines (VM) interfaces with the virtual hardware environment as if it were a physical one, using low-level hardware protocols. The low-level data which is being communicated though these protocols conveys little immediate information about the meaning of the data within the guest operating system. The goal of this paper is to take a step in the direction of building meaning from these interactions.

Inside of the virtual machine, a modern multiprogrammed operating system switches between many computational tasks usually referred to as processes and threads. The tasks each work with their own resources.

On the hardware level, these tasks are not immediately visible. The hardware sees a blur of the activity caused by every one of the tasks. In this paper, I recreate the set of tasks that are running and division the activity done into the tasks that are performing it.

The implications are large. The traditional design of many applications revolves around the division of tasks. If this state can be derived from the low level hardware operations, these applications may be able to function outside of virtual machines.

Intrusion Detection Systems (IDS) are a particularly important application. The more information that these systems have, the more effectively they are able to detect intrusions. If intrusion detection systems run within the operating system they are trying to protect, they must proactively stop the intrusions before the IDS itself is disabled by them. As a result, the IDS frequently is deployed on the network where it is more resistant to attack but is not able to defend as effectively against intrusions as when running on the user's machine because it does not have as much information available to it.

Garfinkel [12] proposed the idea of running the user's operating system within a virtual machine and running the intrusion detection system outside of it. With this method, the intrusion detection system will likely be less vulnerable than when running within the user's operating system but will have more state accessable to it. In this paper, I describe a method of reconstructing this state from outside the virtual machine which is applicable to this form of intrusion detection.

The remainder of paper is organized as follows. In section 2 I give background on the technologies used in my work. In section 3, I give an overview of my goals, the challenges associated with them and the approaches that I take to solve them. In section 4, I discuss related work. In sections 5-7, I describe and analyze my implementation. In section 8, I present future venues for the work and conclude the paper.

2. Background

In this section, I present background information on the technologies used in my work: Virtual

Machine Monitors, Intel VT-x, the Kernel-Based Virtual Machine, and EM64T / AMD64. I also present an overview of how system calls and interrupts are handled on modern computer architectures. In later sections, I describe how my implementation makes use of these technologies as well as decribe adaptations of my implementation to use different technologies.

2.1. Virtual Machine Monitors

Virtual Machine Monitors are pieces of software which present the software running on them with an interface that resembles physical hardware. Operating System software, which traditionally runs on the physical hardware itself, can then interface with this virtual hardware layer as a "guest OS". One challenge in virtualization is presenting the guest OS with a believable and functional virtual hardware interface.

Virtual Machine Monitors vary in implementation based on the hardware they run on, but many share similar techniques. Many VMMs operate by executing all guest instructions in user mode. When privileged instructions are reached, they trap into the VMM and the VMM emulates the effect of instruction execution. In order for this technique to work all sensitive instructions, as defined by Popek and Goldberg [2], must trap. If this is the case, the architecture is considered virtualizable.

The x86 architecture has historically not been virtualizable [3]. There are a number of sensitive instructions for the architecture that do not trap.

There are other common problems that span architectures which make efficient processor virtualization difficult. Since privileged instructions trap, manipulation of guest operating system state can be very inefficient. This problem is called Ring Compression. Another issue is called Address-Space Compression, where the VMM takes up some of the guest OS's linear address space.

2.2. Intel Virtualization Technology - VT-x

Intel developed its VT-x virtualization technology [1] to address the problems associated with running virtual machines on the x86 architecture, ease development of VMMs and increase performance.

VT-x adds two new execution modes to the processor: VMX root mode and VMX non-root mode. The VMM runs in VMX root mode. It populates the Virtual Machine Control Structure (VMCS), which contains guest state when the processor is in root mode and host state when the processor is in non-root mode. In order to run a virtual machine and load guest state, the VMM performs a VM entry. Likewise, to return to the VMM a #VMEXIT occurs which reloads the host state.

The separation of host and guest state solves a number of problems that arise with x86 virtualization. Ring Compression is no longer an issue, as the guest can run in ring 0. When a VM entry occurs, the linear-address space changes so Addres-Space Compression does not occur.

With VT-x the guest OS can usually run without intervention from the VMM. There are some cases, however, where the VMM needs to intervene. These include handling page faults, I/O, exceptions and some interrupts. When these events occur, the processor performs a VM exit and makes the VMM handle it. The VMM has some control over which events cause VM exits and can specify these options in the VMCS.

