A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery

Cheng-Chun Tu
Oracle Labs
cheng-chun.tu@oracle.com

Michael Ferdman
Stony Brook Univiersity
mferdman@cs.stonybrook.edu

Chao-tung Lee
Industrial Technology Research Institute, Taiwan
marklee@itri.org.tw

Tzi-cker Chiueh
Industrial Technology Research Institute, Taiwan
tcc@itri.org.tw

Abstract

As the performance overhead associated with CPU and memory virtualization becomes largely negligible, research efforts are directed toward reducing the I/O virtualization overhead, which mainly comes from two sources: DMA set-up and payload copy, and interrupt delivery. The advent of SRIOV and MRIOV effectively reduces the DMA-related virtualization overhead to a minimum. Therefore, the last battleground for minimizing virtualization overhead is how to directly deliver every interrupt to its target VM without involving the hypervisor.

This paper describes the design, implementation, and evaluation of a KVM-based direct interrupt delivery system called DID. DID delivers interrupts from SRIOV devices, virtual devices, and timers to their target VMs directly, completely avoiding VM exits. Moreover, DID does not require any modifications to the VM's operating system and preserves the correct priority among interrupts in all cases. We demonstrate that DID reduces the number of VM exits by a factor of 100 for I/O-intensive workloads, decreases the interrupt invocation latency by 80%, and improves the throughput of a VM running Memcached by a factor of 3.

Categories and Subject Descriptors C.0 [*General*]: Hardware/software interfaces

Keywords SR-IOV, I/O Virtualization, Interrupts, I/O Performance

1. Introduction

With increasingly sophisticated hardware support for virtualization, the performance overhead associated with CPU and memory virtualization is largely negligible. The only remaining non-trivial virtualization overhead is due to I/O virtualization. The I/O virtualization overhead itself mainly comes from two sources: setting up DMA operations and copying DMA payloads, and delivering interrupts when I/O operations are completed. The advent of SRIOV [12] and MRIOV [32] allows a VM to interact with I/O devices directly and thus effectively reduces the DMA-related virtualization overhead to a minimum [25, 27]. Therefore, the last I/O virtualization performance barrier is due to interrupt delivery. Because the main overhead of interrupt delivery are VM exits, a key approach to reduce the overhead of virtualized server I/O is to deliver interrupts destined to a VM directly to that VM, bypassing the VM exit and avoiding involving the hypervisor. Direct delivery of interrupts to their target VMs not only minimizes the performance overhead associated with I/O virtualization, but also decreases the *interrupt invocation latency*, a key consideration in real-time virtualized computing systems. This paper describes the design, implementation, and evaluation of a KVM-based direct interrupt delivery system called DID, which offers a comprehensive solution to eliminate interrupt delivery performance overhead on virtualized servers.

DID solves two key technical challenges for direct interrupt delivery. The first challenge is how to directly deliver interrupts to their target VMs without invoking the hypervisor on the delivery path. The second challenge is how to signal successful completion of an interrupt to the interrupt controller hardware without trapping to the hypervisor. We set the following goals at the onset of this project:

- If a VM is running, all interrupts for this VM, including those from emulated devices, SRIOV devices, timers, and other processors, are delivered directly.
- When a target VM is not running, its associated interrupts must be delivered indirectly through the hypervisor, but the priority among all interrupts, both directly and indirectly delivered, is correctly preserved.
- The number of VM exits required to deliver and complete an interrupt is zero.

VEE '15, March 14–15, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2731186.2731189

- No paravirtualization or modification to the VM's OS is needed.

To satisfy these goals, DID leverages several architectural features available on modern Intel x86 servers. First, DID takes advantage of the *interrupt remapping table* on the IOMMU to route interrupts to their target VM directly when the target VM is running and to the hypervisor when the target VM is not running, without requiring any changes to the guest OS. Moreover, the hypervisor can run on any of the available CPU cores. If an interrupt is delivered via the hypervisor, it becomes a *virtual interrupt* when it is delivered to the target VM. Second, DID leverages the interprocessor interrupt (IPI) mechanism to inject virtual interrupts from the hypervisor directly into a VM running on another core. This virtual interrupt delivery mechanism effectively converts a virtual interrupt back into a physical interrupt, eliminating a well-known problem of existing virtual interrupt delivery mechanisms, where lower-priority virtual interrupts may override higher-priority directly delivered interrupts because virtual interrupts do not pass through the local interrupt controller (local APIC). Additionally, DID employs special handling of timer interrupts, which do not pass through the IOMMU, and also avoids VM exits when delivering timer interrupts, regardless of whether they are being delivered to a VM or to the hypervisor. This is achieved through careful installation of timer interrupts that the hypervisor sets up on a dedicated core and migration of timer interrupts that a VM sets up when the VM is suspended and migrated.

Existing approaches to this interrupt delivery problem include software patches in guest OSes and host's kernel to enable hypervisor bypassing [18, 19, 31] when interrupts arrive, or vendor-specific hardware upgrades such as Intel's interrupt APIC virtualization support (APICv) [1]. DID takes a software-only approach and is proven to be more effective in reducing the number of VM exits as compared to the existing hardware solutions. After carefully examining existing software-based solutions [18, 19, 31], we have identified several major limitations in these solutions and removed all of them in DID. Specifically, existing solutions distinguish between *assigned interrupts* and *non-assigned interrupts*, and are able to directly deliver only assigned interrupts, which are usually from SRIOV devices. Moreover, these solutions suffer from a cascade effect in which the hypervisor has to turn off the direct interrupt mechanism of a VM while injecting a virtual interrupt to the VM and in the process creates more virtual interrupts. Finally, legacy or closed-sourced OSes cannot enjoy the benefits of these solutions because they require guest OS modification.

The current DID prototype is built into the KVM hypervisor [24] that supports direct pass-through for SRIOV devices. We demonstrate the following performance advantages of DID for virtualized x86 servers equipped with SRIOV NICs:

- The interrupt invocation latency of a cyclictest benchmark is reduced by 80%, from $14\mu s$ down to $2.9\mu s$.
- The intra-machine TCP-based iperf throughput is improved by up to 21%.
- The Memcached throughput, in terms of bounded-latency requests per second, is improved by 330% as a result of reducing the VM exit rate from 97K per second to less than 1K per second.

2. Background

On Intel x86 servers, system software performs the following sequence of steps to carry out a transaction with an I/O device, such as a NIC. First, the system software issues I/O instructions to set up a DMA operation for copying data from memory to the I/O device. Then, the DMA engine on the I/O device performs the actual copy and signals completion by sending an interrupt to the CPU. Finally, the corresponding interrupt handler in the system software is invoked to process the completion interrupt and to send an acknowledgement to the interrupt controller hardware.

In a naive I/O virtualization implementation, at least three VM exits are required to execute an I/O transaction: one when the I/O instructions are issued, another when the completion interrupt is delivered, and the third when the interrupt handler finishes. If an I/O device supports single-root I/O virtualization (SRIOV), a VM is able to issue I/O instructions directly to the device in a way that is isolated from other VMs, and therefore VM exits are avoided when I/O instructions are issued. However, despite SRIOV, the other two VM exits remain. The goal of DID is to eliminate the remaining two VM exits associated with each I/O transaction by delivering completion interrupts to their VMs directly and allowing the VMs to directly acknowledge interrupts, in both cases without involving the hypervisor.

2.1 Intel x86's Interrupt Architecture

On x86 servers, interrupts are asynchronous events generated by external components such as I/O devices. The currently executing code is interrupted and control jumps to a pre-defined handler that is specified in an in-memory table called IDT (Interrupt Descriptor Table). The x86 architecture defines up to 256 interrupt vectors, each of which corresponds to the address of an interrupt handler function that is going to be invoked when the corresponding interrupt is triggered.

It used to be the case that an I/O device interrupts the CPU by sending a signal on a wire connecting itself to the CPU's programmable interrupt controller (PIC). However, modern x86 servers adopt a more flexible interrupt management architecture called *message signaled interrupt* (MSI) and its extension MSI-X. An I/O device issues a message signaled interrupt to a CPU by performing a memory write operation to a special address, which causes a physical interrupt to be sent to the CPU. When a server starts up, the system software

is responsible for allocating the MSI address and MSI data for each I/O device detected in the server. MSI addresses are allocated from the address ranges assigned to the local APICs (LAPICs) and MSI data are the payloads used in the memory write operations that trigger a message signaled interrupts. An interrupt's MSI address specifies the ID of the interrupt's destination CPU core and its MSI data contains the interrupt's vector number and delivery mode.

