Investigating Virtual Passthrough I/O on Commodity Devices

Lei Xia Jack Lange Peter Dinda Chang Bae
{lxia, jarusl, pdinda, changb}@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 two commodity network cards 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) [20] 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 [4] 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 [13]. 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 [14, 16, 18] 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]
- 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]

Effort is funded by the National Science Foundation (NSF) via grants CNS-0709168, CNS-0707365, and the Department of Energy (DOE) via a subcontract from Oak Ridge National Laboratory (ORNL) on grant DE-AC05-00OR22725.
This paper is an extension of our earlier publication on VPIO at the First Workshop of I/O Virtualization (WIOV'08). It also describes an implementation of VPIO in our Palacios VMM, and a performance evaluation using that implementation.

[1]Potentially, such a model could be provided by the device vendor. 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 and access to memory-mapped regions for control or DMA) 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.

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 extent to which the overhead of an exit can be reduced.

The work most closely related to VPIO is that of Williams, et al [22], which is contemporaneous.[5] Similar to VPIO, this work uses a model of the device to validate device driver interactions with the device against a security policy. Their goal is to be able to move device drivers out of the trusted computing base of a kernel. The goal in VPIO is to let a guest kernel interact with a physical device in a secure and controlled manner, and to multiplex the device, if possible.

In this paper, we describe the VPIO idea in more detail, and we show our results in applying VPIO to commodity network cards. Our results support the claims given above. We demonstrate that we can model the network cards, drive the models with guest/device interactions, and determine when the cards can be handed off. The models themselves are quite inexpensive. We have built an implementation of the VPIO concept as well. Our preliminary results indicate that with VPIO, the number of exits that the VMM must handle to support the guest's interaction with the NIC, is half as many needed to support a traditional virtual NIC implementation. In terms of virtualization overhead, this places VPIO squarely between a passthrough or self-virtualizing NIC approach and a traditional full virtual NIC approach.

2. PALACIOS

Palacios[6] is an OS independent VMM designed as part of the the V3VEE project (`http://v3vee.org`). The V3VEE project is a collaborative community resource development project involving Northwestern University and the University of New Mexico. It seeks to develop a virtual machine monitor framework for modern architectures (those with hardware virtualization support) that will permit the compile-time creation of VMMs with different structures, including those optimized for computer architecture research, computer systems research, operating systems teaching, and research and use in high performance computing. Palacios is the first VMM from the project and will form the basis of the broader framework. Support for high performance computing significantly informed its design.

[4]Think of an OUT instruction on a port associated with the device.

[5]The initial publication of this work was at the Workshop on I/O Virtualization [23].

[6]Palacios, TX is the "Shrimp Capital of Texas"

Component	Lines of Code	
	sloccount .	wc *.c *.h *.s
Palacios Core (C+Assembly)	15,084	24,710
Palacios Virtual Devices (C)	8,708	13,406
XED Interface (C+Assembly)	4,320	7,712
Kitten Glue Module (C)	272	428
GeekOS Glue Module (C)	238	495
Total	28,621	46,751

Figure 1: Lines of code in Palacios and its OS interface modules measured with the SLOCCount tool and with the wc tool.

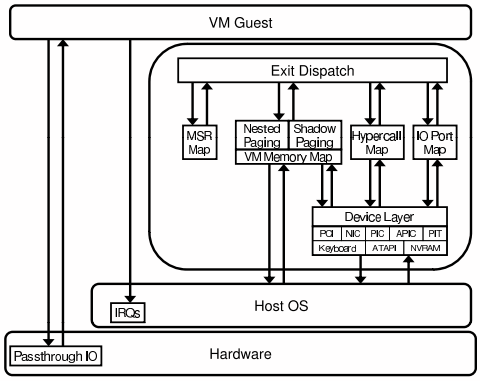

Figure 2: Palacios architecture.

Palacios currently targets the x86 and x86_64 architecture (hosts and guests) and makes extensive, and non-optional use of the AMD SVM [3] extensions (partial support for Intel VT [8, 21] is also implemented). Palacios uses Intel's XED library from Pin [15, 5], to decode instructions in some cases, and it uses the BOCHS [12] BIOS and VGA BIOS to bootstrap a guest machine. Palacios supports both 32 and 64 bit host OSes as well as 32 and 64 bit guest OSes[7]. Palacios supports virtual memory using either shadow or nested paging. It runs directly on the hardware and provides a non-paravirtualized interface to the guest with optional paravirtualized extensions. An extensive infrastructure for hooking of guest resources facilitates extension and experimentation.

Palacios was developed from scratch at Northwestern University. Figure 1 shows the scale of Palacios, as measured by two different source code analysis tools. Note that the Palacios core is quite small. The entire VMM, including the default set of virtual devices is on the order of 28–47 thousand lines of C and assembly. Palacios is publicly available from `http://v3vee.org`, and a technical report [11] describes the initial release in detail. The second release is expected in April, 2009. Palacios is released under a BSD license.

2.1 Architecture

Palacios is an OS independent VMM, and as such is designed to be easily portable to diverse host operating systems. Currently, Pala-

[7]64 bit guests are only supported on 64 bit hosts

cios actively supports Sandia National Lab's Kitten operating system, for high performance and multicore computing environments, as well as GeekOS [7], an educational operating system developed to teach operating system development. Palacios integrates with a host OS through a minimal and explicitly defined functional interface that the host OS is responsible for supporting. Furthermore, the interface is modularized so that a host environment can decide its own level of support and integration. Less than 500 lines of code needed to be written to embed Palacios into Kitten or GeekOS. Palacios is designed to be internally modular and extensible and provides common interfaces for registering event handlers for common operations. Figure 2 illustrates the Palacios architecture.

