Virtualization for Baremetal Cloud Computing

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Bare-metal cloud is an emerging cloud service model in which users rent dedicated physical servers from a remote cloud provider to execute their applications natively, without any virtualization overheads inherent in multi-tenant clouds. However, building a bare-metal cloud service management system is technically challenging due to the constraint that no software agent can be installed on the physical servers being rented out. Such a constraint particularly limits the cloud provider's ability to live migrate customers' workloads for system maintenance or monitor application performance to meet service-level agreements. The ITRI HaaS OS , or IHO, is a bare metal cloud service management system that eliminates this constraint by installing on each physical server a thin single-VM hypervisor that combines the benefits of bare-metal execution and virtualization – customer workloads have complete and direct access to all the hardware devices while the cloud operator retains the serviceability and manageability of the physical machine. We describe a prototype implementation of IHO that provides direct device access to the customer's VM, direct delivery of timer and network interrupts without VM Exits, and supports live migration.

1 Introduction

Infrastructure as a service (IaaS), which was popularized by AWS's EC2 service, has evolved and morphed into multiple forms over the last decade. In the beginning, the basic compute unit for IaaS was a *virtual machine*, which represents a slice of a physical machine carved out by a hardware-abstraction-layer hypervisor. Then the basic compute unit could also be a *container*, which is pre-configured with an operating system and corresponds to a piece of a physical machine delimited by that OS. A more recent option for IaaS's basic compute unit is a *function*, which comes with a complete operating environment constsing of an OS and a middleware layer, and is created on demand. Lately, even a physical machine could serve as the basic compute unit. This type of IaaS is known as *bare metal cloud service* or *hardware as a service* (HaaS). In the past three years, we have been developing a HaaS operating system called *ITRI HaaS OS* or IHO. The focus of this paper is on IHO's virtualization support that enhances the manageability and serviceability required of a modern bare metal cloud service.

The HaaS offerings from cloud operators such as IBM (SoftLayer) and Oracle provide a user a physical data center instance (PDCI), which is composed of a set of physical machines connected in a way specified by the user. HaaS users prefer physical machines to virtual machines primarily because they want to make the best of the underlying hardware resources for workloads that do not need the flexibility afforded by virtualization, such as HPC computation, big data analytics or AI training. Other HaaS use cases include that users have a preferred hypervisor or operating system which is not supported by cloud operators, and that users need special hardware for which virtualization is not sufficiently mature, such as ARM SOC-based micro-server and GPU/FPGA cluster.

In the case of IaaS, cloud operators own and manage the physical machines. In contrast, for HaaS, cloud operators own the physical machines but users manage them. This way, HaaS users are still able to enjoy the multiplexing benefits of cloud computing that are due to sharing of hardware and facilities. A HaaS user or tenant makes a HaaS service request to IHO by specifying a PDCI, which consists of

- A set of physical servers, each with its CPU/memory/PCIe device specification, and configurations on the BIOS, BMC, and PCI devices,
- A set of storage volumes that exist in local or shared storage, and are attached to the servers,
- A set of IP subnets that describe how the servers are connected with one another and to the Internet, and
- A set of public IP addresses to be bound to some of the servers facing the Internet, and their firewall policies.

IHO processes each PDCI request by first making corresponding allocations for server, network and storage resources, and

then setting up the PDCI's required network connectivity. Because a HaaS operator cannot install any agent software on the physical servers rented out on a PDCI, the only way for IHO to programmatically build a virtual network that meets a PDCI's IP subset specification is to leverage the VLAN and VXLAN capabilities in modern network switches and routers by properly configuring them according to the network connectivity specification. Moreover, IHO allows a HaaS tenant to *remotely* check, configure, and update the firmware on the physical servers, as well as install the desired operating systems and applications on them, in a way that minimizes human errors and the adverse side effects that come with these errors. Finally, IHO enables a HaaS tenant to monitor the hardware status of the physical servers and the network traffic among them with full visibility, without revealing anything associated with other co-inhabiting tenants.

