Towards Virtual Passthrough I/O on Commodity Devices

Lei Xia   Jack Lange   Peter Dinda
{lxia,jarusl,pdinda}@northwestern.edu
Department of Electrical Engineering and Computer Science
Northwestern University

Abstract

A commodity I/O device has no support for virtualization. A VMM can assign such a device to a single guest with direct, fast, but insecure access by the guest's native device driver. Alternatively, the VMM can build virtual devices on top of the physical device, allowing it to be multiplexed across VMs, but with lower performance. We propose a technique that provides an intermediate option. In virtual passthrough I/O (VPIO), the guest interacts directly with the physical device most of the time, achieving high performance, as in passthrough I/O. Additionally, the guest/device interactions drive a model that in turn identifies (1) when the physical device can be handed off to another VM, and (2) if the guest programs the device to behave illegitimately. In this paper, we describe the VPIO model, and present preliminary results in using it to support a commodity network card within the Palacios VMM we are building. We believe that an appropriate model for an I/O device could be produced by the hardware vendor as part of the design, implementation, and testing process.

1 Introduction

I/O device virtualization plays a key role in the performance of virtualized systems. This paper describes an approach to provide unmodified guests secure yet high performance I/O using standard commodity devices. Our goal is to safely multiplex a physical commodity I/O device among multiple guests that interact with it using native device-specific drivers, with near-native performance.

Much effort has been, and is being put into achieving high performance, yet secure I/O for virtual machines. For example, device emulation (virtual devices) [15] implements virtualized hardware devices completely in software within the VMM. Multiple virtual devices can then be multiplexed on top of a single physical device. No guest software changes are required, but there is a significant performance overhead. Special guest drivers that talk more efficiently to the VMM can ameliorate some of this overhead. Xen I/O [3] extends this concept by requiring guest changes and having the special driver talk to a special VM that has direct hardware access. Drivers can also be placed into individualized driver VMs for better protection [10]. These techniques can often lead to significantly better I/O performance. However, direct assignment I/O, in which a device is directly controlled by the guest's native driver with no VMM intervention at all, still has the potential for the highest performance. Unfortunately, it fails to guarantee the reliability and security of the whole system, especially the VMM. Passthrough I/O [11, 13, 14] exploits specialized hardware, which we call self-virtualized devices, that allows direct guest access under parameters determined by the VMM, thus providing both high performance and security. However, this technique requires hardware support that is not present in commodity I/O devices, and makes other virtualization features such as migration more difficult.

We propose a novel I/O virtualization technique, *virtual passthrough I/O* (VPIO). VPIO allows the guest's native driver to have direct access to a commodity device (one that does not have self-virtualization support) most of the time. The VMM can assure, however, that the guest does not maliciously or inadvertently program the device to affect the VMM or the other guests. Furthermore, the VMM can hand-off the physical device from one guest to another. The VPIO concept is based on two claims:

- It is possible to build an inexpensive software model of a device.[1]

---

[1]Potentially, such a model could be provided by the device vendor.

- That model can be inexpensively driven by guest/device interactions.[2]

If these claims hold, then VPIO is possible. We also assume that the device can be context-switched.[3]

The essential idea in VPIO is that the VMM maintains a formal model of the I/O device that is driven by guest/device interactions. The model can be far simpler than a driver or virtual device implementation, and must only be sufficiently detailed so that when faced with an interaction[4], the model can determine:

- Whether the device is serially reusable after the interaction.

- Whether a DMA is about to start, and which host-physical addresses will be involved.

With such a model, the VMM is able to determine whether a device interaction should be allowed to continue down to the physical device, and at what points a device can be context-switched to a different guest. Thus the VMM can multiplex a single commodity physical device across multiple guests, each of which uses a native driver.

Of course, if every guest/device interaction involves an exit into the VMM, the performance will be terrible. The practicality of VPIO hinges on the extent to which exits can be avoided through modeling and systems techniques, and/or the extend to which the overhead of an exit can be reduced.

In this paper, we describe the VPIO idea in more detail, and we show preliminary results in using VPIO for a commodity network card. Our results support the two claims given above. We demonstrate that we can model the network card, drive the model with guest/device interactions, and determine when the card can be handed off. The model itself is quite inexpensive. However, it remains to be seen whether VM exits can be sufficiently minimized. We conclude by outlining a technique we are evaluating for doing so.

