NVIDIA VIRTUALIZATION

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

_____

Timothy Corsetti

_____

Konstantin Naryshkin

_____

Amanda Strnad

Date: March 06, 2009

Approved:

_____

Professor David Finkel, Major Advisor

Abstract

For the NVIDIA Virtualization project, our goal was to create a hypervisor that could run Windows with a minimal loss of performance and no loss of functionality. Our hypervisor is designed to be easily customizable through the console interface. The hypervisor takes advantage of the AMD Virtualization hardware support to intercept I/O instructions. We also examine how to intercept events that AMD-V does not directly support, such as memory accesses.

Acknowledgements

We would like to thank Professor David Finkel and Provost John Orr for the opportunity to participate in the Silicon Valley Project Center.

We would also like to thank the staff at NVIDIA, in particular, Allen Martin, who taught us many things and helped the project go smoothly.

Table of Contents

1. Introduction

DELETED

This report will go over the general background of the possible uses of hypervisors, as well as the development of our hypervisor at NVIDIA. In Chapter 2 Part A, the background of what hypervisors are, their implementations, and the challenges associated with their development will be covered. Chapter 2 Part B will be a discussion of the technologies that were emulated by this project. Once the background and technologies involved have been discussed, Chapter 3 will be the development process. This chapter will cover the four iterations of the development of the hypervisor. Finally, in Chapter 4, we will discuss the results and possible future work for the project.

2. Background

A. *Virtual Machines and Hypervisors*

A virtual machine (abbreviated as VM) is a program that emulates the behavior of a physical computer. A VM is most often used to run one or more operating systems (referred to as guest operating systems) on a single physical machine. The operating system and the physical machine that it is running on are referred to as the host. The software that manages VMs, makes sure that they all have access to physical hardware, and stops them from interfering with one another is called the virtual machine monitor, or simply the VMM (Zeichick, Processor-Based Virtualization, AMD64 Style, Part I, 2006).

i. Types

VMMs can be broadly grouped into three categories according to how they are implemented. The first category is a hypervisor, which is a VMM that runs directly on the hardware. It is a VMM that is not dependent on a host OS to exist but rather interfaces with all the hardware directly. This is the category that our VMM would fall into. Another category is middleware virtualization. This is the most common form of virtualization. It is a VMM that is a separate program from the host OS and that relies on the host OS to do tasks such as hardware initialization. The final category is OS virtualization. This is a VMM that is a component of the host OS. This method is primarily used for the OS to isolate each application within a controlled VM (Zeichick, Processor-Based Virtualization, AMD64 Style, Part I, 2006). Virtual-appliances fall into this category. A virtual-appliance is a VM used to isolate the system from a piece of

hardware that needs extra security and management beyond what the drivers provide. Virtual-appliances show potential for enterprise client systems as well as workstation and home computers. (Abramson, et al., 2006) Virtual-appliances serve a similar purpose to our hypervisor.

Middleware VMMs can be further separated into three classes: OS-hosted, stand-alone, and hybrid. OS-hosted VMMs are dependent on the host OS to communicate with hardware. The VMM uses the host OS drivers, components, and subroutines to simplify development. It also allows the VMM to support any hardware that the OS supports with a standard API. The weakness of OS-hosted VMMs is that they also inherit any security holes, instability, and bugs that the host OS has. Stand-alone VMMs are the other extreme. They do not depend on the host OS to handle any of the hardware interfaces. Because of this, they are very stable, since any bugs that they contain can only come from code developed by the VMM maker. An added advantage of them is that they tend to be faster with I/O than OS-hosted VMMs because instructions can go directly to hardware rather than going onto a stack in the host OS. The cost of making a VMM stand-alone is that it can no longer take advantage of the host for drivers, so every device that the VMM should be able to use has to have drivers built into the VMM. Doing this also increases the complexity of the VMM since it has to handle technologies, such as ACPI, which are needed to gain access to the hardware. The hybrid VMM tries to take the best features of both the other two approaches. It works by having the host run a small and deprivileged service OS. The service OS handles I/O devices while the VM controls the CPU and memory. A hybrid VMM is more stable and secure than an OS-hosted one because the service OS only interacts with the VMM. It is also simpler than a stand-alone VMM since

complicated I/O interaction and drivers are offloaded onto the service OS. The hybrid OS is also the slowest performing option. For security purposes, the service OS is isolated from the VMM and run with reduced privileges. The constant switches between two programs that make up the hybrid VMM carry a significant cost on CPU time due to the difference in privilege levels. (Abramson, et al., 2006)

ii. Major Challenges for VMMs

Regardless of the type, virtual machines are generally used as wrappers to run multiple stand-alone operating systems on a single computer. Because of this, the VMM has to deal with two unique problems: running multiple operating systems on one set of hardware and running commercial operating systems in a VM.