Resource hooks. The Palacios core provides an extensive interface to allow VMM components to register to receive and handle guest and host events. Events that can be hooked include

1. guest model specific register reads/writes (MSR hooks),
2. guest I/O port reads/writes (I/O hooks),
3. guest physical memory reads/writes (memory hooks),
4. host interrupts for redirection into the guest (interrupt hooks),
5. host OS events such as keystrokes, passage of time, etc (host events), and
6. guest hypercalls.[8]

It is also straightforward for VMM code to inject interrupts and exceptions into the guest. This combined functionality makes it possible to construct a wide range of different guest environments. We include a configuration interface that supports common configuration options (amount of memory, selection of virtual and physical devices, etc).

Host OS interface. The host OS interfaces with Palacios through a small set of functions:

- `allocate_pages()` and `free_pages()`: Allocates and frees physical memory pages,
- `malloc()` and `free()`: Allocates and frees kernel heap memory,
- `vaddr_to_paddr()` and `paddr_to_vaddr()`: Translates between host virtual addresses and host physical addresses.
- `hook_interrupt()` and `ack_interrupt()`: Passes an interrupt directly to a guest, and acknowledges it.
- `get_cpu_khz()`: Determines CPU clock rate.
- `yield_cpu()`: Yields the CPU to the host OS.

In addition to this interface, Palacios also includes an optional socket interface that consists of a small set of typical socket functions.

Palacios jointly handles interrupts with the host OS. In general, Palacios can disable local and global interrupts in order to have interrupt processing on a core run at times it chooses. For the most part, handling interrupts correctly requires no changes on the part of the host OS. However, for performance reasons, and for complicated interactions such a passthrough devices, small host OS interrupt handling changes may be necessary.

[8]Although Palacios is not a paravirtualized VMM, we do allow direct guest calls to the VMM.

2.2 Palacios as a HPC VMM

Part of the motivation behind Palacios's design is that it be well suited for high performance computing environments, both on the small scale (multicores) and on the large scale (Palacios runs on Sandia's Red Storm large scale distributed memory parallel machine). Palacios is designed to interfere with the guest as little as possible, allowing it to achieve maximum performance. Several aspects of its implementation facilitate this:

- Minimalist interface: Palacios does not require extensive host OS features, which allows it to be easily embedded into even small kernels, such as Kitten[9] and Catamount [10].
- Full system virtualization: Palacios does not require guest OS changes. This allows it to run existing kernels without any porting, including lightweight kernels [17] like Kitten, Catamount, Cray CNL [9], and IBM's CNK [19].
- Contiguous memory preallocation: Palacios preallocates guest memory as a physically contiguous region. This vastly simplifies the virtualized memory implementation, and provides deterministic performance for most memory operations.
- Passthrough resources and resource partitioning: Palacios allows host resources to be easily mapped directly into a guest environment. This allows a guest to use high performance devices, with existing device drivers, with no virtualization overhead.
- Low noise: Palacios minimizes the amount of OS noise [6] injected by the VMM layer. Palacios makes no use of internal timers, nor does it accumulate deferred work.

2.3 Symbiotic virtualization

Palacios also serves as a platform for research on symbiotic virtualization, a new approach to structuring VMMs and guest OSes so that they can better work together without requiring such cooperation for basic functionality of the guest OS either on the VMM or on raw hardware. In symbiotic virtualization an OS targets the native hardware interface as in full system virtualization, but also optionally exposes a software interface that can be used by a VMM, if present, to increase performance and functionality. Neither the VMM nor the OS needs to support the symbiotic virtualization interface to function together, but if both do, both benefit. Symbiotic virtualization has the potential to provide the compatibility benefits of full system virtualization while providing an incremental path towards the functionality and performance benefits possible with paravirtualization. The high performance computing context provides a special opportunity for symbiotic virtualization because there can be a much greater level of trust between the VMM, guest OS, and applications.

Although symbiotic virtualization is an important concept in Palacios, it is not currently used in VPIO. VPIO currently treats the

[9]A paper is forthcoming. More information, including source code, for Kitten is available at `http://software.sandia.gov/trac/kitten`

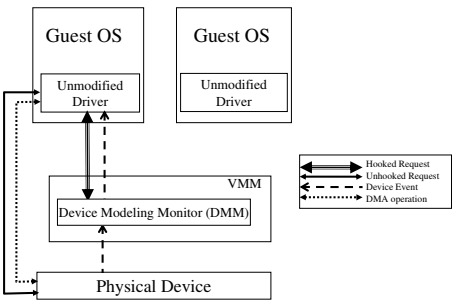

Figure 3: VPIO system.

guest as black box. We describe, in the conclusion, how VPIO could leverage symbiotic virtualization for further performance enhancements.

3. VPIO CONCEPT

We now describe the general VPIO concept as shown in Figure 3. 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 events from physical devices and delivers them to guests, drives an internal device state model for each guest, and can determine whether the device can be handed off to another guest, and what memory addresses a DMA operation will involve. In essence, the DMM is responsible for vetting device requests made by the unmodified device driver in the guest OS. Only device requests necessary to maintain protection of the VMM and other guests need to be vetted.

3.1 Device requests and events

The guest's device driver talks to physical devices by *device requests* made via I/O port reads/writes or by memory reads/writes. In the VPIO system, these device requests either directly go to the physical devices without VMM intervention, or are intercepted by the VMM for further processing. The DMM also intercepts *device events* on the physical device, such as device interrupts.

For performance, it is critically important that the DMM intercept *only* the device requests and events that are necessary to successfully drive the device state model. Generally, all device events are needed, but only a subset of the possible device requests are necessary. The set of device requests to intercept is chosen dynamically by the DMM. It maintains a *hooked device requests* list, a list of the kinds of device requests that currently must be intercepted. The *unhooked device requests* list are device requests which the device supports, but that the model does not currently need.

Reducing the size of the hooked device requests list, and thus the overall number of device request interceptions is critical for performance. The hooked device requests 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 needed to be hooked by the VMM.