2.3. Kernel-Based Virtual Machine

The Kernel-based Virtual Machine (KVM) [6,7] is a combination of a linux kernel-mode virtualization driver and a modified QEMU userspace program that interfaces with the driver. When a virtual machine is to be started, the modified QEMU portion allocates the guest OS memory image, uses ioctl to request that the driver create the architecture-specific portion of the state, and finally runs the virtual machine through the driver. The virtual machine is scheduled as the modified QEMU process. Because of this, the virtual machine can be stopped through normal unix commands like *kill*.

When a VM exit occurs in a KVM Virtual Machine, it is dispatched to kernel-mode handlers which either try to perform the needed action directly in the kernel or return to userspace handlers. In KVM, I/O and CPUID events are handled in user-space while other events are managed in kernel-mode.

KVM currently supports one virtual processor per guest VM. KVM initially required hardware virtualization support like VT-x and AMD's Secure Virtual Machine (SVM) technology [8], but it now supports a paravirtualized architecture [10] as well.

2.4. System Call and Interrupt Handlers

System calls and interrupts are handled in a similar manner on most modern computer architectures. In order to prepare the system to handle the interrupt or system call, the operating system specifies the address of the handler in an

area of memory that is easily locatable by the processor. When a system call instruction or interrupt occurs, the processor switches to kernel-mode and begins executing instructions at the specified handler's address. On the x86 architecture, there is a different handler for each interrupt but there is typically only one handler that is used for system calls.

2.5. EM64T / AMD64 Architecture

Intel's EM64T [19,20,22,23] technology is a set of extensions to the x86 architecture for 64-bit integer computation and extended addressing. EM64T is based off of AMD's AMD64 technology [8,9] so the description is applicable to both.

In order to simplify the development of AMD64-dependent operating systems, AMD decided to remove architectural support for a number of x86 processor technologies..

EM64T/AMD64 introduces a long-mode to the processor which supports the use of 48-bit virtual addressing as well as new 64-bit general purpose registers. In long-mode, segmentation is disabled and a flat 48-bit virtual address space is mapped *only* through paging mechanisms to a 52-bit physical address space.

In addition, system calls made with SYSENTER/SYSEXIT instructions are no longer allowed and only SYSCALL/SYSRET instructions can be used for fast system calls. The processor setup for both SYSENTER and SYSCALL instructions is similar. The Operating System set an entry point Model-Specific Register (MSR) on the processor corresponding with the system call instruction to point to the system call handler location.

3. Overview

My goal is to associate activity visible to the VMM with the processes and threads which cause or respond to it. The hardware activity that I focus on is I/O activity. In addition, I associate system calls with the processes and threads which complete them.

In order to track activity of individual processes and threads, tables of the active processes and threads need to be constructed. I will show in section 5 that there are hardware signatures that can be used to identify the individual threads and processes. Once the threads and processes can be identified, reconstructing these tables is straightforward.

Using the signature of the thread and process that are active, I/O and system call activity can be associated with them so long as the activity itself is visible to the VMM. KVM already handles I/O activity for the guest OS so no changes are needed to intercept it. Intercepting system calls is more of a challenge.

When system calls occur in the guest VM, the are routed through handlers in the guest kernel. The location of the handlers is stored as part of the processor state. As a result, either the location or the handler itself can be modified to jump into the VMM, from which I can associate the call with the running thread and process.

4. Related Work

Internal virtual machine state has been collected before through other mechanisms.

Garfinkel [15] used Mission Critical's crash tool [16] to examine the memory state of the virtual machine and parse kernel structures from it. My work differs as it does not rely on knowledge of a particular operating system type and version, with the exception of using system call calling conventions. My version also does not attempt to interpret guest OS state from its memory image.

SimOS [17,18] takes advantage of the wide range of control offered by simulation platforms to construct state. It active process information is collected by triggering the execution of a collection script whenever a hardware context switch occurs or when code gets executed in the operating system scheduler. The scripts are dependent on the operating system. The methods SimOS share similarities with mine because both trigger the execution of data collection code when low-level events occur.

5. Implementation Details

In this section, I describe the methods that I use to construct process and thread tables and associate I/O and system calls with the processes and threads that are involved with them. In section 6, I will critique and analyse these methods and offer alternatives. Source code can be made available upon request.