The VMM is the program that is in charge of maintaining the VMs and making sure that all of them are able to work properly. The VMM shoulders the responsibility to make sure that each guest is allowed to work and that a single guest malfunctioning or running malicious code will not hamper the execution of the other guests. (Abramson, et al., 2006) In order to maintain isolation, the VMM often has to split up a single physical resource. This partitioning is done through methods like giving each VM only a limited region of memory to use or creating multiple virtual disks on a single physical hard drive. Even if the device is partitioned, it does not guarantee isolation. A rogue guest could still try to access a region it does not own. In situations where security is important, the VMM will encrypt each of the parts. Doing this forces the VMM to encrypt and decrypt the data as the guest tries to read and write it, preventing a rogue guest from accessing memory it should not. Some devices cannot simply be divided. In such a case, the VMs need to share the device. In order to do this as transparently as possible, the VMM maintains

control over the device. When a guest wants to use the device, the instruction is intercepted by the VMM. The VMM then sends the instruction to the physical device and returns the result to the appropriate VM. If the device is one that processes I/O calls out of order (such as a network card), it falls upon the VMM to match the output to the VM that issued the instruction. (Zeichick, Processor-Based Virtualization, AMD64 Style, Part I, 2006)

Since the guest operating systems that customers want to run tend to be commercial, the VMM is usually working with a guest that was not written with ease of emulation in mind. The guest OS will generally have a number of assumptions that originate from the expectation that the OS has full control of the machine it is running on. (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006) One such assumption is that the guest OS is running in ring 0 of the processor, the most permissive privilege level of the CPU. (Advanced Micro Devices, 2007) The VM cannot allow the guest OS to actually run at this privilege level since doing so will allow the guest to have too much control of the CPU. The guest OS has to be moved to a less privileged level, in a process called ring deprivileging. This puts the guest OS at the same level of permissions as the applications running on it. (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006) The process is significantly easier when using a virtualization technology supported in the CPU directly, such as the AMD-V extension that we used, since the CPU supports the VMM running at a special level that can intercept and maintain control over a guest that is in ring 0 as well. (Zeichick, Processor-Based Virtualization, AMD64 Style, Part II, 2006) Another assumption that the guest makes is that it will have its full address space to use. To ease with the transition between the host and guest, the VM has to reserve a small amount of the address space. If the VM

is able to put itself entirely into the guest's address space, it gains a large amount in terms of performance. The guest code cannot be allowed to access or modify this memory. The process of creating these hidden areas in the addresses is called address-space compression. (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006) In our VM, this problem is solved by configuring Windows not to use part of its memory and by setting up our VM to halt if the guest ever tries to enter memory that we reserved. A third assumption that the guest will make is that it can always disable interrupts in the CPU. The VMM cannot allow this for several reasons. The first is that the guest OS would have to be in ring 0 to do this and, as previously discussed, the guest should not have such a high level of permissions. Another reason is that some VMMs use the interrupts as a way to gain control of the CPU unconditionally, so allowing them to be disabled would stop the VMM from working correctly. Regardless of the reasons behind it, emulating the disable of interrupts tends to be expensive to simulate. Since some operating systems do it quite often, it can cause a noticeable loss of performance. (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006) The final assumption that the guest OS makes is that the CPU will work quickly and efficiently. Most guests do not expect to have more than a few CPU cycles pass between the executions of instructions. Since VMMs will often have to halt the guest OS to perform some action behind the scenes, this delay can cause a noticeable lag for the user. This problem is especially noticeable when the guest tries to use a privileged resource, such as a shared I/O device, and the VMM is forced to interrupt the execution of code constantly. (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006) The problem can be alleviated by the use of paradrivers. These are special drivers created to optimize instructions to hardware and pass them directly to the VMM without needing to interrupt

the guest each time. (Zeichick, Processor-Based Virtualization, AMD64 Style, Part I, 2006)

Regardless of the cause, the VMM will often want to change the way that the guest OS works to make it easier to run in a VM. Several methods exist to improve the efficiency of the VM. One method is paravirtualization. Paravirtualization works by modifying the guest binary to make it easier to work with. It can be done either by changing the source code of the OS and recompiling it or by modifying the binary executables that make up the OS. This method causes no loss of speed at run time when compared to running the code without changes. The downside to this method is that it limits the user's choice of OS to ones that the VMM is able to modify. This is the method used by the Denali and Xen VMMs (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006). Another way to improve efficiency is binary translation. It works by analyzing all the instructions that the guest is performing, and if it looks like the guest would try to perform any problematic ones, the VMM modifies the instructions to remove the problem. This method does not restrict the possible guest operating systems used, but comes at a considerable runtime cost, as every single instruction that is run must be analyzed at runtime. This is the approach employed by both VMware and Microsoft for their VMMs (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006). A third method, and the method that we used, is CPU level virtualization. It relies on the VMM being run on a CPU that supports a virtualization extension like AMD-V or Intel VMX. CPU virtualization extensions allow the user to define a number of situations for the CPU to monitor for and to exit out to the VMM if one occurs. Since it is done directly in the CPU, the monitoring is a great deal more efficient then the monitoring used by binary