3.2 DMA

DMA is essential for high performance devices. In VPIO, we allow the guest to directly initiate DMA operations at guest physical memory addresses. Notice that before DMA starts, the guest device driver must set it up, using device requests that convey the DMA parameters (starting address, length of the data, etc). By hooking the relevant device requests, we acquire these parameters and maintain them as part of the device state model. For some devices, the device state model may also be able to simply read these parameters directly from the physical device. The device state model alerts the DMM when a DMA is about to be started, and what the source/target 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.

A possible special case exists for a VMM running a single guest, as appears likely to 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.

3.3 Device multiplexing

VPIO multiplexes a physical device among multiple guests by essentially context switching the device from guest to guest. The device state 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/device that includes the device state model, values of all relevant physical device control 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 the physical device's control registers, flags and device model to the current guest's device context, and then restores these with the values from the device context of the guest that is the next owner of the device.

3.4 Device state model

The DMM maintains a *device state model* for each guest/device that keeps track of the current status (e.g, reusability, DMA operation starting, etc) of the physical device as seen from the guest. The device state model is updated by device requests (e.g. I/O port and memory reads/writes) and physical device events (e.g. interrupts, device faults).

Unlike a behavioral model, or a hardware model intended for verification purposes, the aim of the device state model 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 the model needs to see to update itself.

A device model is conceptually a state machine with additional scratchpad information (e.g., DMA addresses). The edges are annotated with the device requests and physical device events 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 device request list.

In designing a device model, we seek to keep its size (in terms of number of states, number of transitions, and number of checking functions) as small as possible while still being able to answer the questions given above. One or more states must be marked as being reusable in the model if it is to be possible to context switch the device. Without such a marking, the device can only be assigned to a single guest.

3.5 Dealing with failure

A natural question that arises is what the DMM should do if the device state 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 legitimately allocated memory. In this case, 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 invalid 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.

3.6 Dealing with device handoff on interrupt

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

At the present time, we have not yet found a general purpose solution for this problem. Such a solution would allow either efficient device hand offs to the appropriate guest on an interrupt, or efficiently moving data received in the wrong guest to the appropriate guest.

3.7 Performance concerns

The performance of VPIO for a given device is highly dependent on (1) the complexity of updating the model, (2) the number of device requests and events that must be intercepted, (3) the cost of such interception, and (4) the cost of context switches. On its face, VPIO looks like an expensive idea, but there are ameliorating elements. First, the hooked device request list may be small for a device, or for some modes of operation of the device. Second, most of the cost of hooked device request interceptions is due to VM exit and entry and thus will get cheaper with better hardware or software support. Finally, the context of a device that needs to be changed for a context switch may be small.

We consider performance in more detail later, but here we would like to give initial consideration to (1), (3), and (4). In Figure 4, we report CPU cycle timing using an emulated NE2000 network card in the QEMU x86_64 (w/ SVM) emulated processor environment, as well as an HP Proliant ML115 with an AMD Opteron 1210 processor. While unhooked device requests operate at the speed of the hardware, hooked device requests are dominated by the cost of a VM exit, its handling, and the VM entry. Although the data needs to be taken with a grain of salt, given that some of it is from an emulated environment, it shows clearly that we must ameliorate the high costs of hooked device requests by reducing their occurrence, and reducing the cost of VM exit/entry handling. The former is our opportunity, the latter depends on advances in hardware support for virtualization. Notice, however, that the cost of a NE2000 context switch is small—on par with handling a hooked device request.

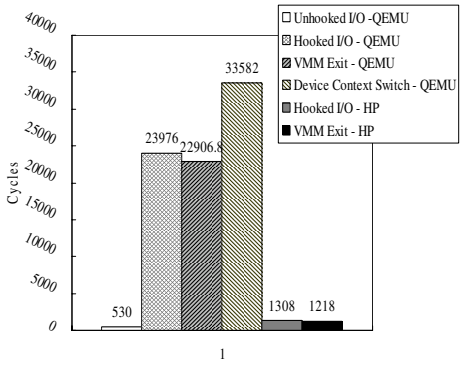

Figure 4: Costs of VPIO operations in Palacios for an NE2000 network card.

Device request and event *representational granularity*, combined with virtualization *interception granularity*, is an important device-dependent feature that affects performance. Hardware and software virtualization technologies limit the granularity at which device requests can be observed. For example, with the AMD SVM extensions, it is easy to intercept reads and writes on a per-I/O port basis, but memory addresses are interceptable only on a per-page basis. If the device designer places multiple control registers/control register fields at the same I/O port or memory address[10], or even on the same page, more exits will be needed than are strictly necessary given the fields/registers being changed.

4. EXAMPLE DEVICE MODELS

To test the claim that efficient VPIO device models can be built, we developed models for two different network cards, the NE2000 and the RTL8139. We now explain these models. The integration of the models into Palacios's virtual network card (VNIC) functionality is discussed later.

The two device models were developed by hand, and each is based on a state machine, an abstraction that is relatively simple to understand and (we claim) to implement. However, this is obviously not the only way to build a device model, especially for more general devices. It is not the structure of the device model that is important, but rather its existence and quick response to device requests and

[10]Multiple conceptual registers can be packed into one I/O register using bitfields, or a device can use an address register plus data register construction.

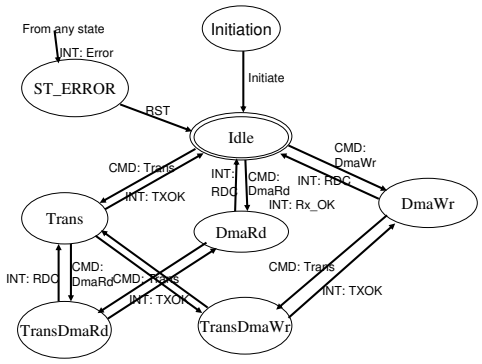

Figure 5: Device model for NE2000 NIC.