MSI is compatible with PCIe, which is the dominant I/O interconnect architecture used on Intel x86 servers. Each memory write operation used to trigger an MSI interrupt is a PCIe memory write request which is issued by a PCIe device and which traverses the PCIe hierarchy to the root complex [7]. An x86 server employs a LAPIC for each CPU core, an IOAPIC for each I/O subsystem, and an IOMMU to isolate the PCIe address space from the server's physical memory space. IOAPIC supports an *I/O redirection table* and IOMMU supports an *interrupt remapping table*. Both tables allow the system software to specify the destination ID, trigger mode, and delivery mode for each PCIe device interrupt. The trigger mode of an interrupt specifies whether the interrupt's signal to the CPU is edge-triggered or level-triggered. Possible delivery modes of an interrupt are (1) the *fixed* mode, in which an interrupt is delivered to all CPUs indicated in the destination ID field, (2) the *lowest priority* mode, in which an interrupt is delivered only to the destination CPU that executes at the lowest priority, (3) the *NMI* (Non-Maskable Interrupt) mode, in which an interrupt is delivered to the destination CPU core at the highest priority and cannot be masked.

IOMMU is an important building block of the I/O virtualization technology built into modern x86 servers [6, 20] that ensures that only authorized interrupts from authorized PCIe devices are allowed to enter a system. Each interrupt remapping table (IRT) entry specifies the interrupt information associated with an MSI address, including a source ID field called SID. When the IOMMU's interrupt remapping mechanism is turned on, a field in an MSI address is used to reference an entry in the IRT. An unauthorized MSI interrupt either points to an invalid IRT entry or an valid IRT entry with a mismatched SID, and is thus blocked by the IOMMU [34].

When an MSI interrupt arrives at its destination CPU, the corresponding interrupt handler in the IDT is invoked. Specifically, an x86 CPU maintains two 256-bit bitmaps: the Interrupt Request Register (IRR) and In-Service Register (ISR). The arrival of an interrupt X with the vector v sets the v-th bit of the IRR (i.e., IRR[v]=1). As soon as X's interrupt handler is invoked, IRR[v] is cleared and ISR[v] is set to indicate that X is currently being serviced. When the interrupt handler associated with X completes, it writes to the end-of-interrupt (EOI) register of the corresponding LAPIC to acknowledge interrupt X to the hardware. Typically, the write to EOI does not contain vector information because it implicitly assumes the completion of the currently highest interrupt. The interrupt controller in turn clears the corresponding bit in the ISR, and delivers the highest-priority interrupt among those that are currently pending, if any.

Finally, an x86 CPU core can send an interrupt to another CPU core via a special type of interrupt called an interprocessor interrupt (IPI). Applications of IPI include booting up, waking up or shutting down another CPU core for more power-efficient resource management, and flushing another CPU core's TLB to maintain TLB consistency. When a CPU core sends an IPI, it writes to the Interrupt Command Register (ICR) of its LAPIC a payload consisting of the IPI's parameters (e.g., the delivery mode, trigger mode, interrupt vector, destination ID, priority, etc). A CPU core is able to send an IPI to its own destination ID, thereby triggering a *self IPI*, an interrupt on the sending core.

2.2 Virtual Interrupt

An x86 CPU core is in *host* mode when the hypervisor runs on it and in *guest* mode when a VM runs on it. A CPU core stays in guest mode until any event configured to force a transition into host mode. When transitioning to host mode, the hypervisor takes over, handles the triggering event, and then re-enters guest mode to resume the VM's execution. The transition from guest mode to host mode is called a *VM exit* and the transition from host mode to guest mode is a *VM entry*. The performance overhead of a VM exit/entry lies in the cycles spent in saving and restoring execution contexts and the associated pollution of CPU caches when executing hypervisor code.

VT support [33] in the x86 architecture enables a hypervisor to set a control bit in the VMCS (Virtual Machine Control Structure) called the *external interrupt exiting* (EIE) bit, which specifies whether or not a VM exit is triggered in the event of a hardware interrupt. More concretely, if the EIE bit is cleared, an interrupt arriving at a CPU core with a running VM causes a direct invocation of the interrupt handler address in the VM, without incurring a VM exit. When EIE is set, the interrupt forces a VM exit and is handled by the hypervisor. The VT support of the x86 architecture also supports another control bit in the VMCS called *NMI exiting* bit, which specifies whether an NMI interrupt triggers a VM exit when it is delivered to a CPU core on which a VM is running, or if the NMI is also delivered directly into the VM.

When an interrupt is directly delivered to a VM, the CPU core uses a different Interrupt Descriptor Table (IDT) than the IDT used in host mode. On the other hand, when an interrupt destined for a VM triggers a VM exit and is delivered by the hypervisor, it is the hypervisor's responsibility to convert this interrupt into a virtual interrupt and inject it into the target VM when the VM resumes execution. Note that a VM exit does not always result in a virtual interrupt injection. For example, if a VM exit is caused by an interrupt whose target is not the running VM (e.g., a timer interrupt set up by the

hypervisor), then this interrupt is not converted to a virtual interrupt and no virtual interrupt injection is performed.

KVM injects virtual interrupts into a VM by emulating the LAPIC registers with an in-memory data structure, mimicking a hardware LAPIC by setting up the emulated registers, such as IRR and ISR, prior to resuming the VM. When a VM is resumed, it checks the IRR, and services the highest-priority pending interrupt by looking up the VM's IDT and invoking the corresponding interrupt handler. After the interrupt handler completes, it acknowledges the virtual interrupt by writing to the (emulated) EOI register, which triggers another VM exit to the hypervisor to update the software-emulated IRR and ISR registers. This design has two drawbacks. First, a virtual interrupt could potentially override the service of a direct interrupt with a higher priority. Second, each EOI write incurs a VM exit in addition to the one that originally triggered interrupt delivery.

2.3 Virtual Device

A VM interacts with an I/O device directly if it is an SRIOV device and indirectly through the hypervisor if it is a virtual device. For an SRIOV device deployed on a server, every VM on the server is assigned a *virtual function* of the SRIOV device. When a virtual function on the SRIOV device issues an interrupt, the hypervisor handles the interrupt and then injects the corresponding virtual interrupt into the target VM. Modern hypervisors split virtual device drivers into front-end drivers, which reside in a guest, and back-end drivers, which reside in the hypervisor. When a VM performs an I/O transaction with a virtual device, the hypervisor terminates the transaction at the virtual device's back-end driver and injects a completion interrupt into the requesting VM via an IPI, because a VM and its backend driver typically run on different CPU cores. Asynchronously, the hypervisor performs the requested transaction with the corresponding physical device and handles the completion interrupt from the physical device in the normal way.

The completion interrupts from both SRIOV devices and virtual devices are handled by the hypervisor and are transformed and delivered to their target VMs as virtual interrupts. Moreover, the current mechanism for handling the EOI write of a virtual interrupt requires the involvement of the hypervisor. As a result, each completion interrupt from an I/O device entails at least two VM exits.

2.4 APIC Virtualization

Since one of the major reasons for VM exits is due to the hypervisor maintaining the states of a VM's emulated LAPIC, the recently released Intel CPU feature, APICv, is aimed to address the issue by virtualizing the LAPIC in the processor. In general, APICv virtualizes the interrupt-related states and APIC registers in VMCS. APICv emulates APIC-access so that APIC-read requests no longer cause exits and APIC-write requests are transformed from fault-like VM exits into trap-like VM exits, meaning that the instruction completes before the VM exit and that processor state is updated by the instruction. APICv optimizes the virtual interrupt delivery process by its *posted interrupt* mechanism, which allows the hypervisor to inject virtual interrupts by programming the posted interrupt related data structures of VMCS in *guest mode*. Traditionally, delivering virtual interrupts requires VM exits into host mode because data structures maintained by VMCS are not allowed to be modified in guest mode. However with APICv, there is no such restriction and hypervisor is able to update the VM's interrupt state registers, such as IRR and ISR, while the VM is running.

Specifically, APICv enables delivering virtual interrupts without VM exits by adding two registers as guest interrupt status, the RVI (Requesting Virtual Interrupt) and the SVI (Servicing Virtual Interrupt), and allows them to be updated in *guest mode*. APICv's virtual interrupt, or posted interrupt, is delivered by setting up the 256-bit PIR (Posted Interrupt Request) registers and the ON (Outstanding Notification) bit. The PIR indicates the vector number of the posted interrupt to be delivered and the ON bit shows that there is an posted interrupt pending. The posted interrupt is delivered to the currently running guest-mode VM and the corresponding states of RVI and SVI are updated by the processor without hypervisor involvement. At the end of the posted interrupt handling, APICv's EOI virtualization keeps a 256-bit EOI-Exit bitmap, allowing the hypervisor to enable trap-less EOI write of the corresponding posted interrupt's vector number. Finally, posted interrupts can be configured in the interrupt remapping table so not only virtual interrupts but also external interrupts can directly injected into a guest.