5.1. Process Identification

I take advantage of the fact that each process runs in its own virtual memory space. On the EM64T architecture, a virtual memory space is defined solely by page tables. When processes switch, the virtual memory space and therefore the

active page table needs to switch as well. This switch is usually done by changing a pointer to the head of the active page table which is stored in a processor register.

On x86 processors the register which points to the head of the page table is the cr3 register. Since the location pointed to by the cr3 is unique to each process, the value stored in the cr3 can be used as a process identifier.

As will be described further in section 6, a VM EXIT can be forced to occur whenever a switch to kernel mode occurs within the virtual machine. When the VM EXIT occurs, the last value of the cr3 – representing the process that was executing – is recorded and previous actions are associated with that process.

5.2. Thread Identification

For a given process, any number of threads could be executing. Like processes, each active thread has a unique characteristic which can be used as a thread identifier. In this case the identifier is the location of the bottom of the current stack in the process virtual memory space.

Two assumptions are made in the current implementation. When a new thread execution is detected, it is assumed that the stack pointer points to a location on the bottom page of the stack. In addition, it is assumed that threads do not use more than two pages of memory for their stack. A more sophisticated implementation which does not have these issues is planned for a future revision.

5.3. Interception of Exceptions and Interrupts

One point of transfer to the kernel, where process and thread switches may occur, is exception and interrupt handling. VM EXITs can occur for exceptions, depending on the configuration of VT-x in the VMCS. By default in KVM only page faults, non-maskable interrupts (NMI) and external interrupts cause VM EXITS. In this implementation, all possible exceptions and interrupts cause VMEXITS except for the #NM "Device Not Available" exception which is used to enable the floating point coprocessor.

5.4. Interception of System Calls

Fast system call instructions (SYSCALL/SYSRET) do not naturally perform VMEXITs when they occur with VT-x. In order to force VMEXITs when these instructions occur, a VMCALL instruction is injected as the first instruction in the system call entry point. The operating system is required to do most post-SYSCALL setup operations manunally. Since the VMCALL is the first instruction executed, most guest state before the call is maintained and can be interpreted as process and thread identifiers, system call numbers and arguments. The format of system call numbers and arguments is operating system dependent. In order to demonstrate that the system call functionality in my implementation works, I read the system call number using the 64-bit Linux convention [21] of storing it in the RAX register. The system call is then associated with the active process and thread.

5.5. I/O Monitoring

I/O operations are already managed through KVM in userspace. My logging code runs in kernel space so I use ioctl to send data to the driver. The driver logs the last I/O completed for each thread.

5.6. Collection and Logging of Data

Data is stored in two levels of tables. One of the tables is the process table. For each process entry there is a table of the process's threads. Each thread and process has a field storing the number of the last system call which was made and the last byte of I/O which occurred. More detailed system call and I/O information can be logged easily if necessary.

5.7. Client Application

I constructed a client program which queries information from the KVM driver. The number of processes and the list of processes can be outputted through a command. There are also commands querying information about the processes and their threads. For processes, there are commands to do the following: print the number of threads used in the process, list the threads, output the number of the last system call made and output the last byte of I/O completed. For threads, there are commands to print the last byte of I/O completed and the last system call made.

6. Analysis of Implementation

In the following subsections, I discuss, critique and analyze the effectiveness of my implementation. The performance analysis is in section 7.

6.1. Reconstruction of Process and Thread Tables

The methods used to detect both processes and threads rely on memory addressing. In both situations, the data could move in memory, resulting in two entries for the same object. I have not yet resolved this situation. A thread or process could also stop executing and a different thread or process could use the same resources, conflicting with the old identifier. Because of this, it is useful to detect the death of threads and processes.

One method that I tried for detecting the death of threads and processes was *Declared Death*, the use of system call information. Of course, this method requires knowledge of the guest OS's system calls and calling conventions. On Linux guests, I monitored exit() and exit_group() calls. exit() and exit_group() calls are frequently used to tell the operating system to terminate a process. They can be embeded in code and are sometimes automatically appended to the end of the main() function by compilers.

The exit() and exit_group() call monitoring reduced the number of processes and threads active reported by my software, but the number of processes and threads active was still about ten times higher than that reported by the guest Operating System. This could mean one of two things. Either the use of the cr3 as a process identifier is an inaccurate one, or that processes die through another mechanism.