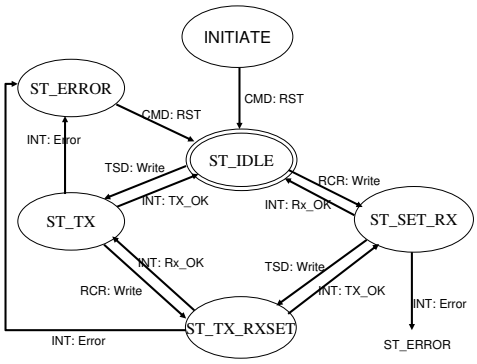

Figure 6: Device model for RTL8139 NIC.

events. Ideally, a device model for VPIO would be provided by the device manufacturer. We claim that this could be done as a supplement to the manufacturer's internal formal description of the device interface and behavior. Leveraging existing and future behavioral models would make it possible to create VPIO device models for past, present, and future devices.

4.1 NE2000 model

The NE2000-compatible network card (specifically, the Realtek RTL 8029A chipset) is a relatively simple network card. It supports DMA for sends and receives using ring buffers.

The NE2000 model is an augmented finite state machine as described in Section 3.4, and is illustrated in Figure 5. In the model, each state represents one or a group of device register contents that correspond to a device state of interest to the DMM. The arrows between states represent state transitions, while their annotations are the events that drive these transitions. "Cmd: xxx" means a write request (a device request) on the control registers by the guest (this is how the guest starts an operation on the card, such as transmitting a packet, starting a remote DMA transfer, etc). "INT: xxx" means an interrupt (device event) was received from the physical device (for example, a packet transmission completion or a packet arrival). The actual I/O port numbers and masks are not shown in the figure. The model's complete implementation consists of approximately 900 lines of C code.

Checking functions are associated with some edges. For example, whenever entering the DmaRD (DMA read is running) or DmaWr (DMA write is running) state, the checking functions validate the DMA parameters (such as destination/source physical memory address and the transfer length). For this specific card, the device 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 the VMM's policy, a device failure would be reported to the DMM. On the NE2000, the DMA-initiating I/O port write request can either be ignored, or a "remote DMA failed" interrupt can be delivered to the guest.

The NE2000's 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 the other guest.

4.2 RTL8139 model

The RTL8139 (specifically, the Realtek RTL 8139D chipset) is a more recent, and more widely used network card. It uses a more general form of DMA to transmit and receive packets. Multi-word DMA is automatically performed without the intervention of the guest's device driver once the driver has configured the DMA transfer. Five control registers are used to describe a DMA operation, while two registers (Transmit Status Descriptor and Receiver Configure Register) are used to initiate the DMA transfer.

Figure 6 illustrates our device model for the RTL8139. The actual implementation is 1300 lines of C code. Although the RTL8139 is more complicated than NE2000, its device model is of comparable complexity. The primary reason for this is that most of the physical status is invisible to the device driver, and thus not of concern to our device model.

5. VPIO-NIC IMPLEMENTATION IN PALACIOS

We have implemented VPIO versions of the NE2000 and RTL8139 network cards in Palacios, along with passthrough and fully emulated versions. The VPIO network device that supports these operating modes is referred to as the VPIO-NIC.

5.1 VPIO-NIC: DMM

The VPIO-NIC design separates the model-independent component (i.e., the DMM) from the specific implementation of the device model. In principle, the DMM should be reusable with other network interface card models.

Device requests. In Palacios, the guest's device driver talks to physical devices either by IN and OUT instructions to relevant device I/O ports (Programmed I/O or PIO) or by reads and writes to memory-mapped registers (Memory-Mapped I/O or MMIO) of the device. For PIO, Palacios provides the ability to intercept any I/O port reads or writes using AMD SVM or Intel VT hardware support. An I/O port read or write in the guest causes a VM exit to Palacios, which decodes the instruction and vectors the read or write to VPIO-NIC. If the operation is allowed, VPIO-NIC can then handle it. For MMIO, Palacios supports memory read/write interception on page size granularity using either shadow paging or nested paging techniques. We support both PIO and MMIO devices in our current VPIO-NIC implementation.

For a device using I/O ports, such as the NE2000, VPIO-NIC maintains a *hooked I/O* list—guest reads or writes to the I/O ports on the list are intercepted and handed off to VPIO-NIC. VPIO-NIC uses these operations to update the device state model. The other device I/O ports are kept on the *unhooked I/O* list. Both lists can be dynamically updated based on the device state model.

The hooked I/O list and hooked memory page list are stored in separate Red-Black trees that provide logarithmic lookup times. However other implementations, such as hash tables, are possible. For a typical device, with < 100 hooked resources, access time is reasonable. Each hooked resource is associated with callback functions for reading and writing, which, in this case, point back into VPIO-NIC.

Device state model interface. The design of the VPIO-NIC DMM attempts to make it independent of the implementation of the device state model. Ideally, device state model implementations act as independent components that can be selected at compile- (and eventually run-) time. VPIO-NIC provides a standard interface for interacting with the device model. VPIO-NIC expects the following interface from the device model:

- `init_model()` : Initiate the device model, setting up initial environment variables for both the DMM part and the device model part. The VPIO-NIC functions like fully virtualized NIC until the device model is initialized.
- `update_model(event-type, port_number/memory/interrupt, read/write, value, other_info)` : Update the model with the event. The function can return success or failure. If success is returned, this means the model successfully transitioned to the next state. If failure is returned, the model indicates the reason for the failure. Event here refers both to device requests from the guest and device events from the physical device.
- `check_model()` : This indicates whether the device is reusable, or if a DMA has been initiated. In the latter case, the address is also indicated.
- `deinit_model()` : Detach the currently used device model, remove any port or memory hooks and other dependent parameters for this device model.

The device state model uses the following interface to hook device requests and access raw devices:

- `hook_port(port_number, read/write)` : Hook the port on read, write or both.
- `unhook_port(port_number, read/write)` : Unhook the previous hooked port access on read, write or both.
- `port_access(port_number, read/write, return/write_value)` : Read or write from/to the port on the physical device.
- Similar functions for hooking and unhooking memory regions.