For the HaaS use case in which a tenant installs an operating system (Linux or Windows) rather than a proprietary hypervisor on the physical servers of its PDCI, IHO runs a specialized hypervisor on each physical server to enable migration of physical machine state and informed monitoring of application-level performance. This hypervisor, conceptually an extension of a physical server's BIOS, allows only one VM to run on each server and enables the VM's operating system to interact with all the server's devices as if it runs directly on a bare metal server. More concretely, the functional requirements of this *single-VM virtualization* hypervisor are

- Direct pass-through of all PCIe devices without involving the hypervisor,
- Direct delivery of interrupts, including PCIe interrupts and local timer interrupts,
- Seamless migration of virtual machine and physical device state, and
- Resource usage of the hypervisor is limited to one CPU core and 100MB of RAM.

Except during VM migration, this hypervisor is not involved in the data-plane processing of any I/O operations, and thus imposes a negligible I/O performance overhead at run time. With such a hypervisor installed on each physical server, IHO is now able to monitor each server's internal user activities using VM introspection, and seamlessly migrate the user state on the server from one physical machine to another.

2 Related Work

2.1 Jailhouse

By far, Xen and KVM are the two Linux de facto hypervisors. The are versatile and cooperate with QEMU to virtualize the entire system. They utilize the modern hardware-assisted virtualization and match the bare-metal CPU, memory and I/O performance with the significantly reduced CPU overhead. Nonetheless, there are some areas that these two general-purpose solutions need further improvements. One such area deals with the real time applications. A real-time application needs to meet the minimal response time and/or the worst latency. For example, the vehicular break system needs to meet the minimal response time , when the driver presses the break. Failing the requirement leads to a detrimental consequence.

The jailhouse hypervisor is a partitioning hypervisor that runs on the bare-metal and works closely with the Linux [6,7]. It is not trying to be the full-fledged KVM. Its responsibility is to partition and assign the available hardware resource to the guests and prevent a guest from interfering with the jailhouse or another guest. Each guest has its own set of dedicated hardware resource and do not share them. In the other words, there is no overcommitment of resources. Moreover, the jailhouse is an example of asymmetric multiprocessing design, which treats one processor core differently from another. For example, one processor can access the hard disk, while another accesses the serial port. The jailhouse uses this design to create an isolated environment, called a cell. When the jailhouse boots up, it creates a Linux cell or root cell, containing all the processor cores, memory and hardware resources. Before the jailhouse boots up a guest, it creates a new cell and allocate the requested hardware resources to the guest or inmate. This set of hardware resources is dedicated to the guest. This partitioning design indicates the jailhouse does not emulate the devices or manage resources for the guests.

To passthrough the devices, the jailhouse requires the hardware-assisted virtualization. On the x86, it is the VT-x and VT-d. For the LAPIC registers, the jailhouse handles the accesses differently depending on whether the host supports the xAPIC or x2APIC mode. If the host only supports the xAPIC mode, the jailhouse traps all the guest's accesses to the LAPIC registers. Even if the guest would like to access the LAPIC using the MSR interface in the x2APIC mode, the jailhouse traps and emulates it on the top of xAPIC mode. If the host supports the x2APIC, the jailhouse only traps the access to the interrupt command register. ICR is used to send IPIs to other process cores. The trap is required, so the jailhouse can prevent the malicious guest from disturbing other guests. In terms of the interrupt handling, the external interrupts are delivered directly to the guest, which handles them through the guest IDT. One exception is the non-maskable interrupt. The jailhouse uses NMI to regain the controls of guest CPUs.

Thus, the interrupts from the assigned devices and timer interrupts are delivered directly into the guest without any indirection, while the guest can have a fully control over its assigned devices and LAPIC timer. These features help to meet the real-time application requirement for the latency and response time or the long-running computations. Furthermore, the jailhouse confines the guest in its own cell and results in security enhancement and no resource overcommitment.