I also experimented with the use of an expiration time for each thread and process. I call this method *Expiration*. With *Expiration*, the last time of thread activity is logged and a thread is declared to be dead when it has not been active within the past few seconds. A problem with *Expiration* is that threads may block on a synchronization object for long periods of time and appear dead. With a process death time of 5 seconds, the process count reported by my software was very close (usually no more than 2 away with a total count of about 37) to that reported within the virtual machine but also very volatile and sometimes was higher than the number of processes reported within the virtual machine.

One source of additional process listings could be kernel drivers. Kernel drivers have their own virtual memory space for mapping high memory physical addresses which may show up as processes through my software. I changed the code to check if the current processor ring is kernel mode when a trap occurs and is to be logged. If it is in kernel mode, the logging does not occur. This change did not solve the problem entirely.

Another possible source are intermediate processes, used to complete work for another a process. On a Linux guest, it is difficult to test whether this is this cause of the extra process listings accurately.

The reported process count by my software did increase as more applications were being used and decrease when the system was idle.

One idea I originally considered for tracking the executing thread was using the location of the program counter when a thread enters and leaves the system as a temporary thread identifier. For example, when a thread enters the kernel, the last user-mode program counter location would be stored as the thread identifier. When the operating system re-enters user-mode, the starting user-mode program counter location would be used to look up the appropriate thread information.

This has a number of problems. First, multiple threads within a process may be interrupted at the same instruction. This is particularly a problem if multiple threads are started at approximately the same time in a process with the same entry point. There is a chance that they will be interrupted by a timer at the same point. Second, a VMEXIT needs to be performed when the thread resumes execution in user mode in order to record the program counter. This can be done in a number of ways. All of the ways I evaluated have accuracy problems with ensuring that a VMEXIT only occurs when the kernel returns to user mode. Also, the additional VMEXIT for this user mode entry decreases the speed of execution.

6.2. Interception of System Calls

I originally planned to implement system call monitoring by injecting a new system call handler into the guest OS's virtual memory. I will refer to this method as *Additional Handler Injection*. When a system call occurs, the processor would begin executing instructions in the new handler. This new handler would perform a VMCALL and then jump into the guest OS's default handler. *Additional Handler Injection* could support many guest OS's with minimal interference. The problem is the complication of finding a location in memory to put the handler into, setting up page tables and ensuring that guest memory management doesn't conflict with the mapping.

In order to simplify my implementation, I avoided this method. My original workaround was admittedly not very good. With *Double Instruction*

Swap, two VMCALLs were performed and the guest OS's default handler was modified frequently. At first, a VMCALL instruction was inserted into the beginning of guest OS's default handler, overwritting 3 bytes previously in the handler The replaced bytes were saved. After the VMCALL occured, the VMCALL instruction was replaced by the original 3 bytes and another VMCALL instruction was placed below them. After this second VMCALL occured, a VMCALL instruction was inserted in the first location and the second VMCALL was replaced by the bytes that were originally there.

This implementation makes two assumptions. First, it assumes that the initial instructions after a system call will be setup instructions which do not branch. Second, it assumes that an instruction doesn't span across both the third and fourth byte of the handler. In addition, this implementation performs two VMEXITs and with them, additional overhead.

Because of these problems, I changed my implementation to take advantage of another observation of guest OS code – that there are frequently gaps between function bodies in memory. Before the system call handler of my Fedora Core 6 guest, there is a gap large enough to fit the vmcall instruction. With *Fill the Gap*, a VMCALL instruction is inserted in the gap and the SYSCALL MSR is set accordingly.

When a system call is performed, the processor begins executing instructions 3 bytes above the guest OS's default handler location, where a VMCALL instruction is located. When the VMCALL returns, the remainder of the handler's instructions are executed normally. Performance improved slightly with *Fill the Gap* over *Double Instruction Swap*, as will be shown in section 7.

For an implementation that supports many different guest OS's, *Additional Handler Injection* would be most appropriate. This would likely have similar performance characteristics as *Fill the Gap*, which performs just one VMCALL. There would be additional overhead, though, as a result of managing page tables to support the handler.

Changes made to the kernel handler are detectable by a scan through the kernel memory which an antivirus program might perform. Injecting a custom handler in a separate page is less detectable, because MSR reads can be emulated.

An alternative method is to set the permissions on the guest OS's normal handler's page to trap on accesses. This method would incur a lot of overhead, because there is likely a lot of kernel code on the page and all of it would need to be emulated.