translation but it retains the ability to work with any guest. (Neiger, Santoni, Leung, Rodgers, & Uhlig, 2006) The biggest downside is that the CPU can only monitor for a subset of possible situations. Since the technology is very new, not many VMMs take advantage of it, but both Xen (Zeichick, Processor-Based Virtualization, AMD64 Style, Part II, 2006) and KVM (Chmielewski, 2007) will use it if it is available.

iii. I/O Virtualization

No matter how well a VM performs or how secure it is, it would be worthless without the ability for the guest OS to be able to use I/O devices. I/O virtualization needs to handle four major tasks: the guest OS has to be able to discover the presence of the device, control the device, transfer data to and from the device, and get I/O interrupts. There are a few techniques to handle I/O virtualization; they are emulation, paravirtualization, and direct assignment. (Abramson, et al., 2006)

Emulation works by emulating an I/O device within the VM software. The emulated device needs to show up in such a way that the guest OS will discover it. The VM then needs to capture any reads and writes to the device and emulate the appropriate action by the device. The VM would also need to emulate an interrupt controller. It would be used to tell the guest OS about any interrupts that occur on the device. If a VM uses emulation for I/O, then the VM becomes independent of the hardware it is running on. The VM can be moved to any set of hardware, and the guest OS will not know since the emulated devices stay the same. Emulation supports sharing a single device between multiple VMs as more than one emulated device can be implemented using a single physical device and the sharing is handled in the emulation code. Because emulation can adapt the device to an interface that the guest OS can use, it is also useful in situations when the guest OS

cannot use a physical device on the system. Emulation can accomplish all this regardless of the guest OS. The major problem with emulation is that it is very resource intensive. Every single call to the I/O device needs to be handled by the driver in the guest OS, intercepted by the VM, processed by the emulation code, passed off to a device driver if one exists, and then finally passed to the physical hardware. The response needs to traverse the layers once again to get back to the guest OS. Emulation is also very difficult to code when the emulated device is supposed to be a specific make and model. When an exact device is emulated all bugs, quirks, and undocumented features of the device must also be emulated. Not doing so could break the drivers for the device. Ultimately, emulation is a very useful way to do I/O virtualization since it can be used in almost any situation, but is rarely more efficient than the other two techniques. (Abramson, et al., 2006)

Paravirtualization offers increases in the performance at the cost of some of the flexibility. Paravirtualization works by modifying the guest OS to use a device driver that calls directly into the VM. This approach can work at a higher level of abstraction than a traditional driver and results in less data transfer between the guest and VM. Paravirtualization supports the same level of VM portability as emulation as long as the new environment supports the same API between the VM and guest as the old environment. Just like emulation, paravirtualization can be used for a device that is shared between multiple VMs. The drawback of using paravirtualization is that it limits the operating systems that the guest can run to those that the VMM can edit. To support interrupts properly, the VM also needs to rewrite some of the interrupt handling routines

of the OS to accept interrupts coming from the VM directly. The Xen VMM uses paravirtualization (Abramson, et al., 2006).

The I/O virtualization technique that offers the best performance is direct assignment. It works by simply passing all communications from the guest OS directly to a dedicated physical device. It then allows the guest OS drivers to directly control the device. It increases performance by eliminating the intermediate step, talking to the VM, from I/O interactions. The VM does not even need to have a driver for the hardware device, since all the communication is handled by the guest OS. The problem is that the physical device needs to be completely dedicated to a single VM. The device cannot be shared and if the VM is moved to system that does not have a device of the same model, the guest OS would need to be reconfigured to reflect that. If the device is reliant on DMA to communicate with the guest OS, then a pass-through driver needs to exist. The driver copies the device's memory region into and out of the guest operating system's memory space. Even when using direct assignment, interrupts need to be manually forwarded to the guest OS by the host. The direct assignment approach is popular in the server market where the customer can afford to have a device dedicated to each VM. The technique is generally used when emulation is too slow and when the OS cannot be edited to support paravirtualization. (Abramson, et al., 2006)

In most cases, I/O virtualization is done with multiple techniques. A VM can use a combination of them, with a different one assigned to each device depending on needs. (Abramson, et al., 2006) In our case, we used both direct assignment and emulation. Direct assignment was used for all the physical hardware, since we only have one VM running at a time and do not have to worry about how portable the VM is.