2.2 Exitless Interrupt

Exitless Interrupt (ELI) is based on the following conditions [1]. First, the guest has its own set of dedicated cores. Second, the guest runs the I/O intensive workload with the directly assigned SR-IOV devices. Third, the number of interrupts that the guest receives from the assigned devices is proportional to the guest execution time. Thus, the ELI delivers the assigned interrupts to the guest directly and non-assigned interrupts to the VMM. This is achieved by the shadow interrupt descriptor table.

When the guest runs in the guest mode, it runs with this shadow IDT prepared by the ELI instead of its own IDT. When the logical processor in the non-root mode receives an external interrupt, it screens the interrupt by the shadow IDT. If the interrupt is from the assigned device, it dereferences the corresponding table entry and invokes the guest's ISR. Otherwise, it traps to the host, which handles the non-assigned interrupts. To make such a distinction, the ELI copies the guest's IDT, including the guest's exceptions and assigned devices, to the shadow IDT. The ELI preserves the device interrupt priorities and keeps the interrupt vector numbers of each device the same between the corresponding guest and host interrupt handlers. It marks guest entries as present and the rest of entries as non-present. Moreover, the ELI configures the logical processor to force exit on the non-present exception. When the host handles the non-present exception, it needs to inspect the exit reason. If it is due to the non-assigned physical interrupt, it converts the exception back to the original interrupt vector and invoke the respective ISR. For the virtual interrupts from the emulated device, the ELI marks them as the non-assigned interrupts. After the trap, the host enters the special injection mode that configures the logical processor to exit on any physical interrupts and the guest to use its own IDT. The host injects the virtual interrupt to the guest.

When the guest's ISR finishes handling its interrupt, it updates the LAPIC EOI register and triggers the VM exit. The VM exit can be disabled through the MSR bitmap, when configuring the VMX module with the x2APIC programming interface. Since the guest does not distinguish between the injected virtual interrupt or the assigned interrupt, it updates the EOI LAPIC register for all cases. This should not be the case for the virtual interrupt from the host emulated device. When the host operates in the special injection mode, it traps the EOI write to the host. Once the guest finishes all the pending EOI writes for the virtual interrupts, the host leaves the special injection mode.

Thus, the ELI delivers the interrupts from the assigned devices directly and virtual interrupts indirectly, while preserving the interrupt priorities. It effectively reduces the VM exits due to the assigned interrupts to 0 and handles the EOI signals properly.

2.3 Direct Interrupt Delivery

Direct Interrupt Delivery (DID) solves the two challenges [8]. First, the interrupts are directly delivered to the guest without the hypervisor intervention on the delivery path. Second, the guest completes the end of interrupts without the VM exit. The two challenges are divided into the following sub-tasks. First, if a guest is running, the interrupts from the SR-IOV devices, timers and emulated devices are delivered directly. Second, when the target guest is not running, its interrupts are delivered through the hypervisor. Third, the priority of physical and virtual interrupts are preserved. Fourth, the number of interrupts that the host needs to complete is zero. In addition, the DID supports the unmodified guests.

The DID routes the interrupts to the guest or host appropriately by configuring the interrupt routing and remapping table from the IOAPIC and IOMMU respectively. For the SR-IOV device interrupts, the DID disables the VM-exit due to external interrupts. Consequently, the interrupts are delivered normally through the guest interrupt descriptor table. If the virtual processor is running, the device interrupt is directly delivered. If the virtual processor is rescheduled by the host, the interrupt is delivered to the host through the non-maskable interrupt. The host injects the corresponding virtual interrupt, when the virtual processor is re-scheduled on the logical processor.