A clever process might take advantage of the fact that only the fast system call handler has been modified and could make calls through interrupts. User-specified interrupts can't directly be set to perform VMEXITs with VT-x, but the interrupt handler can be modified in the same way as the fast system call handler to perform a VMCALL.

6.3. Interception of Exceptions and Interrupts

In order to monitor a full range of guest OS transitions into kernel mode, I modified KVM so that the VM performs VMEXITS for all the interrupts that VT-x can perform VMEXITS for, except #NM. These interrupts include breakpoints, general protection faults and other fixed hardware interrupts but not user defined interrupts. As will be shown in section 7, performance is harmed when VMEXITS are performed for these additional interrupts.

6.4. I/O Monitoring

The same information used to associate system calls with the process and thread which is running could be used for some I/O. One problem occurs with I/O which is completed by the kernel. A different stack would be used by the kernel.
Depending on the cause of the I/O (external interrupt from device, system call, etc.) it may be possible to associate it with a process or thread. A simple way to do this is to associate the I/O with the thread and process that runs just after it occurs.

This is the approach that is taken in my implementation. When I/O occurs, information about the I/O which occured is saved. When the next interrupt or system call occurs, the thread that was running is determined and the saved I/O information is associated with it.

A problem with this approach is that if it is input I/O, the operating system might not immediately handle it and the thread and process that executes afterward may not be involved with it.

My I/O logging implementation performs extra context switches using ioctl because the logging code is inside of the kernel driver and the I/O is handled in user mode. Because most of the overhead in doing the logging is implementation specific (the ioctl call used), I won't perform a performance evaluation of the I/O logging overhead.

6.5. Adaptation for traditional x86 Processors

My implementation takes advantage of the properties of newer 64-bit x86 processors with 64-bit operating systems for simplicity. Similar techniques can be applied to traditional x86 processors.

One EM64T feature I took advantage of was the imposed requirement on 64-bit operating systems to use a flat-address model, without segmentation having any effect. With a flat-address model, all reasonable methods to switch virtual memory spaces involve simply modifying a pointer to the head of the page table so these changes can be interpreted as process switches.

Even without this model, *some* change to at least one table referencing the part of the physical memory being accessed must occur during a process switch. Because of this, an efficient operating system would swap either the active page table, the active segments or both. The VMM could examine both the page table location and the segment configuration to come up with an identifier for the process. This could be as simple as an array of the relevant segment and page data or a cryptographic hash of it.

Thread identification methods are also affected when a 64-bit operating system isn't used. The stack segment is likely changed instead of the virtual address of the stack. The stack segment information can then be used as a thread identifier instead of the stack location itself.

In 64-bit long mode, software must perform system calls through SYSCALL instructions. 32-bit software typically uses SYSENTER instructions instead but has a number of avenues through which it can perform system calls. In order to support this, similar changes can be made. 32-bit SYSCALL and SYSENTER instructions both set MSRs with the call handler location. The handler can be modified with the same method as for the 64-bit SYSCALL instructions. Call gates are rarely used but if necessary a VMCALL could be inserted at the entry point of every one.

7. Performance Evaluation

In this section I provide quantitative data generated from benchmarks run on my test system. In the first subsection, I analyze the KVM's performance without any modifications. In the two subsequent subsections, the performance changes due to my modifications are analyzed.

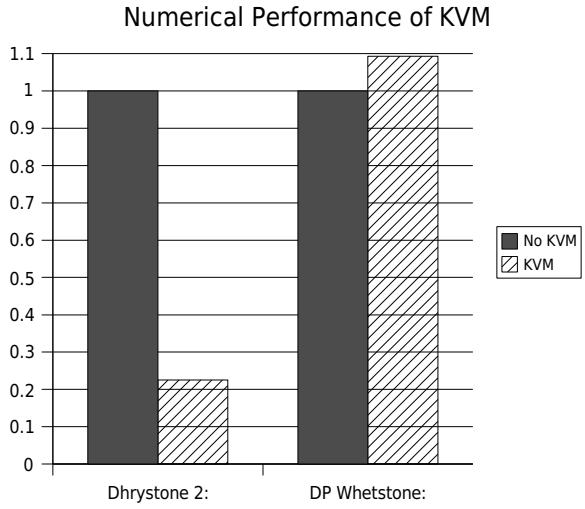
The system configuration is the same across all tests. The system is a 2.00 ghz T7200 Intel Core 2 Duo dual core machine with 2 GB of memory and a Nvidia Quadro NVS 120M graphics chip. The kvm module allocates 1 GB of the memory. The host OS is Debian Etch 4.0 amd64 “Testing” with a modified kvm-12 kernel module running with a Debian 2.6.18-3 kernel. The guest OS is Fedora Core 6 x86_64.