DELETED

iv. CPU based virtualization

Both AMD and Intel have added extensions to their built-in commands to allow virtualization on a CPU level. Both systems focus on creating a hypervisor that is capable of running multiple VMs without them getting in each other's way. This is accomplished by maintaining tight control over which machine is allowed to use each page of memory. Secure virtual machines, VMs capable of coexisting with no chance of a single defective VM compromising others, require this control. The system also maintains tight control over what the VM is allowed to do. This control was a key feature that allowed our project to be feasible. (Advanced Micro Devices, 2007), (Intel, 2008)

a. AMD Virtualization

The AMD extension supports secure virtual machines through a technology called AMD Virtualization (referred to as AMD-V). AMD-V maintains a single 'host state' in a memory location pointed to by the VM_HSAVE_PA model specific processor register. This memory is used to store the state of the VMM while a VM has control of the CPU. The state of each of the virtual machines that the hypervisor runs is maintained in a structure called a virtual machine control block (referred to as a VMCB). The VMCB contains information about the CPU state of the virtual machine as well as some information for the AMD-V VMM about managing the virtual machine. The save state within the VMCB contains the values of some registers such as the segment registers and

the system-control registers. Other registers, such as the general-purpose rBX and rCX registers, are not saved and must be set manually when the VM is started. They must be saved in another location whenever the VMM runs any code that could change them. (Advanced Micro Devices, 2007)

In addition to the save state, the VMCB contains a number of conditions that will cause the guest VM to stop and transfer control of the CPU back to the VMM, also called an intercept. The intercepts are intended to be used as a way for the VMM to handle the guest operating system trying to perform a command that the VM is not allowed to do. One of the types of intercepts is the instruction intercept. An instruction intercept watches the commands that the CPU performs and only causes an intercept if the guest asks the CPU to perform any of a given set of instructions. We use this intercept to stop the guest from running certain instructions, such as the AMD-V ones, that would give away the fact that the guest is running within a VM. Another type of intercept that is useful for us is called IOIO intercepts. These are intercepts that trigger when the guest tries to use the IN, OUT, INS, or OUTS instructions to write to or read from an I/O device. This intercept allows us to emulate an I/O device by injecting valid return values when the VM tries to use the I/O addresses our device is expected to reply on. The third type of intercept that will be of use is the interrupt intercept. The intercept triggers when the guest code tries to trigger an interrupt and will be useful for us to catch so that we can stop the interrupt from occurring if it dealt with hardware that does not exist. (Advanced Micro Devices, 2007)

Along with interrupts and save states, the AMD-V also supports a few other useful features. One such feature is that using the VMCB, the VMM can insert an event, such as

an exception or interrupt, into the VM. Another feature arises out of the need by the VMM to handle multiple VMs each having its own set of memory. To aid this, the AMD-V system supports two technologies: shadow page tables and nested paging. Both techniques are based around giving the VM a fake page table. The page table is the table mapping addresses that programs see to real addresses in memory. With shadow page tables, the processor has multiple page tables in memory, one for the host and each VM, and the host manages all of them. In nested paging, the page table of each VM is not mapped to real memory but is mapped to memory addresses through the page table of the host. AMD-V facilitates both of these techniques by automatically swapping which page table the CPU uses if a VM is entered or exited. Either one of these technologies could be used by us as a method of hiding our code from the guest operating system. (Advanced Micro Devices, 2007)

b. Intel Virtual-Machine Extensions

The Intel virtualization extensions (referred to as VMX) are very similar to AMD-V in terms of their abilities. Both systems were developed to solve the same problem: the running of a secure virtual machine. In the Intel extensions, the main structure for maintaining the state of the guest is called virtual-machine control data structure (abbreviated to VMCS). Like AMD-V, VMX support a variety of intercepts to monitor the behavior of the VM. Intel also supports the insertion of events into a VM. Although Intel does not have built in support for nested paging, it does have a feature that can be used for similar effect. Called extended page tables, this is comparable to AMD-V's shadow page tables (Intel, 2008).

Though the two are very similar, VMX differ from AMD-V in several major respects. Unlike AMD-V, which supports starting a VM at any time as long as the extensions are enabled, Intel handles the transition into a VM by running several commands. First, the VMCS must be loaded into a special register in the processing core that will be running it, and then the host has to enter a state that allows VMX instructions. Only at this point is the host allowed to run a command to enter the VM. This long process is because Intel only permits the host and guest VM to exist in a sub-set of the allowable CPU states. VMX forbids certain combinations of control registers; these combinations tend to be combinations that are not allowed when enabling it. VMX go as far as stopping the host from changing the registers and running commands that will affect them these registers as long as VMX are on. Similar restrictions are imposed on the VM. It is not allowed to enter real mode or disable protected mode paging. This forces Intel, in the specifications, to devote a chapter to a convoluted work around to enable the VM act like a real computer. The VMX are also significantly stricter about certain intercepts, imposing them onto the VM without giving the host the option of disabling them. These restrictions allow VMX to be significantly more stable. In some cases, a VM in AMD-V can cause the CPU to crash and reboot the computer. In VMX, such states are not allowed to occur. In VMX, the host is forced to act before such a state can be reached. (Intel, 2008)

B. *Emulated Technologies*

DELETED

3. Process

We started by breaking down our goal state into two week milestones. Every two weeks, there would be some new functionality.