On the modern x86 architecture, each processor has its own LAPIC. LAPIC generates timer interrupts, which are not routed by the IOAPIC and IOMMU. Instead, the LAPIC delivers the timer interrupts to its associated processor. After the guest handles the timer interrupt, it sets up the next timer event by configuring the LAPIC timer. This requires the host's help. The DID ensures that the guest's timer interrupt only delivers to guest instead of other user-level processes. The DID installs the software timer on the host's dedicated core. After the host receives the guest's timer interrupt, it delivers the physical timer interrupt through the IPI. The host delivers the timer IPI, only when the virtual processor is active. If the virtual processor is preempted, the host delivers the timer IPI, when the virtual processor is scheduled on another logical processor.

The DID delivers the virtual interrupts as the IPIs, which are treated as the external interrupts. Each QEMU virtual device is represented by a thread and runs on its dedicate core. Before delivering the virtual device interrupt, the device thread need to check if the virtual processor is active. If the virtual processor is running, the thread delivers the virtual device interrupt through the IPI. Since the DID disables the VM-exit due to external interrupts, the guest receives the device interrupt directly. If the vCPU is not running, the host receives the interrupt on the behalf of guest. The host injects it to the appropriate guest as the virtual interrupt, when the virtual-processor is scheduled on the logical processor.

When the ISR completes, it instructs the logical processor

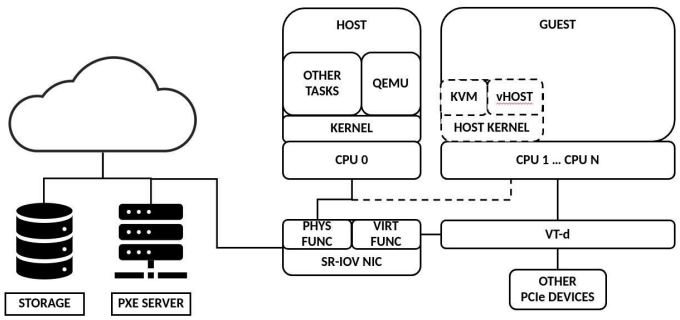

Figure 1: Architecture of IHO

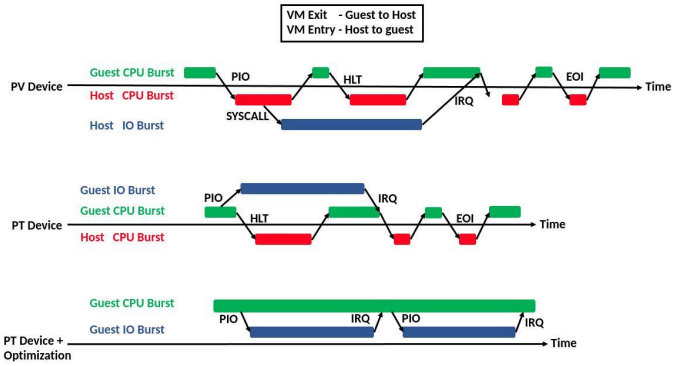

Figure 2: Virtualization overhead

to update the LAPIC end-of-interrupt register. In the DID scheme, the direct EOI write has the following considerations. First, if the handler of virtual interrupts directly updates the EOI register, the LAPIC may thinks there is no pending interrupt. Second, it may also think the pending interrupt is completed, which is still ongoing. Third, LAPIC may dispatch the lower priority interrupt to preempt a higher-priority interrupt. The root cause is that the virtual interrupts are not visible to the hardware LAPIC, when they are injected via IRR and ISR. The DID solves this problem by converting the virtual interrupts as the IPIs, while disabling the VM-exit due to the EOI writes. If there are multiple virtual interrupts, the host issues them as the IPIs one at a time.

Thus, the DID delivers the external interrupts from the assigned devices, timer and emulated devices directly into the guest, while preserving the interrupt priorities. The DID zeroes the number of VM exits due to the interrupts. Moreover, it is not required to modify the guest kernel. This is one of the advantages over ELI and ELVIS.