2   Palacios VMM

This work takes place in the context of the V3VEE project (v3vee.org), which is constructing a virtual machine monitor framework for modern architectures. The Palacios VMM is the first virtual machine monitor under development within the project. It will be publicly available by the time this paper reaches print. Palacios is an operating system independent virtual machine monitor targetting either IA32 or X86-64 architectures (hosts and guests) and makes extensive, and non-optional use of the AMD SVM [2] extensions (partial support for Intel VT [8] is also implemented). It runs directly on the hardware and provides a non-paravirtualized interface to the guest. Palacios uses shadow paging in the virtual MMU model (nested paging is optional). An extensive infrastructure for hooking of guest physical memory addresses (with byte address granularity), guest I/O ports, and interrupts facilitates experimentation, such as that described in this paper. At the present time, Palacios is capable of booting an unmodified Linux distribution from either a physical hardware CD ROM drive, or a virtual CD ramdisk image.

The core Palacios VMM comprises $\sim$ 20 KLOC of C and assembly written from scratch. We have also wrote an additional $\sim$ 10 KLOC of C and assembly to implement necessary basic virtual devices. Compiling Palacios generates a static library that can be linked to either 32 or 64 bit executables and operating systems. Currently Palacios has been fully incorporated into U. Maryland's 32 bit GeekOS [7] and Sandia National Lab's 64 bit Kitten operating system. Debugging support includes serial port logging and control, and the ability to do source-level debugging with gdb when Palacios is run under QEMU [4]. A modified version of the BOCHS [9] BIOS and VGABIOS is used to bootstrap the guest OS. Although we generally can avoid using a full instruction decoder, we can optionally link in the XED component of Intel's Pin [12, 5], which provides full x86 instruction decode/encode functionality.

3   VPIO

We now describe VPIO in more detail, in the context of Palacios, as shown in Figure 1. The main component of VPIO is the Device Modeling Monitor (DMM), which is deployed within the VMM. It intercepts device requests from the guest device driver, receives interrupts from physical devices and delivers them to guests, drives a internal device state model for each guest, and can determine whether the device can be handed off to another guest, and what host physical addresses a DMA operation will involve.

3.1   Device requests and interrupts

The guest's device driver talks to physical devices by device requests made by executing IN and OUT instructions. We do not yet support memory-mapped de-

---

It is essentially a much simplified behavioral model that could easily be produced as side effect of the device design or documentation processes, or after the fact.

[2]This implies that most device programming interactions (such as IN/OUT instructions) do not engage the VMM.

[3]This means either that we can copy the device state (e.g., registers) to/from memory, or that the device is deterministic and so we can restore device state by playing back a trace of interactions from a reset.

[4]Think of an OUT instruction on a port associated with the device.
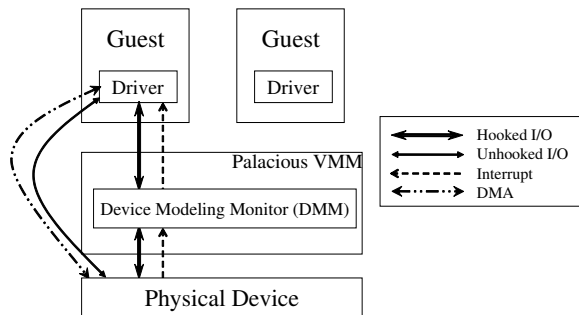
Figure 1: VPIO system.

vices in VPIO, but since Palacios does support memory read/write interception on a byte address granularity, there is no reason why we could not extend VPIO to memory-mapped devices. These requests either go to physical devices directly without VMM intervention, or are intercepted by the VMM for further processing. The choice is dependent on the DMM.

The DMM maintains an internal state-machine model for each guest, which keeps track of the current status (e.g, reusability, DMA operation started, etc) of the physical device as seen from the guest. The model state is updated by guest device requests and physical device interrupts.

The DMM maintains a *hooked I/O* list, a list of I/O ports for which guest reads, writes, or both must be intercepted by the DMM. These reads/writes are needed to update the model. The *unhooked I/O* list are those ports which the model does not require; reads and writes to those ports are not intercepted by the DMM (or VMM). The hooked I/O list is dynamically updated based on changes in the model. In Palacios, I/O port interception is done using SVM or VT's hardware support. An I/O causes a VM exit, which is decoded to vector to the DMM. If allowed, the DMM can decode and handle the guest instruction.