3. Related Work

Interrupts and LAPIC have been identified as the major sources of I/O virtualization overhead, especially pronounced in I/O intensive workloads [13, 15, 18, 19, 25, 27]. To reduce the number of VM exits, hardware vendors are pursuing hardware virtualization support for the APIC, such as Intel's APICv [1], AMD's AVIC [3], and ARM's VGIC [16]. While these techniques may offer an alternative in future hardware generations, DID can achieve the same or better goals of minimizing the VM exits overheads today, without requiring advanced vendor-specific hardware support.

Ole Agesen et al. [14] propose a binary rewriting technique to reduce the number of VM exits. The mechanism dynamically optimizes the VM's code by identifying instruction pairs that cause consecutive VM exits and dynamically translating the guest code to a variant that incurs fewer VM exits. Jailhouse [4] is a partitioning hypervisor that pre-allocates the hardware resources and dedicates them to guest OSes in order to achieve bare-metal performance. However, due to lack of hardware virtualization for all types of physical resources, this approach generally requires heavy guest modifications and loses the benefits of virtualization. On the

	Virtual Interrupt	External Device Interrupt	Timer Interrupt	End-Of-Interrupt	Guest Modification
ELI/ELVIS	Mixed HW/emulated LAPIC	Partially Direct	Indirect	Partially Direct	No/Yes
Jailhouse	Not Support	Direct	Direct	Direct	Yes
APICv	Posted Interrupt	Indirect	Indirect	Direct	No
DID	HW LAPIC	Direct	Direct	Direct	No

Table 1. *Comparison of the interrupt delivering mechanisms between ELI/ELVIS, Jailhouse, APICv, and DID.*

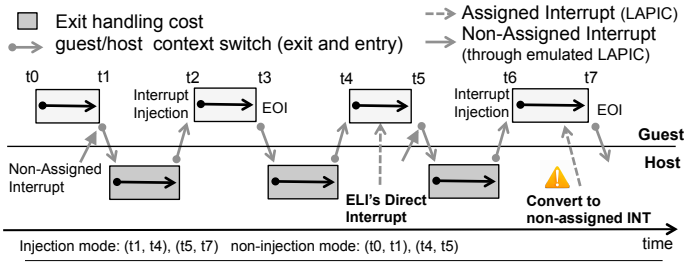

Figure 1. *ELI's mechanism takes effects only at its non-injection mode period, which are between (t0, t1) and (t4, t5), while DID direct delivers all interrupts as long as the CPU is in guest mode.*

other hand, NoHype [22, 29] addresses the VM exit from the perspective of security, because VM exits are the point where control transfers from guest to the host/hypervisor. Unlike NoHype, DID is built for performance rather than security.

ELI and ELVIS are the most well-known software solution for achieving direct interrupt delivery. While ELI directly delivers only SRIOV device interrupts to the VM, DID improves upon ELI by also directly delivering *all* interrupts including timer, virtualized, and paravirtualized device interrupts. ELI solved the mis-delivery problem by using a shadow IDT, modifying the VM's IDT such that all interrupt vectors allocated to the hypervisor and other VMs are made invalid, causing the corresponding interrupts to always force the a VM exit. In the case of paravirtual I/O device interrupts, DID is more general than ELVIS, because it does not require modifications to the guest OS, a major deployment advantage for VMs using closed-sourced OSes and binary OS distributions. Finally, DID leverages the IPI mechanism to inject virtual interrupts into target VMs, thus forcing virtual interrupts to be managed by the HW LAPIC in the same way as directly delivered interrupts. This unifies the delivery mechanisms of virtual and direct interrupts, avoiding priority inversion.

Additionally, the direct interrupt delivery mechanism proposed by ELI/ELVIS takes effects only at its non-injection mode, as illustrated in Figure 1. Specifically, ELI separates interrupt sources to be either assigned interrupts, which is delivered directly, and non-assigned interrupts, which falls back to KVM's virtual interrupt. Non-assigned interrupts must be handled by the emulated LAPIC at the injection mode, which disables the direct delivery. As non-assigned interrupts arrive at $t1$ and until its completion $t4$, the ELI's direct interrupt mechanism is fully off. Even if an assigned interrupt arrives at the injection mode, ($t6$, $t7$), ELI/ELVIS has to convert it to non-assigned interrupt, making the direct interrupt mechanism *partially direct* and the system staying longer handling traditional interrupt injection. We summarize the existing approaches in Table 1 and present the design of DID in the next section.

4. Proposed Direct Interrupt Delivery Scheme

In this section, we describe the mechanisms comprising our Direct Interrupt Delivery approach and prorotype implementation.

4.1 Overview

The main challenge we address to support direct interrupt delivery on x86 servers is in avoiding the *mis-delivery* problem, the problem of delivering an interrupt to an unintended VM. The mis-delivery problem mainly results from the following architectural limitations. First, the x86 server architecture dictates that either every external interrupt causes a VM exit or none of the external interrupts cause a VM exit. This limitation makes it difficult to deliver an interrupt differently depending on whether its target VM is currently running or not. One possible solution is shadow IDT [18, 19]. However, it carried several security issues. Second, the hypervisor is able to inject a virtual interrupt into a VM only when the hypervisor and the VM both run on the same CPU core. For virtual devices, this causes a VM exit for every interrupt from a back-end driver to one of its associated front-end drivers, because these drivers tend to run on different CPU cores. Third, LAPIC timer interrupts do not go through the IOMMU and therefore cannot benefit from the interrupt remapping table. As a result, existing mechanisms for timer interrupt delivery trigger VM exits to the hypervisor. This incurs significant performance overheads as high-resolution timers are used in ever more applications. Moreover, triggering VM exits on timer interrupts increases the variance of the interrupt invocation latency because an additional software layer (i.e., the hypervisor) is involved in the interrupt delivery.

DID leverages the flexibility provided by x2APIC [11] to remove unnecessary VM exits when programming timers and signaling completion of interrupts. With x2APIC, the hypervisor can specify which registers in the LAPIC area can be directly read or written by the VM without triggering a VM exit. Specifically, DID exposes two model-specific register to the VMs, the x2APIC EOI register and

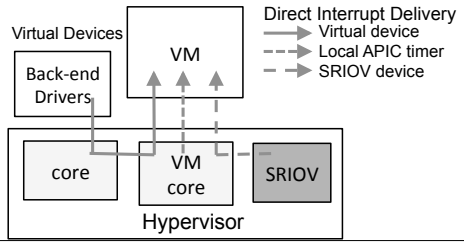

Figure 2. *DID delivers interrupts from SRIOV devices, virtual devices, and timers directly to the target VM.*

the TMICT (Initial Timer Count) register. As a result, a VM can program the LAPIC timer and write to the associated EOI register directly, without incurring a VM exit and associated performance overheads.

In the following subsections, we will go into the details of how DID delivers interrupts from SRIOV devices, virtual devices, and timers directly to their targets, and how to support direct EOI write while preserving the priority among interrupts regardless of how they are delivered.

4.2 SRIOV Device Interrupt

When a VM M is started on a server with an SRIOV device (e.g., a NIC), it is given a virtual function F on the SRIOV device. Once the binding between M and F is established, M can issue memory-mapped I/O instructions directly to F and F can only interrupt M. In DID, when F generates an interrupt, if M is running, this interrupt goes through the PCIe hierarchy, an IOMMU, and eventually reaches the LAPIC of the CPU core on which M is running in guest mode; otherwise, DID arranges to deliver the interrupt to the hypervisor, which then injects a virtual interrupt into M.

To achieve the above behavior, for every VMCS, we clear the EIE bit, so that delivery of an interrupt to a running VM does not cause a VM exit. We also set the NMI exiting bit, so that an NMI interrupt forces a VM exit, even when the EIE bit is cleared. When our DID hypervisor schedules a VM M to run on a CPU core C, it modifies the IOMMU's interrupt remapping table entries assigned to M's virtual functions so that the destination of the interrupts generated by these virtual functions is C. This ensures that every SRIOV device interrupt of M is routed directly to the CPU core assigned to M when M is running. Additionally, when the DID hypervisor deschedules a VM M, it modifies the IOMMU's interrupt remapping table entries assigned to M's virtual functions so that the delivery mode of the interrupts generated by these virtual functions is changed to the NMI mode. This ensures that every SRIOV device interrupt for M causes a VM exit and is delivered to the hypervisor as an NMI interrupt when M is not running. The additional modifications to the interrupt remapping table at the time when the hypervisor schedules and deschedules a VM enable direct delivery of an SRIOV device interrupt only when the interrupt's target VM is running.

When an SRIOV device interrupt is delivered indirectly through the DID hypervisor, the hypervisor runs on the CPU core on which the interrupt's target VM originally ran, rather than on a dedicated CPU core. This allows the processing overhead of our indirectly-delivered interrupts to be uniformly spread across all CPU cores.