Interrupts. Interrupts are generated by network card devices to notify the device driver that an event has occurred, for example, a packet has been transmitted, a packet has been received, a device error has occurred, etc.

In Palacios, an interrupt that occurs while the guest is running causes a VM exit. Network card interrupts are hooked using Palacios so that they ultimately vector into VPIO-NIC.[11] The VPIO-NIC handler determines the interrupt type by reading the interrupt status registers of the physical device and updates the device state model with the interrupt event and type. If the model is updated successfully, VPIO-NIC injects a copy of the interrupt into the guest using Palacios's virtual PIC or APIC. If the model is not updated successfully, VPIO-NIC can choose not to inject an interrupt or to inject a different kind of interrupt. Interrupts can be injected at other times as well. For example, if the model indicates that the guest is initiating an illegal DMA, VPIO-NIC might inject a "DMA failure" interrupt.

For most devices, the guest device driver determines the reason for an interrupt by reading interrupt-related status registers. For this reason, VPIO-NIC virtualizes all of the interrupt status registers instead of allowing the guest to access the physical registers directly.

DMA. DMA operations need to be handled carefully in VPIO-NIC. In the case of transmitting a packet the guest typically stores a packet of data at some guest physical address, and informs the NIC of this address and the length of the packet to the device by writing these values to a set of control registers (ports). After that, the device driver issues a "DMA start" command by writing to a specific command register (or registers). The NIC will then start transferring the data from the specified source memory address to the destination address (typically, via a ring buffer). When done, it will raise an interrupt. The process for packet reception is similar.

In principle, VPIO-NIC knows when the guest is initiating a DMA because of its interception of read/writes to the command registers. Additionally, it can determine the DMA parameters (source/destination memory address, transfer length) by either intercepting the write operations to the relevant registers, or by reading the address back from the relevant registers on the NIC just before the DMA is initiated. The latter option can further reduce the device requests that need to be intercepted.

The device state model plays the principle role in detecting initiation of DMA since it is familiar with the device. In particular, it notifies VPIO-NIC that the subsequent state is the DMA operation state. The checking function is called before going to the DMA state to validate the DMA address and other parameters. If the parameters are legal, the device state model returns success and VPIO issues the DMA start command to the physical NIC. The check function is also responsible for setting the ultimate DMA source or destination address. This is important as the guest device driver uses guest physical memory addresses (GPA) when it issues DMA operations, and these are usually not the same as the host physical memory addresses used by the physical device. The model can also determine that the DMA is illegal and return failure. When this occurs, the response is device-dependent.

[11] This is strictly the case when passthrough or virtual passthrough support is used. When the fully virtualized NIC support is used, the interrupt is handled by the host OS's device driver, which in turn creates a host event for VPIO-NIC.

Device failure. When the device state model shows that the guest is about to put the device into an improper state (if, for example, the guest has initiated a DMA write to memory the guest does not own) the guest's device request should not be allowed. How VPIO-NIC handles the failure depends on the failure type, the underlying physical device, and the guest operating system.

The VPIO-NIC provides several options to respond to a device failure. If the failure to update the device model is due to an illegal I/O port access (a write to a disallowed port, or a write of an illegal value), VPIO-NIC provides two options for handling it. The first is to ignore the I/O request and let it fail silently. In this case writes are discarded, while reads returns zero. Another option is to inject a machine check exception into the guest, typically making the guest OS halt. This option is used when the guest tries to access sensitive ports where silent failure would likely confuse the guest. The hooked I/O port list structure includes the failure handling method, and can be changed by the device model during run time.

If the failure to update the device model is caused by an illegal DMA operation (an illegal memory address or illegal device state), the VPIO-NIC can respond in three possible ways. The basic response is silent failure—a DMA read returns zeros while a DMA write is ignored. The assumption here is that the guest device driver or higher level code will fix the error. For example, TCP can retransmit the packet. Another option is for VPIO-NIC to inject a virtualized device interrupt into the guest (DMA failure, send or receive failure, etc) to notify it of the failure. The final option is to inject a machine check. In this case, the choice of failure handling is made at compile time.

Reusable state and device context switch. When the current state of the device model is a reusable state, it is safe for VPIO-NIC to context switch the physical NIC to a different guest. The VPIO-NIC device context contains all of the visible registers of the NIC, a copy of the accessible on-chip memory (if any), the device state model for that specific guest, and other control information for VPIO-NIC (values contained in the virtualized device registers, flags, etc). Once the current device state model indicates the device is in a reusable state, and there is another guest requesting the device, VPIO-NIC stores the device context for the current guest, resets the device, and loads the device context corresponding to the new guest, including all of the register contents.

5.2 NE2000

We now discuss the part of the VPIO-NIC implementation that is specific to the NE2000 NIC.

Device model. As described in Section 4.1, The NE2000 device model is implemented as an extended state machine, including states, state transitions, events, and checking functions. This includes a "reusable" state (Idle), as well as an "illegal" state, which is the state transitioned to if an unexpected event occurs. In our implementation, each state and event is represented as a unique index with attached information. States and events are stored in hash tables.

The state machine transitions are represented as a matrix that uses the current state and event as indices. Each entry contains the subsequent state (possibly the illegal state), and the checking function associated with the state transition. For the NE2000, there are about 10 states with 20 events/requests, yielding a matrix with only $\tilde{2}00$ entries.

Device requests (I/O port accesses) and interrupts are mapped to an internal event type. An event table translates the event data supplied by VPIO-NIC (the arguments to `update_model()` to an internal event that is used to drive the state machine.

Any necessary address and security validations are performed inside the checking function. In our current version, this function validates the DMA destination address, which it determines by reading from the control registers (Remote Address and Remote Count registers) on the physical NE2000 NIC.