Reducing the size of the hooked I/O list, and the overall number of I/O port interceptions is critical for performance. The hooked I/O list can be reduced by careful modeling of the device. In a later section, we also discuss the possibility of using code injection from the VMM into the guest to push modeling functionality into the guest context, further reducing the number of device requests that cause exits.

All of the physical device interrupts are intercepted and delivered to the DMM by the VMM. The DMM updates the device model and then decides whether to deliver the interrupt to a guest, and to which guest.

3.2 DMA

DMA is essential for high performance I/O. In VPIO, we allow the guest to directly initiate DMA operations to guest physical memory. Notice that before DMA starts, the guest device driver must set it up, using device requests (I/O port reads/writes, currently) that convey the starting memory address, length of the data, etc. By hooking the relevant I/O ports, we acquire these parameters and maintain them as part of the device model. For some devices, the device model may also be able to simply read these parameters directly from the physical device. The device model alerts the DMM when a DMA is about to be started, and what the source/target physical addresses (which are guest-physical addresses) are. This allows the DMM to (1) change the addresses to appropriate host-physical addresses, and (2) validate the addresses against the guest's memory map.[5]

3.3 Device multiplexing

VPIO multiplexes a physical device among multiple guests by essentially context switching the device from guest to guest. The device model determines when a device is in a reusable state, and can be switched. If a guest attempts to perform an operation on a device it does not currently hold, it is blocked until the device becomes available.

The DMM keeps a device context for each guest that includes the current state of the device model, values of all relative physical device registers, and other device-specific flags related to that guest. When the DMM hands off a physical device to another guest, it performs the device context switch. The context switch saves all values of physical device's registers and other flags to the current guest's device context, and then restores these physical registers with values from the device context of the guest that is the next owner of the device.

3.4 Performance

In Figure 2, we consider the performance of different elements of the DMM in Palacios. Here, we report CPU cycle timing on the QEMU x86_64 SVM-equipped emulated processor environment, using an the default NE2000 emulated network card, as well as for a hardware environment, an HP Proliant ML115 with an AMD Opteron 1210 processor. Unhooked I/Os, of course, operate at the speed of the hardware. Hooked I/Os are dominated by the cost of a VM exit and its handling. The

---

[5]A special case exists for a VMM running a single guest, as appears likely be common for high-end computing environments such as the forthcoming Petascale machines: The VMM can be loaded high in physical memory, the guest can be loaded at the start of physical memory, and DMA address translation can be ignored.
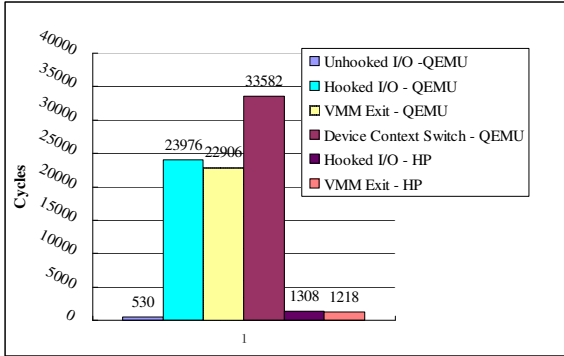
Figure 2: Costs of VPIO operations in cycles.

device context switch depends, of course, on the device, but at about 34K cycles to switch an NE2000, the card could be comfortably switched many times per second on a modern processor.

3.5   Device model

Unlike a behavioral model, or hardware model intended for verification purposes, the aim of the device model in the DMM is only to determine (1) whether the device is reusable, (2) whether a DMA is about to be initiated, and to where, and (3) what device requests (e.g., I/O ports) that the model needs to see to update itself.

The device model is conceptually a state machine with additional scratchpad information (e.g., DMA addresses). The edges are annotated with the device requests (e.g., I/O port reads/writes, interrupts) that trigger them, as well as with *checking functions*. A checking function is called before a state transition occurs, and must approve the state transition. If state transition is denied, the device request fails, and no state transition occurs. Optionally, a notification of failure can be delivered to the guest. The checking functions reflect VMM policy. As side effects, they also can change the hooked I/O list.

The number of states is as small as possible to still answer the questions given above. One or more states must be marked as being reusable.