The primary performance benchmark I used was unixbench. Unixbench [16] is a comprehensive benchmark which performs the following tests multiple times each: Dhrystone 2 [17] using Register Variables, Double-Precision Whetstone [18], execl() Throughput, File Copying, Pipe Throughput, Process Creation, Shell Scripts and System Call Overhead. All benchmarked results will be normalized. The File Copy benchmark has units of amount of data per second and the other benchmarks have units of operations per second. For all the benchmarks, higher is better.

7.1. KVM Performance

Before evaluating the impact of my modifications, it is important to first evaluate the performance of KVM itself. In the charts that follow, the baseline “No KVM” represents the results of the test being run on my System without using virtualization. “KVM” represents the results of the test being run inside KVM on the same system.

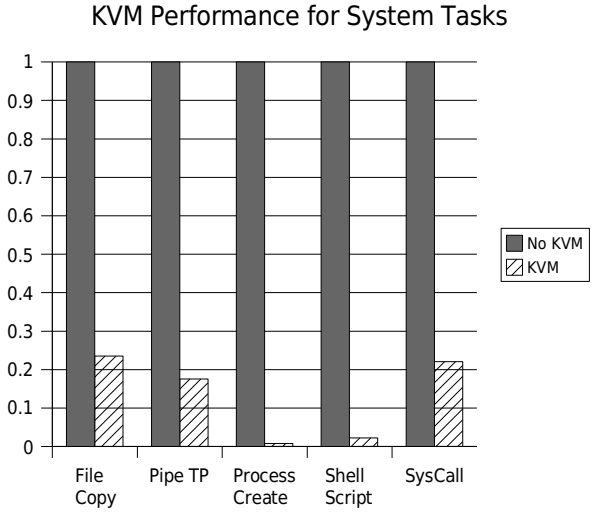
First, I evaluated KVM's numerical computation performance using the Dhrystone integer performance benchmark and the Whetstone floating point performance benchmark. The normalized results are shown below.



KVM had about 23% the integer performance of that without KVM and approximately the same

floating point performance both with and without KVM.

Next, I will evaluate the performance degradation for system tasks due to KVM. The tests performed were the File Copy, Pipe Throughput, Process Creation, Shell Script and System Call benchmarks. The results presented below are normalized.



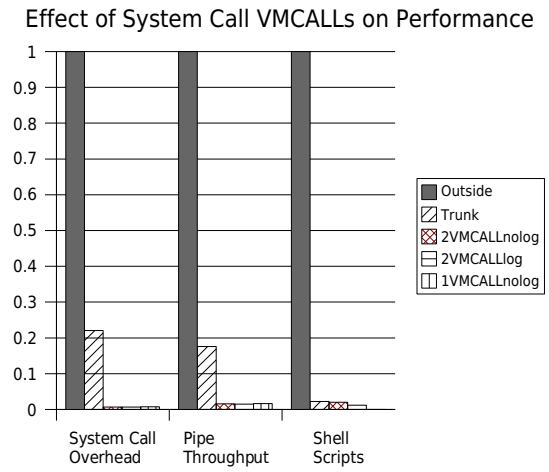
All tests show a large amount of performance degradation. The file copy, pipe throughput and system call tests resulted in 15-25% of the performance seen running on the system without KVM.

KVM performed very poorly with these benchmarks. Previous results [11] showed that it performed similarly but slightly worse than Xen.

7.2. System Call Performance

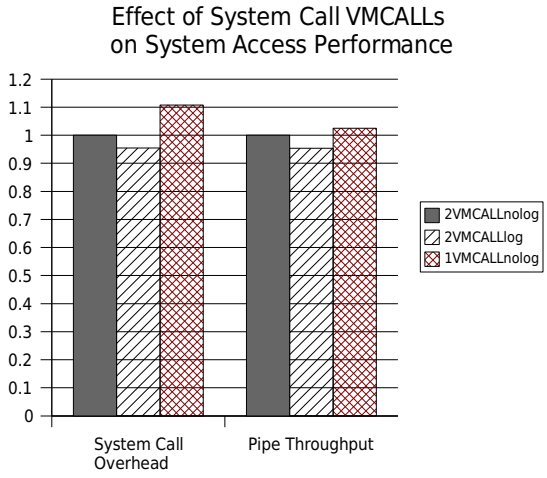
I assessed the performance of system calls on the system, using the standard trunk build of kvm, and three modified versions of kvm. The modified versions are as follows. The first (2VMCALLnolog) is the version that performs two VMCALLs as described in section IV (*Double Instruction Swap*), but does not call my code which logs the process and thread information. The second version (2VMCALLlog) is identical to the first but now logs process and thread information. The third (1VMCALLnolog) is my newer version (*Fill the Gap*), described in section 6 which performs one system call but does not log. All three modified versions have only been modified in the way they handle system calls.