In our design, even when a VM M is running on a CPU core C, it is possible that, when a directly-delivered SRIOV device interrupt reaches C, C is in fact in host mode (i.e., the hypervisor is running, rather than M). In this case, the DID hypervisor converts the received interrupt into a virtual interrupt and injects it into M when resuming M's execution.

4.3 Virtual Device Interrupt

To exploit parallelism between physical I/O device operation and VM execution, modern hypervisors, such as KVM, dedicate a thread to each virtual device associated with a VM. Normally, a VM's virtual device thread runs on a different CPU core than the CPU core on which the VM runs. On a DID system, when a virtual device thread delivers a virtual device interrupt I to its associated VM M, the virtual device thread first checks if M is currently running, and, if so, issues an IPI to the CPU core on which M runs with the IPI's interrupt vector set to I's interrupt vector. Because we clear the EIE bit, this IPI is delivered to M without causing a VM exit. The end result is that a virtual device interrupt is directly delivered into its associated VM without a VM exit.

Even though the DID hypervisor tries to deliver a virtual device interrupt to its associated VM only when the VM is running, there is a possible race condition. An IPI-based virtual device interrupt can only be delivered to a CPU core on which its associated VM should be running, but it is possible for the CPU core to be in host mode rather than in guest mode when the interrupt is delivered. In this situation, the hypervisor accepts the IPI on behalf of the associated VM, converts the IPI-based virtual device interrupt into a virtual interrupt and injects it into the associated VM before resuming guest execution.

4.4 Timer Interrupt

For direct delivery of SRIOV device interrupts, we solve mis-delivery problem in DID by taking advantage of the flexibility offered by hardware support for interrupt remapping. In direct delivery of virtual device interrupts, DID solves the mis-delivery problem by making sure that the target VM is running on the target CPU core before sending an IPI to that core. However, on x86 servers, timer interrupts are associated with the LAPIC and do not pass through an interrupt remapping table before reaching their target CPU core. As a result, the hypervisor does not have the flexibility of modifying how a timer interrupt is delivered after it is set up. Consequently, if a timer interrupt is delivered directly, without involving the hypervisor, a timer set up by a VM can be erroneously delivered to the hypervisor, if the target CPU core is in host mode or it can be delivered to the wrong

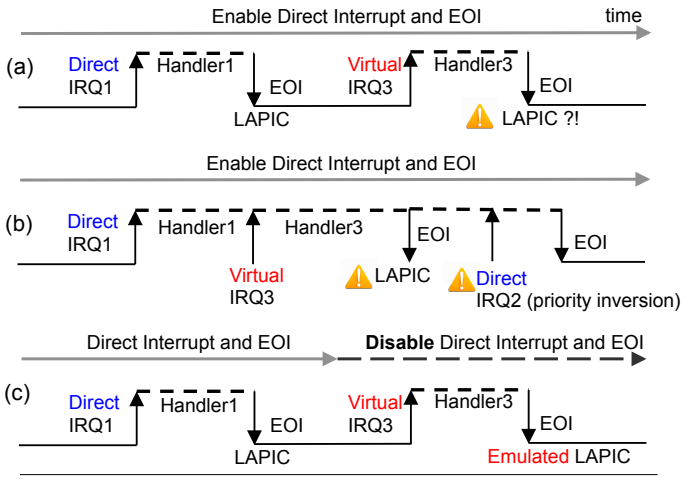

Figure 3. *(a) The LAPIC may receive an EOI write when it thinks there are no pending interrupts. (b) The LAPIC may dispatch an interrupt (IRQ2) when the current interrupt (IRQ1) is not yet done because it receives an EOI write. (c) ELI [18, 19] avoids the confusions caused by direct EOI write by turning off direct interrupt delivery and EOI write whenever at least one virtual interrupt is being handled.*

VM if another VM is running on the target CPU core at the time of timer expiration.

To support direct delivery of timer interrupts while avoiding the mis-delivery problem in DID, we restrict timers set up by the hypervisor to a designated core. Moreover, when the hypervisor schedules a VM M on a CPU core C, the timers that M configured are installed on C's hardware timer; when the hypervisor deschedules a VM M from CPU core C, the timers that M configured are removed from C's hardware timer and installed on the hardware timer of the designated CPU core.

Our design enforces the invariant that, except for the designated CPU core, the only timers installed on a CPU core's hardware timer are set up by the VM currently running on that CPU core. Therefore, this invariant guarantees that no mis-delivery problem is possible when timer interrupts are delivered directly. On the designated CPU core, the DID hypervisor is prepared to service timer interrupts configured by the hypervisor and by those VMs that are not currently running. The timer interrupts destined to non-running VMs are delivered to them as virtual interrupts when those VMs are resumed.

4.5 Direct End-of-Interrupt Write

When an interrupt handler completes servicing an interrupt in DID, it writes to an x2APIC EOI register on the associated LAPIC to acknowledge to the interrupt controller that the service of the current interrupt is finished and the interrupt controller is allowed to deliver the next pending interrupt. The x86 architecture allows our system software to choose whether to trigger a VM exit or not when a VM writes to the associated EOI register. Although it is desirable to avoid a VM exit when a VM writes to the associated EOI register, there may be undesirable side effects if writing to the EOI register does not involve the hypervisor, depending on the mechanism used by the hypervisor to inject virtual interrupts into VMs.

A common way to inject virtual interrupts into a VM, as implemented by KVM, is to properly set up the emulated LAPIC of the VM's VMCS before resuming the VM. However, this emulated LAPIC approach requires EOI writes to trigger VM exits to ensure the consistency in the states of the emulated and physical APICs. If the handler of a virtual interrupt directly writes the EOI, the LAPIC may receive an EOI notification when it thinks there is no pending interrupt, as shown in Figure 3 (a), or may think the currently pending interrupt is already completed when in fact it is still on-going, as shown in Figure 3 (b). Moreover, the LAPIC may incorrectly dispatch a lower-priority interrupt (e.g., IRQ2 in Figure 3) (b) to preempt a higher-priority interrupt (e.g., IRQ1), because the handler for the virtual interrupt IRQ3 writes to the EOI register directly.

The root cause of this priority inversion problem is that virtual interrupts are not visible to the LAPIC when they are injected via software emulation of IRR/ISR. To solve this problem, existing direct interrupt delivery solutions [18, 19, 31] disable direct interrupt delivery and direct EOI writes for a VM whenever the VM is handling any virtual interrupt, as shown in Figure 3 (c) and as called *injection mode* in ELI/ELVIS. Our approach to this problem in DID is different, in that we use a self-IPI to inject a virtual interrupt into a VM. Specifically, before the DID hypervisor resumes a VM, it issues an IPI to its own CPU core. This IPI is then delivered to the injected VM directly after the VM resumes. If multiple virtual interrupts need to be injected into a VM, our DID hypervisor sets up multiple IPIs, each corresponding to one virtual interrupt.

DID's IPI-based virtual interrupt injection mechanism completely eliminates the priority inversion problem due to direct EOI write. When a virtual interrupt is delivered in the form of an IPI, it becomes visible to the target CPU core's LAPIC, enabling it to compete with other direct and virtual interrupts. Because a LAPIC observes every interrupt delivered to its associated CPU core and every EOI write, it allows our system to not mistake an in-service interrupt for being completed when in fact it is not and to not deliver a new interrupt prematurely.

Because DID uses IPIs for direct delivery of virtual interrupts, regular IPIs no longer trigger VM exits in our system. For the original applications of IPIs, such as shutting down CPU cores or flushing remote TLBs, we use special IPIs in DID whose delivery mode is set to NMI. The NMI setting forces a VM exit on the target CPU core, enabling the DID hypervisor to regain control and take proper actions corresponding to the special IPIs.

Regardless of whether or not the DID hypervisor runs on the same CPU core as the VM into which a virtual interrupt is being injected, our DID design uses the same IPI-based mechanism (with proper interrupt vector setting) to deliver the virtual interrupt. There are two key advantages of our IPI-based virtual interrupt delivery mechanism. First, when the source and destination involved in a virtual interrupt delivery run on different CPU cores, no VM exit is needed. Second, because each virtual interrupt takes the form of a hardware interrupt (i.e., IPI) and goes through the target CPU core's LAPIC, the priority among interrupts delivered to a CPU core is correctly preserved no matter how these interrupts are delivered, directly or otherwise.

5. Performance Evaluation

5.1 Evaluation Methodology

To quantify the effectiveness of DID, we measured the reason for and the service time spent in each VM exit using a variety of workloads. We then calculated the time-in-guest (TIG) percentage by summing up the time between each VM entry and VM exit as the total time in guest, and dividing the total time in guest by the total elapsed time.