A. *Tech Demo*

For the first two weeks, our primary goal was to get a serial debugger working. This was especially important since most runtime errors cause the computer to reboot instantly, making it impossible to read any information printed to the screen. By sending debug messages via a serial port, there was a record of each debug message sent. Since the messages remain visible, this greatly enhances our ability to do any debugging.

Along with the ability to print out fixed strings, we also needed to be able to print out variables. Normally, this can be done using the standard printf function in C. However, since the function is in GNU code, we decided to implement our own version. Had we used any GNU code, we would have been forced to include our source code with any commercial release of our software. It is less complete, but allows for the basic functionality of printf. It can be used to output string literals, character arrays, and numbers.

Our other goal was to create a working, though minimalistic, hypervisor. It would be able to start from GRUB, then boot into Windows. This turned out to be more complicated than expected, and became part of our Alpha Version.

B. *Alpha Version*

For the halfway mark, our goal was to have a functional hypervisor that could initialize the virtualization options that the hardware provides and then boot into Windows from there.

We were able to accomplish this by dividing the tasks. We were able to work on booting Windows separately from initializing virtualization, then combining the code.

In order to run code within an AMD-V virtual machine, we would need to properly set up a VMCB, a region of memory that needed to store the initial state of the processor as well as information for the CPU about how to run the program. One of the largest difficulties with the task lay in the fact that the AMD-V technology was only developed recently. Because of this, sample code for a working VMCB did not exist. Also, since the technology was so new, the documentation for it had several omissions. The lack of complete documentation meant that development involved a large amount of experimentation. Because we had access to a hardware debugger, we were able to find some of the initial CPU state by looking at the state of the CPU as the system booted.

We were able to boot Windows by analyzing the boot process used by the GRUB chain loader. The first stage of GRUB is located in the MBR, and its only purpose is to load the second stage of GRUB. This second stage then determines what to boot from. Since the code in the MBR is limited to only 446 bytes (DEW Associates Corporation, 2002), we used the same division of code GRUB did. When the computer is booted from our code, the code is loaded into location 0x7C00 in memory. This is where the BIOS expects any boot code to be. Our code copied itself into memory at 0x8200 so that it would not be overwritten when loading the Windows boot loader at 0x7C00. Once

copied, the code analyzed the partition table in the MBR to determine the active partition. If Windows was installed on the computer, it would be located on the active partition. Once the partition to boot from had been chosen, the boot sector from that partition was copied to 0x7C00. Once we combined the code, the virtualization initialization code was run at this time. Finally, the program jumped to 0x7C00 and relinquished control to Windows.

The virtualization code allowed for the possibility of setting up custom interrupts, and trapping on specific instruction or memory accesses. However, in this iteration of the code, our program was overwritten once Windows began to boot. Thus, it was unable to react to any trapped instructions or intercepts.

C. *Beta Version*

For the second to last milestone, we needed to have a demonstrable piece of software. It did not need to be fully functional, but it did need to be able to show that we could meet our goal. We chose to focus on making sure that the hypervisor was in its near-final state and functioning completely.

The beta version implemented the working 3-stage booter. The 3-stage booter broke the boot process into three parts, Stage 1, the bootstrap, Stage 2, the low memory code, and Stage 3, the high memory code. The bootstrap program is what loads all the code in memory. This code must be loaded into the first 512 bytes of the boot partition as per the x86 specification. In this case, the x86 architecture requires that the bootstrap be exactly 512 bytes long, and terminated with the hex bytes 0xaa55, otherwise it will not be recognized as a valid bootstrap. Unlike Stage 2, the code in Stage 1 will not be needed

after Stage 1 has finished execution. Stage 1 can be overwritten without trouble. Once Stage 1 has finished loading the code execution is passed off to Stage 2.

Stage 2 contains code that must be run in real mode. Therefore, the Stage 2 code must be contained in the first megabyte of memory. This is because real mode uses a 20 bit segmented addressing space, limiting addressable memory to 1 MB. Since this code is loaded into low memory, it will be overwritten by the guest OS when the guest is loaded. Some of the code in Stage 2 will be used by Stage 3 until the guest OS is loaded and running. Once the guest OS is running, Stage 3 will not need to return to real mode. At this point, Stage 2 may be overwritten by the guest with no consequences. When Stage 2 takes over execution, it relocates the Stage 3 code to a section of memory that will not be used by the guest OS. Stage 2 then resolves any unresolved symbolic links in the Stage 3 code before it passes execution to Stage 3. This was accomplished by making use of the headers provided in the .ELF file specification. The Stage 2 code searches the Stage 3 symbol table and relocates the symbols to point to the correct function. Once all the symbols in the table are resolved, execution is passed to the Stage 3 code.