First I will compare all of these systems using the System Call Overhead, Shell Script and Pipe Throughput benchmarks. The following chart uses normalized data from these benchmarks.



There is a large drop in performance in all three tests when using KVM versus the performance seen when running on the host operating system. In addition, there is another large performance decrease when system calls are monitored for both the System Call Overhead and Pipe Throughput benchmarks .

In order to better examine the affect of logging and of the change to a single VMCALL on performance I present Pipe Throughput and System Call Overhead results for the three modified versions of kvm in their own chart. This chart is shown below.



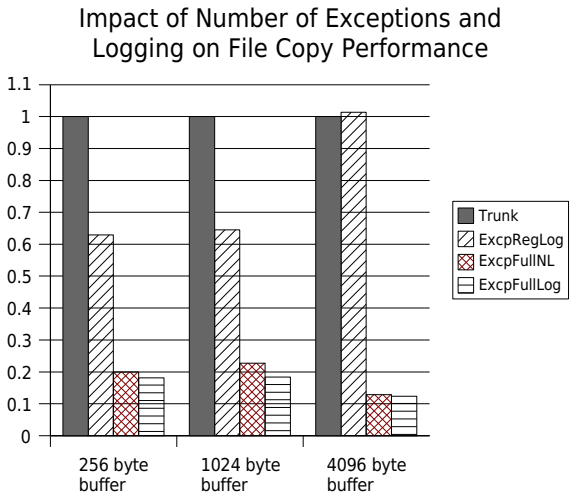
As can be seen in the chart, logging incurs a small 5% overhead. Reducing the number of VMCALLs increased performance by 2-15%.

7.3. Performance of Performing Logging when Exceptions and Interrupts Occur

For these tests, I used four versions of KVM. “Trunk” is the regular kvm-12 with default exceptions enabled and no logging. “ExcpRegLog”

is like the “Trunk” version, but has logging enabled. I also created two versions which use a larger number of exceptions. These versions are identical in every respect, except one performs logging and one doesn't. “ExcpFullNL” is the version which does not log while “ExcpFullLog” does. All of these versions of KVM are only modified in how they handle exception and interrupts..

In the first test, I evaluate file copying performance. I use results from three file copying tests with varying buffer sizes of 256, 1024, and 4096 bytes. The results are provided below, normalized to the “Trunk” version's results.



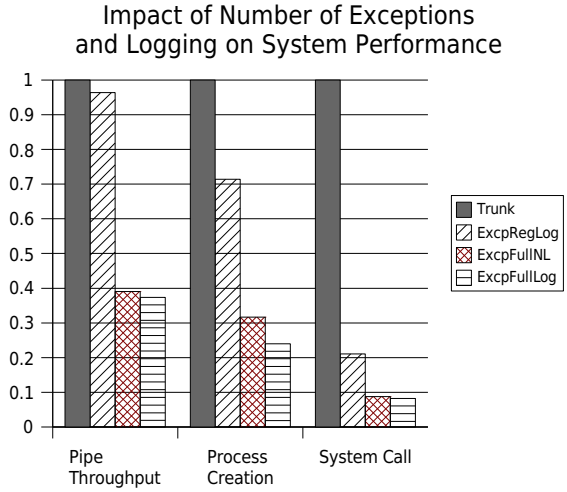
Even when no additional exceptions and interrupts are caught, logging has a non-negligible impact of file copying performance. Performance is then 60-65% of that without it with 256 and 1024 byte buffers. With a 4096 byte buffer, the performance difference is minimal.

Catching a larger number of exceptions reduces performance to 20-25% of what it is otherwise, with 256 and 1024 byte buffers. Enabling logging results in a 10-20% decrease in performance with 256 and 1024 byte buffers.