The hardware testbed used in the evaluation of our DID prototype consists of two Intel x86 servers that are connected back to back with two Intel 10GE 82599 NICs. DID is installed on one of the servers, which is a Supermicro E3 tower server and has an 8-core Intel Xeon 3.4GHz CPU with hardware virtualization (VT-x) support and 8GB memory. The other server acts as a request-generating host, which is equipped with an 8-core Intel i7 3.4GHz CPU and 8GB memory. The server on which DID is installed runs KVM with Intel's VT-d support enabled so that multiple virtual machines could directly access an SRIOV device without interference.

We run Fedora 15 with Linux kernel version 3.6.0-rc4 and qemu-kvm version 1.0 on both servers. We provision each VM with a single vCPU, pinned to a specific core, 1GB memory, one virtual function from the Intel SRIOV NIC, and one paravirtualized network device using virtio and the vhost [10, 26] kernel module.

We boot each VM with the same CPU type setting as the host and enable x2APIC support. The virtual machine started into the graphical user interface mode since the console mode (with -nographic) carried extra performance overhead due to VM exits triggered by MMIOs [23]. We also set idle=poll to prevent a HLT instruction from causing a VM exit. For timer experiments, we enable the kernel parameter "NO_HZ".

We configure all CPU cores to run at their maximum frequency, because the *cyclictest* program tends to report longer latency when the CPU core runs in a power efficient or on-demand mode. For all network experiments, we set the Maximum Transmission Unit (MTU) to its default size of 1500 bytes.

For each configuration, we turn DID on and off to evaluate the benefits of DID. The following benchmark programs are used in this study.

- *WhileLoop*: a loop running for 2^{34} iterations, where each iteration performs one integer addition.
- *Cyclictest*: program for measuring the *interrupt invocation latency* (the average time interval between the moment a hardware interrupt is generated and the moment the corresponding handler in the user-level cyclictest program receives control). We run cyclictest with the highest priority on a dedicated core, measuring 100,000 interrupts at a rate of one per millisecond.
- *PacketGen*: a UDP-based program that sends 128-byte UDP packets to a UDP-based receiver at the rate of 100K, 250K, 400K, and 600K packets per second, where both the sender and receiver programs run at the lowest priority level.
- *NetPIPE* [28]: a ping-pong test to measure the half round-trip time between two machines. In our experiments, we vary the message size from 32 bytes to 1024 bytes.
- *Iperf* [30]: program for measuring the TCP throughput between two machines. We report the average of five 100-second runs.
- *Fio* [2]: single-threaded program performing 4KB random disk reads and writes to a virtual disk backed via virtio by a 1GB ramdisk with cache disabled.
- *DPDK l2fwd* [21]: user-level network device drivers and libraries that support line-rate network packet forwarding.
- *Memcached* [5, 17]: key-value store server. We emulate a twitter-like workload and measure the peak requests served per second (RPS) while maintaining 10ms latency for at least 95% of requests.
- *SIP B2BUA [9]*: a SIP (Session Initiation Protocol) Back-to-Back User Agent server software which maintains complete call states and requests. We use SIPp [8] to establish 100 calls per second with each call lasting 10 seconds.

5.2 Reduction in VM Exit Rate

In the 64-bit Intel x86 architecture with VT-x, there are 56 possible reasons for a VM exit. Each VM exit leads to its corresponding exit handler in the hypervisor and reduces the number of CPU cycles spent in the VM. We identify the most-frequently occurring reasons for triggering a VM exit under I/O-intensive workloads as (1) EXTINT: Arrival of an external interrupt, which includes IPIs sent from the hypervisor's I/O thread and hardware interrupts from SRIOV and para-virtualized devices, (2) PENDVINT: Notification of a pending virtual interrupt to a VM that was previously uninterruptible, (3) MSRWR: Attempt by a VM to write to a

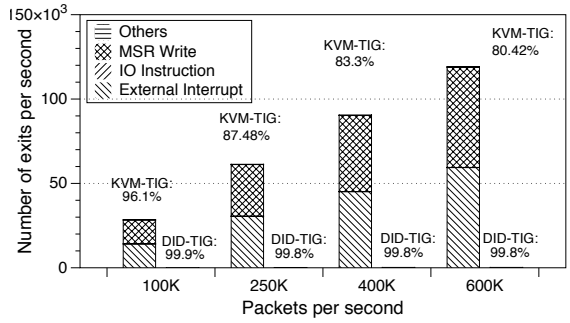

Figure 4. *The breakdown of VM exit reasons for a test VM running on KVM when it is receiving UDP packets through an SRIOV NIC at different rates and when DID is turned on or off*

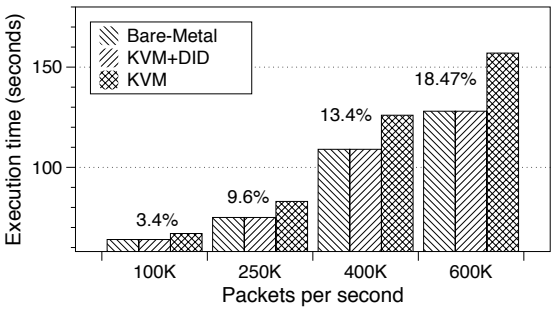

Figure 5. *The execution time of a while-loop program on a bare metal machine, a VM running on KVM, and on KVM with DID, when there is a background load of receiving UDP packets through an SRIOV at different rates.*

model-specific register (MSR) (e.g., programming LAPIC registers and EOI registers), and (4) IOINSR: Attempt by a VM to execute an I/O instruction (e.g., configuring a hardware device).

To assess the effectiveness of DID under a network-intensive workload, we measure the VM exit rate of a running VM when it is receiving UDP packets at different rates. Specifically, we measured vanilla KVM Linux against a system with DID on a test server equipped with an SRIOV NIC. We used a test VM provisioned with a VF on the SRIOV NIC and ran a UDP receiver program in the test VM, collecting the VM exit statistics using the Linux kernel's ftrace facility while a separate program sends UDP packets to the receiver inside the test VM. As shown in Figure 4, when the UDP packet rate at the test VM reaches 100K packets per second, the VM exit rate reaches 28K exits per second, with 96.1% of the time spent in guest mode (TIG). The two dominant reasons for VM exits are external interrupts (EXTINT) and writes to model-specific registers (MSRWR). Because the NIC used in this test supports SRIOV, most external interrupts come from the MSI-X interrupts generated by the VF assigned to the test VM when it receives UDP packets. When using para-virtualized network device, the external interrupt exit is caused by an IPI (inter-processor interrupt) sending from the backend driver, usually the QEMU I/O thread. Additionally, by analyzing the target of each MSR write operation, we conclude that writing to the EOI (End-Of-Interrupt) register accounts for more than 99% of MSR writes. The fact that only 28K VM exits per second are observed when the test VM is receiving 100K packets per second demonstrates that the NIC supports interrupt coalescing. As the packet rate is increased to 250K, 400K, and 600K, the VM exit rate increases to 62K, 90K and 118K, respectively, and the time in guest (TIG) decreases to 87.48%, 83.3%, and 80.42%, respectively. Because the NIC coalesces interrupts more aggressively at higher packet rates, the VM exit rate grows less than linearly with the packet rate.

Figure 4 shows that DID eliminates almost all VM exits due to external interrupts and EOI writes, and reduces the VM exit rate to under 1K per second regardless of the UDP packet rate. With DID, the main reason for VM exits is the I/O instructions (IOINSR) that the guest VM's drivers (i.e., SRIOV VF driver and virtio-net/virtio-blk) use to program the assigned VF's configuration registers, such as the descriptors in the transmission/reception ring buffers. When the test VM is receiving packets at 600K per second, DID saves $(99.8 - 80.42) = 19.38\%$ of the CPU time by avoiding unnecessary VM exits.

5.3 Application-Level CPU Saving

To quantify DID's performance benefits at the application level, we ran the WhileLoop program on a physical machine running Linux (bare metal), on a Linux VM under KVM without DID (vanilla KVM), and on a Linux VM under KVM with DID (KVM+DID). The WhileLoop program does not execute any privileged instruction and thus incurs no VM exit overhead during its execution. At the same time, we ran the UDP receiving program in the background, receiving UDP packets at different rates. Figure 5 shows that for all tested packet rates, the total elapsed time of WhileLoop in the KVM+DID configuration is nearly identical to that of the bare metal configuration. This is because DID eliminates almost all VM exit overheads, allowing the vast majority of the CPU time to be spent in guest mode while executing the WhileLoop program. In contrast, the elapsed time of WhileLoop in the vanilla KVM configuration increases with the UDP packet rate because higher packet rates lead to more VM exit overhead and thus lower TIGs. Accordingly, the WhileLoop performance gains of KVM+DID over vanilla KVM for the tested packet rates are 3.4%, 9.6%, 13.4%, and 18.47%. As shown in Figure 5, the performance gains are closely correlated with the reductions in TIGs that DID enables, 3.8%, 11.42%, 16.6%, and 19.38%, respectively.