3.6   Dealing with failure

A natural question that arises is what the DMM should do if the device model shows that the guest is about to put the device into an improper state. For example, suppose the guest attempts to initiate a DMA into memory the guest does not own by using a guest physical address for which there is no legitimate allocated memory. The DMM cannot translate the guest physical address and cannot allow the DMA to be initiated.

If the DMA is to read memory, the operation could be completed, but using zero-filled pages allocated by the DMM. If the DMA is to write memory, the operation could be silently ignored. After all, a DMA to physical memory addresses where memory does not exist would amount to a discard of the data. However, although the DMA is not completed, the guest now expects the device to be in some state valid with respect to the DMA it thinks it has initiated.

A simple approach to both DMA reads and writes to invalide guest physical addresses is simply to inject a machine check exception, or otherwise halt the guest. While probably the best solution, this does make the physical device exposed via the VMM act slightly differently than they would were the VMM not there.

3.7   Dealing with device handoff on interrupt

When a device interrupt occurs, we would ideally vector the interrupt to the appropriate guest. However, for many devices, the appropriate guest is not known at interrupt time. For example, an incoming packet on a network card may not have its destination MAC address known until after it has been DMAed to memory. If we simply let the current guest DMA it, we will need, minimally, to be prepared to copy or page-remap to move the received data to the appropriate guest (and this assumes we can also make the current guest ignore it).

At the present time, we have not yet found a general purporse solution for this problem. A general purpose solution would either allow for efficiently handing off a device to the appopriate guest on interrupt, or for efficiently moving data received in the wrong guest to the appropriate guest.

4   Example device model

We have developed a device model for NE2000-compatible network cards (specifically the Realtek RTL 8029A chipset) to test the claim that efficient VPIO device models can be built. The NE2000 is a simple network card, but not too simple. It supports DMA for sends and receives, including ring buffering. The NE2000 model is not yet incorporated into Palacios, but we have been able to evaluate its stand-alone performance and overheads.

4.1   Model

The NE2000 model is an augmented finite state machine as described in Section 3.5, and is illustrated in Figure 3.
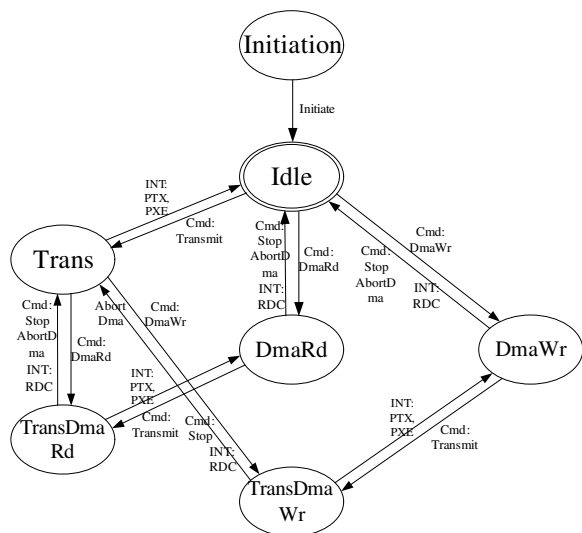
Figure 3: Device model for NE2000 NIC.

Scenario	Total I/O	I/O hooked	Ratio (%)
Linux: ssh	1865055	257324	13.80
Linux: small dl	2602002	69700	2.68
Linux: large dl	294508810	9429917	3.20
Windows: ssh	3769071	286671	7.61
Windows: small dl	1081738	39089	3.61
Windows: large dl	132898230	988535	0.74

Figure 4: Hooked I/O port reads/writes for various benchmarks.

In the figure, the arrows between states represent state transitions, and their annotations are the events that drive these transitions. "Cmd: xxx" means a write of a register on the card by the guest, and "INT: xxx" means an interrupt from the device. The actual port numbers and masks are not shown in the figure.

Checking functions are associated with some edges. For example, whenever entering the DmaRD (DMA read is running) or DmaWr (DMA write is running) states, the checking functions validate the destination/source physical memory address and the transfer length. In the case of the NE2000, the model can directly read this information from card registers, avoiding the need to hook the relevant ports to capture writes of those registers. If a DMA were to violate VMM policy, the DMA-initiating I/O port write can either be ignored, or, in the case of the NE2000, a "remote DMA failed" interrupt can be delivered to the guest. More generally, a machine check exception could be injected.