With a 4096 byte buffer, performance is even lower. Performance is then about 12-13% of that without the additional exceptions.

Next, I evaluated system performance with the same four versions of KVM. I used the Pipe Throughput, Process Creation and System Call benchmarks. The normalized results are on the next page

An increase in the number of exceptions caught reduced the pipe throughput significantly. The impact of enabling logging was less significant. With the additional exceptions, logging decreased the throughput by less than 5%. When using the additional exceptions, performanced drop to 40% of that seen otherwise. Enabling logging again caused a less than 5% drop in performance over the similar version without logging.



Process creation performance was also impacted. Enabling logging with the default set of exceptions decreased performance to about 2/3 of that otherwise seen. When the number of caught exceptions is increased, performance is about 1/3 of that otherwise. When logging is then enabled, ¾ of the process creation performance of ExcpFullNL is experienced.

System call performance is also severely impacted. When logging is enabled with default exceptions, performance is about 20% of that without logging. When the number of caught exceptions is increased, performance is about 8% of that of the trunk, both with and without logging.

8. Conclusion and Future Directions

I presented a system that could reconstruct process and thread information from low level data and associate I/O and system calls with those processes and threads. The process and thread information was reconstructed in a manner independent of the running operating system. I/O and system call information is accessable the same independent way, but can only be interpreted with knowledge of the operating system's conventions.

While I succeeded in building a system to construct this data, the performance impact of its use is significant. Coupled with the overhead already caused by virtualization, the slowdown reduces the usefulness of the system.

Despite this, the ability to divide system activity into the work of individual processes and threads may help make sense of the activity. Once data

streams within the virtual machine can be isolated, the data itself may be interpreted more simply.

Applications which work with process and threads as objects may be able to move outside of a virtual machine and experience the benefits of doing so. They will be able to find and access the process's memory easily.

The performance degradation due to this system is significant but the benefits of its use are as well. As a result, these methods may still find some use.

9. Acknowledgment

I am grateful for Dr. Peter Dinda's advice, guidance and support for this work.

10. References

[1] G. Neiger, et al. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal, Volume 10 Number 3.* (August 2006), 167-178.

[2] G. Popek, R. Goldberg, Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM* (July 1974), 413-421.

[3] J. Robin, C. Irvine. *A*nalysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium* (August 2000).

[4] Avi Kivity, et al. KVM: Kernel-based Virtual Machine for Linux. URL http://kvm.qumranet.com accessed March 2007.

[5] Avi Kivity, et al. KVM: Kernel-based Virtualization Driver. URL http://www.qumranet.com/wp/kvm_wp.pdf accessed March 2007.

[6] F. Bellard. QEMU: Open Source Processor Emulator. URL http://fabrice.bellard.free.fr/qemu/ accessed March 2007.

[7] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 Annual USENIX Technical Conference* (April 2005). pp 41-46.

[8] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Sunnyvale, CA., 2006.

[9] *AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions*. Sunnyvale, CA., 2006.

[10] Barham, et al. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles* (Oct 2003, Boston Landing, NY). pp 164-177.

[11] Michael Larabel. http://www.phoronix.com/scan.php?page=article&item=623&num=3

[12] T. Garfinkel, M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *The Proceedings of the 2003 Network and Distributed System Security Symposium* (Febuary 2003, San Diego, CA.)

[13] Crash Core Analysis Suite. URL http://www.missioncriticallinux.com/projects/crash/ accessed March 2007.

[14] Rosenblum, et al. Using the SimOS Machine Simulator to Study Complex Systems. *ACM Transactions on Modelling and Computer Simulation, Volume 7, Number 1* (January 1997), 78-103.

[15] S. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. Stanford, CA., 1998.

[16] Unixbench. URL http://www.tux.org/pub/tux/niemi/unixbench/ accessed March 2007.

[17] R. Weicker. Dhrystone. 1984. URL http://www.netlib.org/benchmark/dhry-c accessed March 2007.

[18] Whetstone. National Physics Laboratory, 1972. URL http://www.netlib.org/benchmark/whetstone.c accessed March 2007.

[19] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference A-M. Santa Clara, CA.*

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B Instruction Set Reference N-Z. Santa Clara, CA.*

[21] Linux kernel 1.6.18. URL http://www.kernel.org accessed March 2007.

[22] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide. Santa Clara, CA.*

[23] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide. Santa Clara, CA.*