The Stage 3 code contains the code that needs to remain resident in memory when the guest OS is running, as well as most of the code needed to load the guest OS. When the guest OS is being loaded, Stage 3 needs access to code contained in Stage 2 that allows reading from the hard drive. When the guest OS is loaded and running, it will likely have overwritten Stage 2 in memory, so Stage 3 will no longer have access to the functions available in Stage 2. For this iteration of the project, Windows was modified to not use memory beyond a certain range, and the Stage 3 code is hard coded to be loaded beyond this range.

Once we had implemented the 3-stage booter, we were able to test the hypervisor's ability to watch IO ports and return control to the hypervisor when these were used. In order to test our code, we set up the hypervisor, allowed the computer to boot into Windows Vista, and then used a program called NVPCI32 to test reading and writing to specific ports. When the hypervisor was supposed to break in, it returned to the debug monitor that we had created. This allowed us to examine the state of the CPU registers while the hypervisor was running. We could then return to Windows from the debug monitor.

In doing this, we discovered a few issues with our code. One issue was that AMD-V only saved a subset of the CPU registers. When we exited the VM, we could unintentionally overwrite some of the registers in the CPU and have those changes get passed on the VM. To fix this, we included code to save the rBX, rCX, rDX, rBP, rSI, and rDI registers when the VM exits and reload them when we were ready to start the VM again. Another issue was that, initially, we had difficulty triggering the intercepts. When we used the NVPCI32 program to read or write from the port that we had programmed to have an intercept, nothing triggered. We found that by opening PuTTY for serial communication, we were able to repeat the previous steps, but now the port access was intercepted. We suspect that the unusual behavior is due to Windows overwriting the serial port configuration. As well, our code was only controlling instructions run on the boot core of a dual core system. This meant that when we tried to read or write from the port we were trying to intercept, it had a probability of about 50% to trigger the interrupt. To solve this, we adjusted the boot options for Windows to use only one core. After this, we were able to intercept the port every time. Another way

around this would be to run a hypervisor on all the cores. The hypervisors would need to work cooperatively to ensure that if a hypervisor intercepted an action on one core, no other cores could continue running until the intercept had been taken care of.

D. *Final Version*

DELETED

In the process of developing our final version, we were tasked with developing a Windows-based installer for the hypervisor. Previously, the hypervisor was installed by copying the file to a Linux system, copying the file onto the USB drive using the Linux dd command, and then moving the USB drive to the computer running Windows. This required either multiple computers or a system running both Windows and Linux. To allow for future development and deployment of the hypervisor, the installation needed to be as simple as possible for the customer. This meant that there needed to be a way to install the hypervisor from Windows.

The installer is a simple GUI that allows the user to select the hypervisor executable to install and the device to install it to. The installer then writes the hypervisor to the specified device. When the hypervisor is written, a message lets the user know that the installation has finished. In the event of an error, the installer halts and displays an error message containing the error code number. The main limitation of the installer is that it must be run by a user with administrative privileges on the system, otherwise Windows will not allow the reads and writes needed for the installation to be successful.

The installer works as long as the USB drive it is installing onto has no data on it. We were able to write all zeros to the drive in Linux, then use the installer to install the hypervisor. This worked, and they hypervisor ran correctly. When we tried formatting the

drive in Windows, however, Windows does not write all zeros to the disk. We discovered that while the installer writes all nonzero bytes correctly, it does not work properly for zero bytes. Any byte that should be a zero remains the same as it was prior to the install. This has the unfortunate effect of a hypervisor installed on an unclean USB drive tending to have strange errors or unexplained behavior.

4. Results and Conclusions

DELETED

In accomplishing some of our goals, we had to impose some limitations on the final product. Originally, we aimed for our hypervisor to be able to run either AMD or Intel chipsets, but later decided to remove Intel support. This was done for several reasons. The first was that there was no standard for virtualization extensions. AMD-V and Intel VMX were developed independently and work in drastically differing ways. Although it is possible to separate out the AMD specific code, doing so would have forced us to write a second almost complete hypervisor to provide support for Intel as well as AMD. We chose AMD over Intel because Intel only released the virtual machine extension in their latest generation. This made the Intel hardware more difficult to get, and we were not able to get access to an Intel CPU and motherboard until about half way through the project. Our decision was reinforced later when we learned that we would not be able to acquire an Intel motherboard with a debug output. We further narrowed our list of supported CPUs down to AMD processors that support nested paging because being able to use it significantly simplifies implementing memory controls. This limitation will get less significant with time as more and more AMD CPUs with nested paging support are made. Another limitation we had was that we needed to shut down all but one core of the CPU through Windows. This was necessary since our hypervisor had to be manually initialized on every CPU core that it would control and since we had no control of which core Windows would use, we had to limit it to only have one choice. A final limitation was that we had to configure Windows to limit its memory usage instead of tricking Windows into thinking that it had less. One necessary limitation is that the guest OS will

not be allowed to run a VM, since one is already running. This is a limitation of using AMD-V, but should not be a problem for most users.