In the NE2000 model, the only reusable state is the "Idle" state. When the model is in this state, the physical network card is idle. The DMM could thus switch the network card to other guest.

The model's implementation consists of approximately 900 lines of C code.

4.2 Performance

We carried out a preliminary experiment to test the performance and overheads of our NE2000 model. As noted before, the model is not yet incorporated into Palacios. The overheads of hooked and unhooked I/O, sans model, in Palacios are given in Section 3.4. Here, we integrated the NE2000 model into QEMU's NE2000 emulator in order to drive it with I/Os from different guest OSes.

Figure 4 illustrates the number of device requests (I/O port reads/writes) issued in three scenarios run on Linux and Windows: an interactive ssh session, downloading several small files, and downloading several large files. The figure shows the total number of device requests made by the guest, the number that needed to be hooked to drive the NE2000 model, and the ratio. Notice that typically only about 1 in 30 I/O port reads/writes need to be hooked to drive the model. The average per-request cost of updating the model is 1360 cycles.

5 Conclusions

We proposed a new technique for I/O virtualization, virtual passthrough I/O (VPIO), for commodity I/O devices, and presented an initial evaluation of its prospects. We are now working to incorporate the VPIO framework and the NE2000 model into Palacios to do complete evaluation of the VPIO concept.

It is clear that while we can reduce the number of hooked I/Os through modeling, the high cost of I/O interception in the VMM is the most problematic issue with the VPIO model. We are exploring how to reduce this cost by pushing as much of the model as possible into the guest OS through code injection. The model would then only cause exits from the guest under unusual conditions. Of course, this means the VMM must be able to dynamically insert binary code into the guest, transform guest code it finds (e.g., the guest's device driver needs to have its I/O operations changed to calls to the model), and guarantee that I/O operations cannot occur outside of those in the injected code.

While our analysis of the feasibility of VPIO focuses on I/O operations, the mechanisms exist to extend the approach to memory mapped devices. In addition to intercepting I/O instructions VMMs are also fully capable of intercepting memory reads and writes through shadow or nested page tables. We believe that adding this functionality is a straightforward exercise.

What we have not addressed in this paper is how to

safely handle device input that is being multiplexed to different VMs. Specifically in the case of a network card where network traffic can arrive anytime. We can only say that it is currently unclear exactly how to extend VPIO to allow guests to securely receive data destined for another guest. However, it should be noted that while that is a very real issue for a device such as a network card, other devices don't necessarily follow that I/O model. We should also acknowledge that it is increasingly becoming apparent that device manufacturers are beginning to look at designing self-virtualizing devices. While self-virtualization is a powerful abstraction, we note that hardware virtualization techniques have yet to fully prove that they can offer better peformance than software based approaches [1].

Finally, we conclude by noting that device models for VPIO functionality could readily be provided by hardware manufacturers. A model such as that of Figure 3 is essentially a behavioral model that is already produced as part of the design and verification process. For this reason, models for past, present, and future devices could be readily created.

References

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 2–13.

[2] AMD CORPORATION. AMD64 virtualization code-named "pacific" technology: Secure virtual machine architecture reference manual, May 2005.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (October 2003).

[4] BELLARD, F. Qemu: A fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track* (April 2005).

[5] BUNGALE, P., AND LUK, C.-K. Pinos: A programmable framework for whole systtem dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE)* (June 2007).

[6] DUNKELS, A. Full tcp/ip for 8-bit architectures. In *Proceedings of the first international conference on mobile applications, systems and services (MOBISYS)* (May 2003).

[7] HOVENMEYER, D., HOLLINGSWORTH, J., AND BHATTACHARJEE, B. Running on the bare metal with geekos. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)* (2004).

[8] INTEL CORPORATION. Intel virtualization technology specification for the ia-32 intel architecture, April 2005.

[9] LAWTON, K. Bochs: The open source ia-32 emulation project. http://bochs.sourceforge.net.

[10] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOETZ, S. Unmodified device driver reuse and improved system dependability. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (2004).

[11] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (May 2006).

[12] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementatin (PLDI)* (June 2005).

[13] RAJ, H., AND SCHWAN, K. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2007).

[14] SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., ZWAENEPOEL, W., AND WILLMANN, P. Concurrent direct network access for virtual machine monitors. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 306–317.

[15] SUGERMAN, J., VENKITACHALAN, G., AND LIM, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference* (June 2001).