The `update_model()` call supplies the current event parameters as either *{port number, read/write, value}* or *{interrupt, interrupt status value}*, which are translated to the internal event type. Using the current state and the current event, the model finds the corresponding entry in the state transition matrix. If this indicates that the next state is legal, it runs the attached checking function. If the checking function returns true, it sets the current state to the next state and returns success. If the next state is illegal or the checking function returns false, the current state is not changed and `update_model()` returns false with a specific code for the failure.

There are three approaches for handling device failure in the NE2000 VPIO. If the failure is caused by a read on an invalid port, the read returns a NULL value. Writes to invalid ports fail silently. For an invalid DMA operation, a DMA failure interrupt is injected into to the guest to notify the device driver. A DMA error is signified by setting the transmit error bit in the interrupt status register. For other events that would result in a transition to an illegal state, VPIO-NIC injects a machine check exception into the guest.

Interrupts. The guest accesses the Interrupt Status Register (ISR) to determine the cause of an interrupt. The guest can also mask any interrupt by setting the corresponding bit in the Interrupt Mask Register (IMR). Details on the status of the most recent packet transmission and reception can be read from the Transmit Status Register (TSR) and Receive Status Register (RSR). In the NE2000 VPIO-NIC, reads and writes to these four registers are always hooked and virtualized. This allows for both physical device interrupts and software-generated interrupts to be handled in a common manner.

DMA. DMA operations on the NE2000 NIC function as follows:

1. A pair of registers denoted as RBCRx are loaded with the data size to be transferred.
2. Another pair of registers, RSARx, are loaded with the low and high byte of the target DMA address (normally this address should fall in the device's ring buffer).
3. The COMMAND register is set to "start" and "remote write (or read) DMA".
4. Packet data is now written to or read from the "Data Port" of the NIC in a loop. The NIC updates its remote DMA logic after each byte and places the byte into the target memory address.
5. Finally, the NIC sets the "Remote DMA completed" bit in the ISR register.

The NE2000 VPIO-NIC does not have to actually hook and virtualize the DMA parameter register pairs (RBCRx and RSARx) since

it can simply read their values directly from the device. When the guest writes to the COMMAND register (step 3), the state model changes to the DMA read or write state after the checking function validates and adjusts the target DMA address.

5.3 RTL8139

The device state model implementation for the RTL8139 NIC is very similar with the NE2000 model described above. The primary difference is that the RTL8139 uses memory-mapped I/O instead of programmed I/O. This has important implications for performance since memory can only be hooked at the page level. On the RTL8139, all of the control registers are mapped into a 256 byte memory region. Since memory can only be hooked per page, access to *any* of these registers will cause a VM exit, even if the address does not correspond to a register needed by the device model. Accesses to registers required by the model are handled in a similar fashion as the NE2000 implementation. All other requests are simply applied directly to the physical device, albeit with the overhead of the VM exit/entry.

5.4 VNIC

VPIO-NIC also supports a mode of operation in which it behaves like a traditional, fully virtualized network device of the relevant type. To implement this, we ported the virtual NE2000 and RTL8139 NIC implementations from QEMU to Palacios.

5.5 Passthrough NIC

VPIO-NIC can also operate in a pure passthrough mode in which the guest has direct access to the underlying network card. No I/O or memory operations by the guest cause VM exits. Here, the NIC is fully assigned to only a single guest. However, interrupt handling still requires VM exits, since Palacios requires exits on all interrupts to allow the host OS the opportunity to handle them. When an interrupt from the physical NIC occurs, the CPU performs a VM exit, Palacios injects the interrupt into the guest, and performs a VM entry. The only effect is that interrupt latency increases due to the cycles required for a VM exit/entry.

6. PERFORMANCE

We now present the results of an initial performance evaluation of VPIO-NIC. The high level results are the following:

- Most importantly, the number of VM exits needed to support VPIO is greater than the number needed for passthrough I/O, but less than the number needed by a traditional virtualized NIC (VNIC). This supports our claim that the VPIO concept is an interesting point in the design space for I/O virtualization, providing some of the performance of passthrough I/O without the security issues.
- The VM exit handling cost for VPIO-NIC is greater than that for passthrough I/O, but less than that of a traditional virtualized NIC. This also supports the case that VPIO is an interesting design possibility.
- The achievable throughput of VPIO-NIC is between that of passthrough I/O and a traditional virtualized NIC.

Our results are preliminary. It is especially important to note that our evaluation is done in the QEMU emulator. Our VM exit results are robust, as the same exits occur on QEMU as in real hardware.

Scenario	Total I/O	I/O hooked	Ratio (%)
Linux: ssh	62393	21692	34.8%
Linux: small dl	212147	69700	32.9%
Linux: large dl	32945087	9429917	28.6%
Windows: ssh	184195	28667	15.6%
Windows: small dl	236065	39089	16.6%
Windows: large dl	2413277	413779	17.1%

Figure 9: Number of I/O exits for NE2000 Device Model in various network workloads.

However, the other numbers need to be taken with a grain of salt. We use the cycle counter to measure time, but because QEMU is not a cycle accurate emulator the values are not reflective of actual hardware. We are currently working to evaluate VPIO-NIC on real hardware.

6.1 Setup

We have evaluated the performance of passthrough, VPIO, and VNIC versions of both the NE2000 and RTL8139 network cards. We tested communication between a VM running on Palacios within QEMU and a physical machine. Specifically, we used a version of Palacios embedded into GeekOS to run (32 bit) Puppy Linux 3.0.1. Two instances of QEMU were run on a dual quadcore 1.6 GHz Intel Xeon with 8 GB RAM running Red Hat Enterprise Linux 5 (2.6.18 kernel), with each connected to a local network created with a TAP interface.

The primary benchmark we use is IPERF [1], communicating 1 MB of data over TCP using 64 KB socket buffers.