5.4 Interrupt Invocation Latency

In addition to reduced VM exit overheads, another major performance benefit of DID is reduction in the interrupt in-

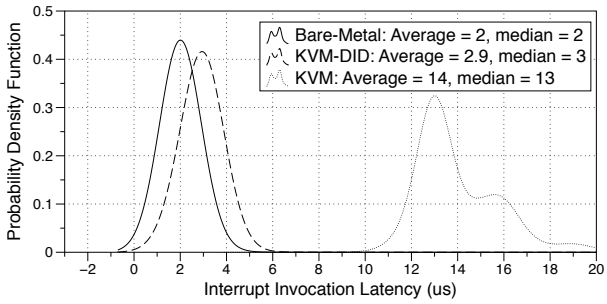

Figure 6. *The probability density function of the interrupt invocation latency of the bare metal, vanilla KVM and KVM+DID configuration during a 100-second cyclictest run*

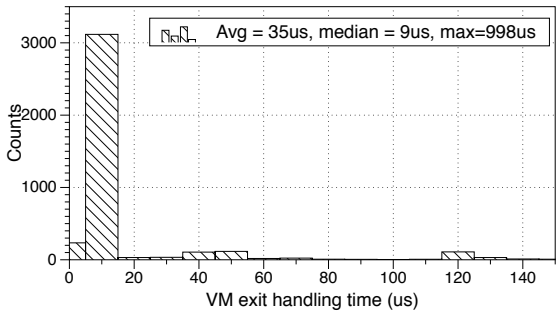

Figure 7. *Service time distribution of the VM exits during which a timer interrupt of a cyclictest program run is delivered*

vocation latency, the time between when a hardware interrupt is generated and when the corresponding interrupt handler starts processing it. Cutting down the interrupt invocation latency is crucial to real-time computing systems because it reduces their worst-case delay bound. DID reduces the interrupt invocation latency by removing the hypervisor from the interrupt delivery path. We used the cyclictest program to evaluate DID's interrupt invocation latency, which in this case is specifically defined as the time difference between when a timer generates an interrupt and when the user-level cyclictest program is invoked to handle it. In the vanilla KVM configuration, where interrupts are delivered indirectly through the hypervisor, factors that affect the interrupt invocation latency are:

1. The hypervisor may temporarily disable interrupt delivery and thus delay the delivery of interrupts from hardware devices to the hypervisor.
2. The hypervisor may introduce additional delays before converting a received interrupt into a virtual interrupt and injecting it into its target VM.
3. A VM's guest OS may disable interrupts and thus delay the delivery of virtual interrupts from the hypervisor to the guest OS.
4. There may be delays in scheduling the cyclictest program after the guest OS handles an incoming virtual interrupt.

In this test, we raised the scheduling priority of the cyclictest program to the highest possible, thus decreasing the variation in the fourth factor above. However, the first three factors are determined by the interrupt mechanisms in the hypervisor and guest OS.

Figure 6 plots the probability density function of the interrupt invocation latency of the bare metal, vanilla KVM, and KVM+DID configuration after running 100,000 timer operations of the cyclictest program. The average interrupt invocation latency of vanilla KVM is $14\mu s$. As expected, this configuration exhibits the highest interrupt latency, because each timer operation in the cyclictest program takes at least three VM exits to set-up the LAPIC timer (specifically TMICT register), receive a timer interrupt, and acknowledge the completion of a timer interrupt. The VM exits are chiefly responsible for the increase in the delay and variability associated with the second factor above.

The average interrupt invocation latency of KVM+DID is $2.9\mu s$, because DID eliminates all VM exits due to TMICT register programming, timer interrupt delivery, and EOI write. Although close to bare metal, the average interrupt invocation latency of KVM+DID is $0.9\mu s$ higher. Although most timer interrupts are delivered directly to the CPU core under DID, it is possible that the target CPU core is in host mode rather than in guest mode at the time of interrupt delivery. When this happens, the hypervisor sets up the self-IPI bitmap to generate a timer interrupt to the target VM when guest execution is resumed. Therefore, the interrupt invocation latency is increased by the amount of time that the hypervisor takes to complete the operation in progress when it receives the interrupt. In our tests, even in an idle VM, there remain approximately 500 VM exits per second, most of which are due to I/O instructions and extended page table (EPT) violations in the VM. The service times for these VM exits account for the small interrupt invocation latency gap between bare metal and KVM+DID.

In a 100-second cyclictest run, there were 100,000 timer interrupts, of which 991VM exits were due to EPT violations with an average VM exit service time of $9.9\mu s$, and 6550 VM exits were due to I/O instructions, with an average VM exit service time of $8.65\mu s$. During this time, only 3,830 timer interrupts were delivered to the target CPU core when it is in host mode or during a VM exit; the service time distribution of these VM exits is shown in Figure 7. 1782 of the 3830 timer interrupts land in VM exits due to EPT violations, with an average VM exit service time of $11.07\mu s$, while the remaining timer interrupts land in VM exits due to I/O instructions, with an average VM exit service time of $24.11\mu s$. As a result, the total contribution of these VM exit service times to the timer interrupt's invocation latency is $84{,}300\mu s$ over the 100-second run. Because the average interrupt invocation latency of the bare metal configuration is $2\mu s$, the average interrupt invocation latency in KVM+DID can be approximated by ((100,000-3,830) * 2 + 84,289) /

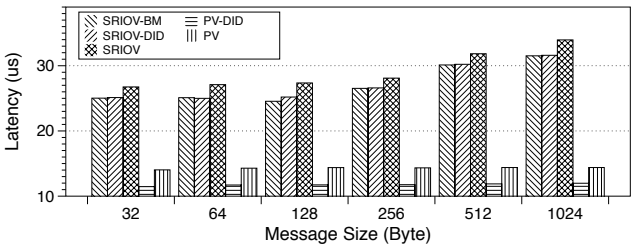

Figure 8. *One-way packet latency between an external server and another server running in the bare metal configuration (SRIOV-BM), the KVM+SRIOV+DID configuration (SRIOV-DID) and the KVM+SRIOV configuration (SRIOV), and between a process running directly on top of a Linux-KVM machine and another process running inside a VM that in turn runs on the same Linux-KVM machine with DID turned on (PV-DID) or off (PV)*

100,000 = 2.76μs, which is close to the measured result, 2.9μs.

5.5 Network Performance Benefits

To measure DID's impact on packet latency, we used the NetPIPE benchmarking tool [28], which employs ping-pong tests to measure the half round-trip time between two servers. Figure 8 shows the latency measurements reported by NetPIPE as we vary the message size from 32 bytes to 1024 bytes. The left-most three bars for each message size correspond to NetPIPE results measured between an external server and another server running in a bare-metal configuration (SRIOV-BM), the KVM+SRIOV+DID configuration (SRIOV-DID), and the KVM+SRIOV configuration (SRIOV). When the SRIOV NIC is used, programming the VFs does not trigger a VM exit. As a result, there is no noticeable difference between the packet latency of the SRIOV-BM configuration and the SRIOV-DID configuration, because the latter does not incur a VM exit on interrupt delivery. In the SRIOV configuration, the only VM exits observed are due to interrupts generated by the VFs and EOI writes, resulting in the average service times for these VM exits being 0.85μs and 1.97μs, respectively. Under the case of SRIOV, we observe two types of VM exits per packet received when executing the NetPIPE benchmark. The first exit is due to the arrival of the external interrupt, indicating the packet arrival, and the second VM exit is due to acknowledgement of interrupt (EOI). The average exit handling time of EOI takes 0.85us, while the eixt handling time of the external interrupt is 1.97us. Consequently, the average packet latency of the SRIOV configuration is higher than that of the SRIOV-BM configuration by approximately 2.44μs, which is comparable to 1.97 + 0.85 = 2.82μs. When the packet size is increasing, the latency also increases as the per-byte overhead starts to dominate the packet latency and accordingly the packet latency increases with the packet size.

The two right-most bars for each message size in Figure 8 correspond to the NetPIPE results measured between a process running on a Linux-KVM host and another process running inside a VM that, in turn, runs on the same Linux-KVM machine with DID turned on (PV-DID) or off (PV). These processes communicate with each other through a para-virtualized front-end driver, a virtio-net back-end driver, and a Linux virtual bridge. In theory, each packet exchange requires three VM exits, one for interrupt delivery, another for EOI write, and a third for updating the back-end device's internal state. In practice, the virtio-net implementation batches the device state updates required by the processing of multiple packets and significantly cuts down the number of VM exits due to I/O instructions, as compared to the number of VM exits caused by EOI writes and interrupt delivery. Consequently, the average packet latency of the PV configuration is higher than that of the PV-DID configuration by at most 2.41μs, which is comparable to the sum of the average service times of the VM exits caused by EOI writes and interrupt delivery. Whereas the packet latencies of SRIOV and SRIOV-DID increase with the message size, the packet latencies of PV and PV-DID are independent of the message size, because the latter does not copy the message's payload when packets are exchanged within the same physical server [10, 26].