3 Virtualization support for HaaS

The design goal of IHO is to eliminate the hypervisor from the guest I/O path while achieving bare-metal performance in the guest. To achieve this IHO considers direct device assignment to the guest using Intel's VT-d support. First, the guest meets the bare-metal network and disk I/O performance with direct device assignment. We further apply optimizations to reduce the CPU utilization by hypervisor. Second, the timer interrupts are transformed into the posted interrupts, which are directly delivered to the guest by the logical processor without causing any VM exits. Finally, we present the migration of virtual machine with directly assigned devices.

3.1 CPU Idleness and Processing

One of our design goals is to let the guest stay on its CPU as long as it can. We encounter two different types of virtualization overheads. First, the idle guest issues the privileged HLT instruction. Such an instruction induces the VM exit and transfers the control to the KVM which starts to poll on the CPU until the event arrival. This incurs the high CPU utilization due to the HLT-related VM exit. It is eliminated by updating the primary processor-based VM execution control and disabling the VM exit due to the HLT instruction. It allows the idle guest to stay on its CPU without polling and results in the CPU clock frequency remains at minimal. Thus, disabling HLT-induced VM exit helps to reduce the CPU power consumption of idle guest. Second, the local timer interrupt fires and causes the VM exit, when the guest's time quantum is expired. The longer the guest stays on its CPU, the higher number of physical timer interrupt it receives. To support our objective, our work utilizes the posted-interrupt mechanism and directly deliver the interrupt into the guest without triggering any VM exit. This feature is discussed in the Shared-PID DTID.

3.2 Direct PCI Device Assignment

The hardware-assisted direct device assignment helps the guest achieve the baremetal I/O performance without the additional virtualization overhead using the VT-x and VT-d support. After proper VMCS and EPT configuration, the guest gains the control of assigned device by MMIO/PIO without the help of hypervisor. The VT-x APIC virtualization permits the guest to write to the end-of-interrupt register without a VM exit. With the VT-d support, the guest performs DMA with the enhanced security and eliminates the VM exit overhead due to the device interrupts by using the posted-interrupt mechanism. In addition to the hardware support, VFIO provides the software framework for the userspace device drivers. It works with VT-d and QEMU and sets up the direct PCI device assignment. Although it is expected to move the hypervisor out of the guest I/O path, the hypervisor still induces high CPU utilization due to the HLT emulation. This greatly deviates our goal of guest having its own dedicated cores. Nonetheless, our CPU optimization strategies remedy such a problem.

3.2.1 VFIO

Virtual function I/O is a secure framework for the userspace device drivers. It does not only support the direct PCI device assignment to the userspace processes of VM, but also the platform devices. Why do we need to allow the userspace programs to gain the control of physical devices? For the field of high performance computing, the I/O performances has a great impact on the overall system performance. The performance congestion comes, When the rate of data being read is slower than the rate of data being consumed. Or it happens, when the rate of data being written is slower than the rate of data being computed and produced.

The VFIO needs to fulfill the three requirements of device assignment to a userspace process. First, the userspace driver can access to the device resources such as I/O ports. Second, the userspace driver can perform the DMA securely. This is provided by the IOMMU protection mechanism. Third, the device interrupts is delivered to the device owner in the userspace. The way how the VFIO fulfills the three requirements and applies them with QEMU is briefly described below.

First, the VFIO exposes the configuration registers of physical device as the memory regions. Typically, the device driver communicates with the device registers by the PIO or MMIO operations to the designated memory address space. VFIO would like to expose the address space to the user space. The VFIO retrieves the device information such as BARs from the PCI configuration spaces and IRQ. It reconstructs them as different memory regions in a file. The userspace driver uses the device file descriptor and offset to access each region and retrieve the device information. The VFIO decomposes the physical device to a software interface. Such software interface is turned into the assigned device by QEMU. Essentially, the device read and write handler in the QEMU memory API is forwarded to the VFIO read and write handler. Nonetheless, accessing some parts of PCI configuration requires the KVM/QEMU emulation. PCI configuration space is not handled as memory regions in QEMU. Some of the accesses to the PCI configuration space is passthroughed directly, while others, such as MSI, BARs, and ROM, need to be emulated.