6.2 VM exits

As described in Section 3.7, the *number* of VM exits per packet transmission or reception is a serious performance impediment for virtualized network cards, but is almost entirely avoided for passthrough cards. To compare these techniques against VPIO, we measured the number of exits caused by device requests and device events. Recall that for passthrough I/O, all of the VM exits are caused by interrupts from the physical device being forwarded via Palacios.

Figure 7 shows the number of exits needed to send and receive a packet for passthrough I/O, VPIO, and VNIC implementations of the NE2000. The totals for VPIO and VNIC include exits due to I/O port access and interrupts. As can be seen, the number of exits for a VPIO implemantation are about half that of a VNIC implementation.

Figure 8 shows the equivalent results for the RTL8139. Again, the number of exits per packet for VPIO is about half of those for the VNIC. Again, the totals for VPIO and VNIC include exits due to interrupts and register accesses. However, it is important to note that we are only counting register accesses *that are needed*. However, since it is only possible to hook the RTL8139's memory mapped registers at page granularity, we are taking additional exits for accesses not required by the device model.

Figure 9 suggests why the number of exits needed for VPIO is so much lower than for the VNIC. Here, we have run a wider range of application performance tests, including interactive ssh and scp-based downloads of large and small files. The figure examines the number of I/O port interceptions needed to support the VPIO implementation. As we can see, only 15.6–34.8% of I/O port accesses

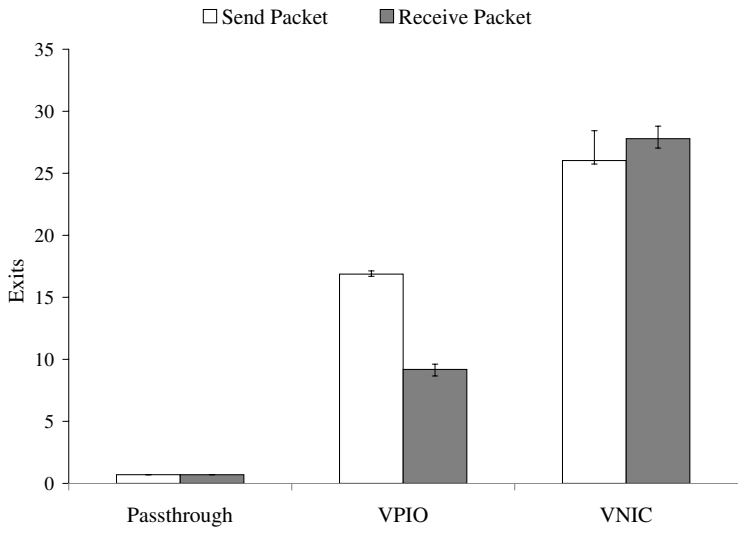

Figure 7: Average number of exits to send or receive a packet on the NE2000, comparing passthrough I/O, VPIO, and VNIC implementations.

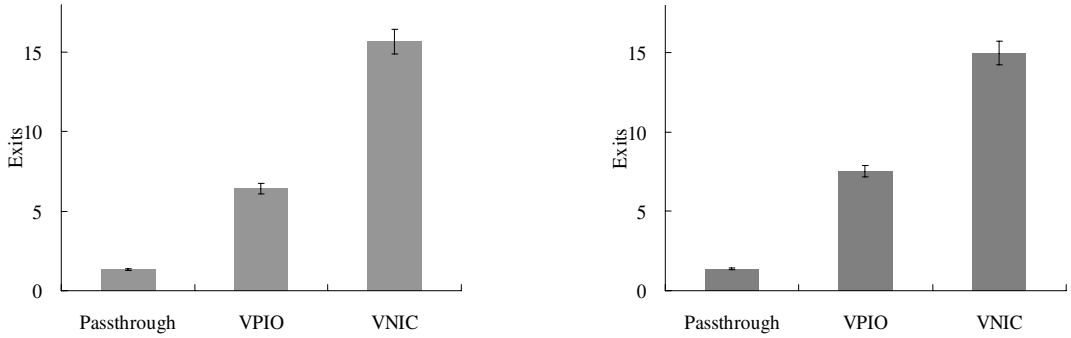

Figure 8: Average number of exits to send or receive a packet on the RTL8139, comparing passthrough I/O, VPIO, and VNIC implementations.

need to be intercepted in order to drive the VPIO version of the NE2000. The update cost for the NE2000 device model is on the order of 100 cycles (on the real hardware) per intercepted I/O.

VPIO offers an intermediate option which requires far fewer exits than a pure VNIC, while simultaneously providing security that a pure passthrough implementation cannot. The VPIO-NIC is more efficient than a pure VNIC in terms of the number of exits per packet needed to support it.

6.3 Exit costs

In addition to the number of exits per packet, it is also important to consider the costs of exit handling. After all, it could be that VPIO takes fewer exits, but each is more expensive. Figure 10 shows the average number of CPU cycles needed for exit handling per packet on the various implementations of the NE2000. As we can see, the total time per packet spent in the VMM for VPIO handling is, in fact, an order of magnitude less than that needed for the VNIC. It is, however, an order of magnitude more than for the passthrough option. Again, we see that VPIO is an interesting point in the design space between these two.

Figure 11 shows a further breakdown of the average overheads in-

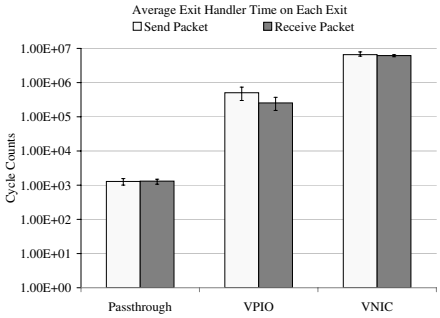

Figure 10: Average exit handling cost per packet for the NE2000, comparing passthrough, VPIO and VNIC options.