DELETED

To continue our work and remove the limitations, one of the first steps would be to change the hypervisor to load at a dynamic location determined at runtime. The amount of memory in the system can be determined by looking at the memory map returned by interrupt 15h with AX=E820. This would allow the hypervisor to run on computers with more or less memory available. Along with choosing the hypervisor's location dynamically, it will be necessary to trick Windows into believing that the memory at the hypervisor's location is not available. To do this, interrupt 15h needs to be hooked so that we can change what memory is returned as available. It will also be necessary to change the results of the Extensible Firmware Interface (EFI) query for the memory map if the system is EFI enabled (Hewlett-Packard Corporation; Intel Corporation; Microsoft Corporation; Phoenix Technologies Ltd.; Toshiba Corporation, 2006)

The next step would be to run the hypervisor on all cores if the system has more than one. The hypervisor would also need to be adjusted for any race conditions that could occur if more than one core is intercepted at the same time.

Once these two steps have been completed, the hypervisor will be portable to any computer with virtualization and nested paging support without any additional setup. The hypervisor is not specific to any piece of hardware or software on our test machine, so these are the only things limiting the hypervisor's portability. Since the hypervisor does not need to be installed directly onto the computer to function, the USB drive can be

moved to any recent AMD computer with no changes. It will work without any modification or recompiling.

DELETED

Hypervisor Framework API

A. *Technical Details*

i. Multi-stage booting

When the 8086 processor was developed, it defined a 512-byte boot sector at a fixed location in memory that was responsible for loading the code to be run. At the time it was developed, this size provided more than enough space to load programs into memory, or boot an operating system. The code contained in the boot sector is processed in what is now referred to as real mode, an operational mode using 16-bit registers, and has a 20-bit physical address space, allowing for addressing of up to 1024 kilobytes. When the 80286 was developed, it added protected mode, an operating mode that allowed the use of a 24-bit address space, allowing for 16MB of memory. To maintain compatibility with programs developed for the 8086, the 80286 began operation in real mode. All following x86 processors are started in this way (X86: Encyclopedia - X86).

As operating systems became more complex, the limitations of the real mode address space became apparent. The code needed to boot an operating system soon reached the limit of the available memory in real mode. In order to boot a more complex system, the boot process needed to be split up into multiple stages. Each stage is responsible for loading the next stage and then handing execution over to the stage it loaded. Most current operating systems are booted with a 2-stage boot loader. The first stage resides in the 512-byte boot sector defined by the 8086. The first stage loads the second stage below

the first megabyte of memory and then hands execution over to the second stage. The second stage will then switch to protected mode and load the operating system.

While a two-stage boot is most common, there are times when three or more stages are needed to boot an operating system. One such example is the GRUB boot loader and the way it handles booting non multiboot-compliant operating systems. The GRUB boot loader is designed to boot a multiboot-compliant operating system. Most distributions of Linux are multiboot-compliant and can be loaded directly from GRUB. Windows, however, is not multiboot-compliant and is reliant on its own boot loader. In this case, GRUB will load the Windows boot loader and hand execution over to it so it can boot Windows. This process is called chain loading (GNU GRUB Manual 0.97).

For our booter we needed three distinct stages. The first stage resides in the first 512 bytes of the MBR, and is responsible for loading and executing Stage 2. Stage 2 contains code that must be run in real mode, and is responsible for loading and executing Stage 3. Finally, the Stage 3 code is where our monitor program resides, and it is responsible for initializing the virtual machine and any virtualized devices before loading the guest operating system. The code in Stage 3 must not be overwritten by the guest operating system. The Stage 3 code is what handles exits from the virtual machine and controls the execution of the guest.

This separation of the stages was necessary for booting the guest and loading the code that will handle our virtualized device. To prevent the code in Stage 3 from being overwritten by the guest operating system, it needs to be loaded at an address that the guest would not be allowed to use. The problem with this is that this code must be loaded into a high area of memory, memory beyond the highest memory region accessible to the

guest. This means that Stage 3 cannot be loaded directly from Stage 1. Stage 3 also needs to use function held in Stage 2 as these function can only be called from real mode. When Stage 3 hands execution over to the guest, the functions contained in Stage 2 are no longer needed and can be safely overwritten.

B. ***Usage***

DELETED

Glossary

3-stage booter - A bootstrap program that is broken up into three stages, where each stage performs a specific task before handing control off to the next stage. The first stage is responsible for loading the other stages into memory.