Second, the VFIO programs the IOMMU to transfer the data between the userspace driver and device in a hardware protected manner. The VT-d IOMMU provides the device isolation using the per-device IOVA and paging structure. The virtual virtual address requested by the device is translated to the physical address through the set of paging structures by IOMMU. In the case of VM, the address space of assigned device is embedded within the guest address space. The IOMMU is programmed to translate such IOVA to the host physical address which is mapped to the guest address space. Such translation is both realized and protected by the IOMMU.

Third, the VFIO has a mechanism to describe and register the device interrupt to signal its userspace driver. When the VM accesses the device configuration space, it is trapped through QEMU. QEMU configures the IRQs by the VFIO interrupt ioctls and sets up the event notifiers between the kernel, QEMU and guest. When the kernel signals the IRQ to QEMU, QEMU injects it into the VM. The interrupt signaling is further speeded up by moving QEMU out of the way. KVM supports both ioeventfd and irqfd. ioeventfd registers PIO and MMIO regions to trigger an event notification, when written by the VM. irqfd allows to inject a specific interrupt to the VM by KVM. Once ioeventfd and irqfd are coupled together, the interrupt pathway remains in the host kernel without exiting to the userspace QEMU. Using the VT-d, the KVM and QEMU is completed removed from the signaling path way. It enables the direct interrupt delivery from the assigned device to its VM without a VM exit.

3.2.2 CPU Optimization

To ensure the guest have the bare-metal I/O performance and reduced system CPU utilization, our design utilizes the following optimization.

First, the direct assigned network card and disk drive under the VFIO framework removes the host from the forward I/O path. The guest has the control over the assigned devices and avoids the virtualization overhead, when accessing the device control registers and performing the DMA [4, 5, 9]. After the assigned device services the request, it delivers the interrupt to the guest and triggers the VM exits due to the external interrupts and EOI respectively. Using the VT-d posted-interrupt support and APIC virtualization from VT-x, the guest handles the device interrupt and update the EOI without any VM exit. Thus, the host is completed removed from the guest I/O path for the passthrough devices. Nonetheless, utilizing the hardware posted-interrupt support is not enough to reduce the system CPU utilization, especially when the guest is idle.

Second, the VM exit due to the HLT instruction is disabled. When the guest is idle, it issues the privileged HLT instruction and yields the processor for other processes. It induces the VM exit and transfers the control to KVM. KVM is busy waiting for the incoming events before blocking the virtual processor. This induces the high CPU overhead. There are two solutions. We can either set the polling delay to 0 or disable the VM exit due to the HLT instruction. After setting the delay to 0, the host does not wait in the busy loop but make a trip to the scheduler, which is costly. The latter solution is chosen to further prevent the host from intervening how the guest uses the logical processor. It results in that the idle guest remains on its processor even when it is idle. There is no VM exit due to the HLT instruction. One of side effects of disabling the HLT exiting is that the guest receives the increase number of interrupts from the assigned device. The virtualization overhead of direct interrupt delivery are handled by the posted-interrupt hardware.