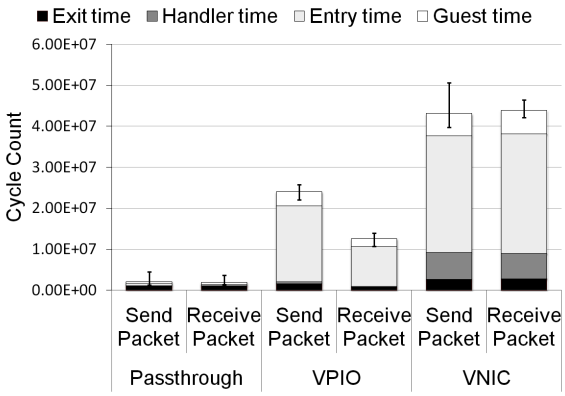

Figure 11: Breakdown of overheads for sending and receiving a packet on the NE2000 NIC.

curred in sending or receiving one 1500 byte packet. We break down the overheads into cycles spent in the guest, cycles spent in basic VM exit/entry processing, and cycles spent in the VMM to process device handling. While analogous measurements for the RTL8139 have not yet been completed, we expect them to be quite similar.

6.4 Application performance

We measured the throughput for the passthrough, VPIO, and VNIC versions of the NE2000 and compared this to native performance. The throughput was measured using IPERF, with the actual measurement done on a physical machine to get accurate timing. It is important to take the results with a grain of salt given that in all cases the tests were run inside the QEMU emulator, and not on real hardware.

Figure 12 shows the results. None of the virtualized options (Passthrough I/O, VPIO, VNIC) are able to achieve the throughput the NIC is capable of, although their performance does rank them as we might expect, with VPIO being intermediate in performance between Passthrough I/O and VNIC. We are currently investigating these results.

7. CONCLUSIONS

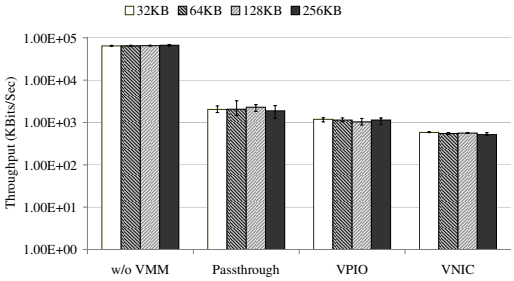

Figure 12: Network throughput using the different versions of the NE2000 device. We also show the performance achieved without virtualization.

We have proposed a new technique for I/O virtualization of commodity I/O devices, virtual passthrough I/O (VPIO). VPIO is an intermediate option between passthrough I/O and traditional fully emulated virtual devices. It provides some of the performance of the former, while maintaining the security/protection of the latter. VPIO implements a device state model in the VMM that vets guest access to the physical network card. The overhead of running the model is much less than the overhead of fully virtualizing the device.

We implemented VPIO versions of two network cards, the NE2000 and the RTL8139, and presented an initial performance evaluation that suggests the VPIO point in the design space for I/O virtualization is quite an interesting one. We found that the costs of a VPIO NIC, in terms of number of exits and exit handling cost per packet, are about half that of a traditional fully virtualized NIC.

The key challenge in further improving the performance of VPIO is to decrease the number of exits and their costs even more. It is clear that while we can reduce the number of device requests and events that we need to intercept through careful device modeling, the high cost of interceptions and VM exit/entry 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, a form of symbiotic virtualization. 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.

Hardware virtualization features could also enhance VPIO performance. Driving the raw overhead of VM exits/entries down would create significant benefits. Further, being able to intercept device requests at a finer granularity would be extremely beneficial. This is particularly the case for memory-mapped I/O. Current hardware supports interception only at the granularity of pages, which is far too large for many interesting devices.

This paper did not address 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 performance than software based approaches [2].

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 5 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.

8. REFERENCES

[1] http://sourceforge.net/projects/iperf.

[2] 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.

[3] AMD CORPORATION. AMD64 virtualization codenamed "pacific" technology: Secure virtual machine architecture reference manual, May 2005.

[4] 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).

[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] FERREIRA, K., BRIDGES, P., AND BRIGHTWELL, R. Characterizing application sensitivity to os interference using kernel-level noise injection. In *Proceedings of ACM/IEEE SC (Supercomputing)* (November 2008).

[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] KAPLAN, L. Cray CNL. In *FastOS PI Meeting and Workshop* (June 2007).

[10] KELLY, S., AND BRIGHTWELL, R. Software architecture of the lightweight kernel, Catamount. In *Proceedings of the 2005 Cray Users' Group Annual Technical Conference* (May 2005), Cray Users' Group.

[11] LANGE, J. R., AND DINDA, P. A. An introduction to the Palacios Virtual Machine Monitor—release 1.0. Tech. Rep. NWU-EECS-08-11, Northwestern University, Department of Electrical Engineering and Computer Science, November 2008.

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

[13] 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).

[14] 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).

[15] 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).

[16] 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).

[17] RIESEN, R., BRIGHTWELL, R., BRIDGES, P., HUDSON, T., MACCABE, A., WIDENER, P., AND FERREIRA, K. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience 21*, 6 (April 2009), 793–817.

[18] 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.

[19] SHMUELI, E., ALMASI, G., BRUNHEROTO, J., CASTANOS, J., DOZSA, G., KUMAR, S., AND LIEBER, D. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *Proceedings of the 22nd International Conference on Supercomputing* (New York, NY, USA, 2008), ACM, pp. 165–174.

[20] 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).

[21] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A., MARTIN, F., ANDERSON, A., BENNETTT, S., KAGI, A., LEUNG, F., AND SMITH, L. Intel virtualization technology. *IEEE Computer* (May 2005), 48–56.

[22] WILLIAMS, D., REYNOLDS, P., WALSH, K., SIRER, E. G., AND SCHNEIDER, F. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)* (December 2008).

[23] XIA, L., LANGE, J., AND DINDA, P. Towards virtual passthrough I/O on commodity devices. In *Proceedings of the Workshop on I/O Virtualization at OSDI* (December 2008).