To quantify the network throughput benefits of DID, we used the iperf tool [30]. Our results show that, over a 10Gbps link, the iperf throughput of the SRIOV-DID configuration is 9.4Gbps, which is 1.1% better than that of the SRIOV configuration (9.3Gbps), even though the TIG improvement of SRIOV-DID over SRIOV is 16.8%. The CPU-time savings cannot be fully translated into network throughput gain, because the physical network link's raw capacity is nearly saturated. Over an intra-machine connection, the iperf throughput of the PV-DID configuration is 24Gbps, which is 21% better than that of the PV configuration (19.8Gbps), even though the TIG improvement of PV-DID over PV is only 11.8%. The CPU-time savings are more than the network throughput gain, because no payload is actually copied for intra-machine communication and therefore reduction of CPU time does not directly translate to throughput gain.

On the other hand, we also found that DID does not show observable improvement over the DPDK l2fwd benchmark. For DPDK, we set-up the SRIOV NIC and executed DPDK's layer 2 forwarding program, l2fwd, using a VM's VF device. We generate the forwarding traffic from the request generating server to the VM using DPDK's version of Pktgen, and measure the maximum number of received and forwarded packets processed by l2fwd program. Due to the polling nature of DPDK, all network packets are delivered to the l2fwd program via VF device without triggering any interrupt. As a result, either with or without DID, l2fwd shows capable of forwarding 7.9 millions of 128-byte packets per second.

5.6 Block I/O Performance Benefits

To analyze the performance benefits of DID under a high-performance directly-attached disk I/O system, such as an

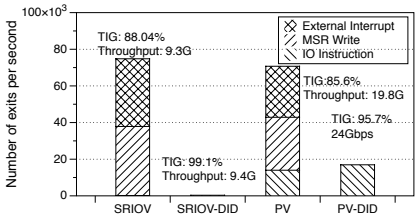

Figure 9. *The iperf throughput improvement of DID when a SRIOV NIC is used for inter-server communication and a paravirtualized NIC is used for intra-server communication*

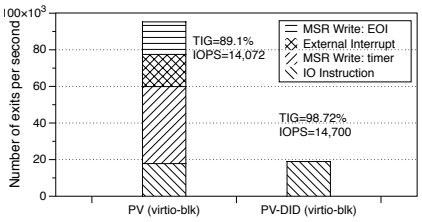

Figure 10. *The VM exit rate and the breakdown of VM exit reasons of the Fio benchmark*

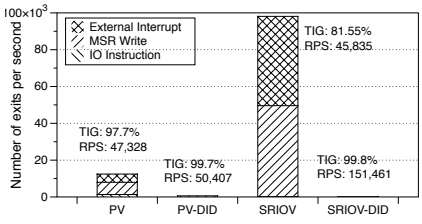

Figure 11. *The VM exit rate and the breakdown of VM exit reasons of a Memcached server under the PV, PV-DID, SRIOV, and SRIOV-DID configurations*

array of solid state disks, we configured a 1GB ramdisk on the host, exposing it to the test VM running on the host using virtio-blk, and ran the Fio benchmark inside the test VM. We measured the IOPS and the I/O completion time, which is the time difference between when Fio issues an I/O request and when that request is completed and returned to Fio. Figure 10 shows that, when DID is turned off, the IOPS is 14K with an average I/O completion time of $34\mu s$. When DID is turned on, the IOPS increases to 14.7K with an average I/O completion time of $32\mu s$. These performance differences again result from the fact that DID eliminates VM exits due to interrupt delivery (EXTINT) and MSRWR writes. As expected, the performance gain of DID is limited by the block I/O rate and thus the associated interrupt rate is generally much lower.

Unlike iperf, where the number of VM exits due to interrupt delivery is approximately the same as the number of MSRWR writes, Fio observes three times the number of VM exits due to MSRWR writes compared to the number of VM exits due to interrupt delivery. Analysis of the Fio benchmark reveals that the program sets up a timer before submitting an I/O request to protect itself from unresponsive disks and clears the timer after each request is completed. Therefore, for every I/O request, three MSRWR writes are needed, one for EOI write, and two for TWICT writes. DID successfully eliminates all VM exits due to these MSR writes.

5.7 Memcached Workload

To evaluate the performance improvement of DID on a popular server workload, we set up a dual-threaded Memcached 600MB server inside a VM (the test VM is configured with one vCPU and 1GB of RAM). We generated a 600MB twitter dataset and warmed up the server by preloading the dataset. We then run a Memcached client simulator that creates eight threads and 200 TCP/IP connections with a get/set ratio of 4:1. To guarantee quality of service in each experiment, we empirically find the peak request rate that allows the server to complete 95% of all requests within 10 msec. We turn off Nagle's algorithm (TCP nodelay option) on both client and server ends.

Figure 11 shows the VM exit rate of the PV, PV-DID, SRIOV, and SRIOV-DID configurations, whose RPS are 47.3K, 50.4K, 45.8K, and 151.5K, respectively. SRIOV-DID outperforms all other configurations by a large margin, because it enjoys the benefits of both SRIOV and DID and removes the majority of VM exits, with a TIG of 99.8%. We compared the performance of the Memcached server on a bare-metal setup of the same hardware, observing a 152.3K RPS, which is only 0.6% higher than SRIOV-DID. The second best setup is PV-DID, with a TIG of 99.7%, followed by the PV configuration, with a TIG of 97.7%. Notably, SRIOV comes in last, with a TIG of 81.55%. Even though SRIOV does not incur any VM exit overhead due to I/O instructions, SRIOV still performs worse than PV, because it incurs a larger number of VM exits due to interrupt delivery and EOI writes than PV. In the PV configuration, the vhost thread periodically polls the physical NIC, batches incoming packets, and then interrupts the front-end driver in the target VM. As a result, the number of packets delivered to a target VM per interrupt is noticeably higher in the PV configuration than in the SRIOV configuration.

One way to achieve the same interrupt aggregation benefit as a polling vhost thread in the PV configuration is to leverage Linux's NAPI facility, which is designed to mitigate the interrupt overhead through polling when the incoming interrupt rate exceeds a certain threshold. To confirm that interrupt rate reduction via polling is the reason behind the inferior performance of SRIOV, we reduced the NAPI threshold of the Linux VM from its default value of 64 down to 32, 16, 8, and 4, essentially increasing the likelihood that the guest's SRIOV VF driver runs in polling mode. When the NAPI threshold is set to 4 or 8, the resulting RPS of the SRIOV configuration rises to 48.3K, improving over the PV configuration. However, the price for lowering the NAPI threshold to 4 or 8 is an increase in CPU utilization by 3% and 6%, respectively. These results confirm that careful tuning can mitigate VM exit overheads of SRIOV in some cases, making them comparable to PV.

In addition to higher CPU utilization, the PV and PV-DID configurations also increase the request latency due to

request batching. Because of the increased request latency, the quality-of-service target cannot be achieved at the same request rate. This explains why, even though the TIG difference between PV-DID and SRIOV-DID is only 0.1%, the RPS of SRIOV-DID is about three times higher than that of PV-DID.

5.8 VoIP Workload

To evaluate the performance benefits of DID in a B2BUA system, we configured the SIPp [8] UAC (User Agent Client) as the call originating endpoint at the request-generating host, the B2BUA server inside a VM at the DID server, and the SIPp UAS (User Agent Server) as the call answering endpoint at the DID server's hypervisor domain. All the SIP messages between UAS and UAC are processed and forwarded by B2BUA's call control logic. Specifically, a call between UAC and UAS is initiated from the UAC by sending an INVITE message to the B2BUA's call control Logic, which performs authentication and authorization. Then, B2BUA forwards the INVITE message to the UAS, the answering endpoint. The UAS receiving the INVITE message will start ringing and sending back an 180 SIP provisional response. As soon as the answering endpoint picks up the phone, an 200 OK SIP message is sent to the originating endpoint and the session is established. Since we set-up 100 calls per second with each call lasting 10 second, the maximum simultaneous call sessions maintained in B2BUA is 1000.

Table 2 shows the call session set-up latency under five configurations. For each experiment, we configured the UAC to make 10,000 calls and measured the call session set-up latency, which is from the UAC sending an INVITE message to the UAC receiving 200 OK message. We observed that although the UAC generates 100 calls per second, the best average call rate we can achieve is 90.9 from the Bare-Metal configuration, and 90.8 from the SRIOV-DID configuration. An important factor affecting the call rate result is the number of retransmitted INVITE messages. PV shows the lowest call rate of 85.5, because it incurs a higher number of INVITE message retransmissions. For session set-up latencies, except the Bare-Metal configuration, SRIOV-DID achieves the best performance with 9061 call set-ups that are completed under 10ms, while PV performs the worst, with 8159 call set-ups that are completed under 10ms and 1335 call set-ups that are completed over 200 ms. The measured VM exit rates for SRIOV, SRIOV-DID, PV, and PV-DID are 4608, 1153, 6815, and 1871. Overall, DID's improvement over SRIOV and PV comes from keeping more CPU time in guest mode by avoiding VM exits and as a result, allowing B2BUA server to process more SIP messages and lower the overall session set-up latency.