Advanced Configuration and Power Interface (ACPI) – An interface for configuration and power management for all devices in the computer

Advanced Host Controller Interface (AHCI) – An interface for communication with hard drives

AMD Virtualization (AMD-V) - An extension to the command set in processors made by AMD created to run a virtual machine at the processor level.

Bootstrap – A program that is responsible for loading code to be executed. Comes from the old expression "to pull oneself up by one's bootstraps" meaning that one can do something without outside help.

Guest- The code running within the VM. In most cases, this will be a commercial operating system.

Host- The physical hardware and operating system (if any) upon which the VMM is running.

Hypervisor - A term that can be used for any VMM, it is sometimes defined to be only VMMs that run directly on the host hardware without a host OS.

Multiboot-compliant – An operating system that follows the Multiboot Specification.

Multiboot specification – An open standard that describes a method of booting different kernels with one compliant boot loader.

Virtual machine (VM) – The software that insulates an operating system or low-level program from having access to real hardware. It allows multiple operating systems to be run on a single computer. It can also be used to run an operating system to run as though it has access to hardware that differs from the hardware available to it.

Virtual machine control block (VMCB) - A region of memory used by AMD-V to store the state of the guest program when the guest does not have control of the processor. It is also used to define how the guest code will be treated.

Virtual machine monitor (VMM) - A program that monitors one or more virtual machines, allows them to have access to hardware, and makes sure that they are not allowed to interfere with one another.

References

Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., et al. (2006, 8 10). Intel® Virtualization Technology for Directed I/O. *Intel® Technology Journal , 10* (3).

Advanced Micro Devices. (2007, September). *AMD64 Architecture Programmer's Manual Volume 2: System Programming.* Retrieved from AMD Developer Central: Developer Guides & Manuals: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

Boyd, J. (2008, June). *AHCI Specification.* Retrieved January 20, 2009, from Intel: http://download.intel.com/technology/serialata/pdf/rev1_3.pdf

Chmielewski, T. (2007, 4 12). paravirtual drivers? *gmane.comp.emulators.kvm.devel* .

DEW Associates Corporation. (2002). *The Master Boot Record (MBR) and What it Does*. Retrieved January 6, 2009, from http://www.dewassoc.com/kbase/hard_drives/master_boot_record.htm

GNU GRUB Manual 0.97. (n.d.). Retrieved Januay 9, 2009, from http://www.gnu.org/software/grub/manual/grub.html#fnd-10

Grimsrud, K., & Smith, H. (2003). *Serial ATA Storage Architecture and Applications.* Hillsboro, OR: Intel Press.

Hewlett-Packard Corporation; Intel Corporation; Microsoft Corporation; Phoenix Technologies Ltd.; Toshiba Corporation. (2006, October 10). *Advanced Configuration and Power Interface Specification.* Retrieved February 2, 2009, from http://www.acpi.info/spec.htm

Intel. (2008, November). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2.* Retrieved from Intel® 64 and IA-32 Architectures Software Developer's Manuals: http://download.intel.com/design/processor/manuals/253669.pdf

Marshall, A. (2006, June 5). *ACPI In Windows Vista.* Retrieved March 2, 2009, from http://download.microsoft.com/download/5/b/9/5b97017b-e28a-4bae-ba48-174cf47d23cd/CPA002_WH06.ppt

Neiger, G., Santoni, A., Leung, F., Rodgers, D., & Uhlig, R. (2006, 8 10). Intel® Virtualization Technology: Hardware support for efficient processor virtualization. *Intel® Technology Journal , 10* (13).

PCI-SIG. (2004, February 3). *PCI Local Bus Specification Revision 3.0.* Retrieved February 18, 2009, from http://www.pcisig.com/specifications/conventional/

Serial ATA International Organization. (2008, August 18). *NEW SATA SPEC WILL DOUBLE DATA TRANSFER SPEEDS TO 6 Gb/s.* Retrieved February 5, 2009, from SATA IO: Enabling the Future: http://www.sata-io.org/documents/SATA_6Gb_Phy_PR_Finalv2.pdf

X86: Encyclopedia - X86. (n.d.). Retrieved March 2, 2009, from http://www.experiencefestival.com/a/X86/id/1901454

Zeichick, A. (2006, 6 30). *Processor-Based Virtualization, AMD64 Style, Part I.* Retrieved 2 19, 2009, from AMD Developer Central Articles & Whitepapers: http://developer.amd.com/documentation/articles/pages/630200614.aspx

Zeichick, A. (2006, 6 30). *Processor-Based Virtualization, AMD64 Style, Part II.* Retrieved 2 19, 2009, from AMD Developer Central Articles & Whitepapers: http://developer.amd.com/documentation/articles/pages/630200615.aspx