5.9 VM Exits Analysis of APIC Virtualization

To analyze the performance benefits of APICv, we set up a server equipped with Intel Xeon E5-2609v2 CPU, 16GB memory and installed Linux kernel 3.14 with the latest support for APICv in KVM. We present the VM exit rates under three types of workloads: the cyclictest workload representing the LAPIC timer interrupts, the Iperf-PV TCP workload for virtual interrupts, and the Iperf-SRIOV TCP workload for external interrupts. Figure 12 shows the result, with each bar from left to right representing vanilla KVM set-up, KVM with APICv enabled, and KVM with DID enabled and APICv disabled.

	<10	10-100	100-200	>200	Call Rate	INVITE Retrans.
Bare-Metal	9485	112	147	256	90.9	79
SRIOV	8342	186	248	1224	86.8	5326
SRIOV-DID	9061	159	242	538	90.8	2440
PV	8159	243	263	1335	75.6	5961
PV-DID	8473	280	61	1186	85.5	4920

Table 2. *Call session set-up latency (ms) distribution of 10,000 calls processed by SIP B2BUA server .*

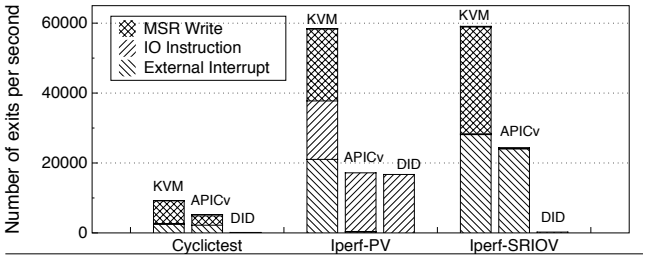

Figure 12. *The VM exit rate and the breakdown of exit reasons under KVM, KVM with APICv support, and DID.*

The cyclictest result shows that the number of MSR Write VM exits associated with APICv is half of that of vanilla KVM. This is because APICv avoids the EOI exit with EOI virtualization, while the rest of the MSR Write exits are caused by programming the timer register (TMICT). In contrast, DID completely eliminate these types of VM exits. For Iperf-PV experiment, APICv gives the same improvement in reducing the number of VM exits as DID. This is because APICv's posted interrupt mechanism enables delivering virtual interrupts from the back-end driver to the VM running core without triggering VM exits, whereas DID achieves the same effect without modifying the guest OS or requiring hardware support. Finally, in the Iperf-SRIOV experiment, APICv shows that although EOI virtualization helps to eliminate the MSR Write exits, external interrupts arriving at the VM running core still trigger VM exits. As a comparison, DID disables the EIE bit in VMCS so that external interrupts do not trigger any VM exit.

6. Discussion

Interrupts are triggered and handled in one of two scenarios. Interrupts are either triggered by direct-passthrough devices configured for VMs or they are triggered by devices configured for the host. When the system is not fully loaded (has spare physical cores available), DID directs interrupts for the

host to the spare physical cores, avoiding interference on the cores where VMs are executing. As a result, interrupts from the host's devices are never delivered to cores which run VMs. However, when the system is oversubscribed, it is possible that interrupts destined for the host arrive at a core which is executing a VM, because the host and VMs are time-sharing a physical core. Under such circumstances, DID configures the host devices to deliver interrupts in NMI-mode. When a device triggers interrupts destined for the host, but this interrupt arrives at a core which is running a VM, the NMI forces a VM exit and passes control to the host. The host's interrupt handler (do_IRQ in Linux) examines the vector number of the interrupt and dispatches the interrupt to the host's interrupt handler based on the host's IDT. Note that configuring an interrupt for NMI mode does not lose the interrupt's original vector number. As a result, when the control is passed to the host, the host is aware of the source of the interrupt.

DID configures NMI not only for hardware interrupts, but also for IPIs triggered by the hypervisor. Because DID uses IPIs to send virtual interrupts directly to the target VM, the host's original use of IPIs, intended for operations such as rescheduling interrupts and TLB shutdown, must use NMI-mode interrupts to force a VM exit. The NMI-mode IPI triggers a VM exit and invokes the host's interrupt handler by using the interrupt's original vector number. Note that it is possible for the NMI to arrive at a core already running in the host mode instead of guest mode. Because DID is capable of identifying the source device or core of the interrupt, it can correctly distinguish whether the interrupt is intended for the guest and requires generating a self-IPI, or if the interrupt is intended for the host and requires directly invoking the corresponding interrupt handler.

7. Conclusion

The performance overhead of I/O virtualization stems from VM exits due to I/O instructions and interrupt delivery, which in turn comprise interrupt dispatch and end-of-interrupt (EOI) acknowledgement. Whereas SRIOV is meant to remove VM exits due to I/O instructions, this paper presents DID, a comprehensive solution to the interrupt delivery problem on virtualized servers. DID completely eliminates most of the VM exits due to interrupt dispatches and EOI notification for SRIOV devices, para-virtualized devices, and timers. As a result, to the best of our knowledge, DID represents one of the most efficient, if not the most efficient, interrupt delivery systems published in the literature. DID achieves this feat by leveraging the IOAPIC's interrupt remapping hardware, avoiding mis-delivery of direct interrupts, and employs a self-IPI mechanism to inject virtual interrupts, which enables direct EOI writes without causing priority inversion among interrupts. In addition to improved latency and throughput, DID significantly reduces the interrupt invocation latency, and thus forms a crucial technology building block for *network function virtualization*, which aims to run telecommunication functions and services on virtualized IT infrastructure.

References

[1] Enabling Optimized Interrupt/APIC Virtualization in KVM. KVM Forum 2012.

[2] Fio - Flexible I/O Tester. http://freecode.com/projects/fio.

[3] Introduction of AMD Advanced Virtual Interrupt Controller. XenSummit 2012.

[4] Jailhouse Partitioning Hypervisor. https://github.com/siemens/jailhouse.

[5] Memcached: memory object caching system. http://memcached.org/.

[6] Secure virtual machine architecture reference manual. AMD.

[7] Single-Root I/O Virtualization and Sharing Specification, Revision 1.0, PCI-SIG.

[8] SIPp: traffic generator for the SIP protocol. http://sipp.sourceforge.net/.

[9] Sippy B2BUA. http://www.b2bua.org/.

[10] virtio- and vhost-net need for speed performance challenges. KVM Forum 2010.

[11] Intel 64 Architecture x2APIC Specification, Intel Corporation, 2008.

[12] Single-Root I/O Virtualization and Sharing Specification, Revision 1.0, PCI-SIG, 2008.

[13] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM ASPLOS'06*.

[14] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference*, pages 373–385, 2012.

[15] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

[16] Christoffer Dall and Jason Nieh. Kvm/arm: Experiences building the linux arm hypervisor. 2013.

[17] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. volume 40, pages 37–48. ACM, 2012.

[18] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. *ACM SIGARCH Computer Architecture News*, 40(1):411–422, 2012.

[19] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual i/o system. In *USENIX Annual Technical Conference*, pages 231–242, 2013.

[20] R. Hiremane. Intel Virtualization Technology for Directed I/O (Intel VT-d). *Technology@ Intel Magazine, 2007*.

[21] DPDK Intel. Intel data plane development kit.

[22] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 350–361. ACM, 2010.

[23] Jan Kiszka. Towards linux as a real-time hypervisor. *RTLWS11, 2009*.

[24] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[25] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L Cox, and Scott Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 61–70. ACM, 2009.

[26] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev. 2008*.

[27] Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX Annual Technical Conference*, pages 29–42, 2008.

[28] Quinn O Snell, Armin R Mikler, and John L Gustafson. Netpipe: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6. Washington, DC, USA), 1996.

[29] Jakub Szefer, Eric Keller, Ruby B Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 401–412. ACM, 2011.

[30] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The tcp/udp bandwidth measurement tool. *http://dast.nlanr.net/Projects*, 2005.

[31] Hitachi Tomoki Sekiyama, Yokohama Research Lab. Improvement of real-time performance of kvm.

[32] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. Secure i/o device sharing among virtual machines on multiple hosts. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 108–119. ACM, 2013.

[33] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[34] Rafal Wojtczuk and Joanna Rutkowska. Following the white rabbit: Software attacks against intel vt-d technology.