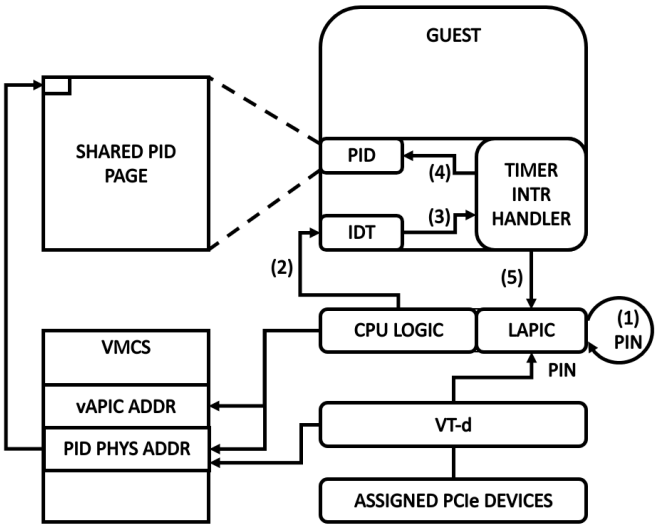

Figure 3: Direct Timer Interrupt Delivery

Third, the guest has a dedicated set of logical processors [1]. This is done by pinning the virtual processors to the isolated logical processors in the one-to-one relationship. Doing so not only prevents the host processes from competing with the guest's virtual processors, but also reduces the degrees of interrupt routing from the assigned devices.

3.3 Direct Interrupt Delivery

Our design uses the hardware-assisted posted interrupt mechanism and achieves the direct interrupt delivery of assigned devices and local timers without the intervention of hypervisor. It reduces the hypervisor CPU utilization and dedicates the CPU time to the guest. Under the normal circumstances, the guest can not handle physical interrupts without the hypervisor. When a physical interrupt is delivered to the guest, it induces the following overheads. First, it triggers the VM exit and transfer the control to the host. Saving and loading the execution context between the root and non-root mode waste the CPU cycles. Second, the host needs to examine and handle the physical interrupt. If the physical interrupt is meant for the guest, the host needs to deliver it as the virtual interrupt upon the next VM entry. Otherwise, the host handles it and schedules the next VM entry. Third, when the guest's interrupt handler finishes, it writes to the EOI register. Such write operation may induce the VM exit. Since the guest is not aware of the distinction between the physical and virtual interrupt, it signals the completion of interrupt in the same way. After the guest handles the virtual interrupt, its EOI update is normally emulated by the host. Fourth, the host may need to use CPU cache and reduce the time to handle the physical interrupt. This introduces the CPU cache pollution.

3.3.1 Posted-Interrupt Mechanism

VT-d supports the posted-interrupt capability and deliver the external interrupts directly from the I/O devices and external controllers without the cost of VM exits and the hypervisor intervention. Before utilizing such feature, the system software needs to define the posted interrupt notification vector. The PIN signifies the incoming external interrupt from the assigned device is subjected to the posted-interrupt processing. The processing is achieved by updating the posted-interrupt descriptor dynamically. When the VMCS is actively used by the logical processor in the non-root mode, it is prohibited to update its data structures. The PID is the exception. Nonetheless, there is one requirement that the PID modifications must be done using locked read-modify-write instructions. Here is another benefit of posted-interrupt support. When the virtual processor is scheduled on another vCPU, the VMM can co-migrate its interrupts from the assigned devices by setting the corresponding bits in posted-interrupt register of PID.

The posted-interrupt support is accomplished in three general steps. First, the VMM programs the interrupt-remapping hardware with the mapping between the external interrupt and virtual interrupt. Second, when the external interrupt is delivered to the interrupt-remapping hardware, it sets the outstanding bit and corresponding bit of virtual interrupt in the posted-interrupt register of PID. It generates the PIN. The IOAPIC delivers the PIN to the appropriate LAPIC. Third, PIN notifies the logical processor that it is the posted-interrupt event. The logical processor starts the posted interrupt processing and delivers the virtual interrupt without any VM exit.

The posted-interrupt processing is described in the following steps. First, when the external interrupt is delivered to the guest's processor, it is acknowledged by the LAPIC. LAPIC provides the processor core the interrupt number. Second, if the physical interrupt is equal to the PIN, the logical processor starts the posted interrupt processing. Third, the processor clears the outstanding notification bit from the posted-interrupt descriptor. Fourth, the processor acknowledges the EOI. Fifth, the processor updates the vIRR by synchronizing it with the PIR. Sixth, the processor acquires the next request virtual interrupt. It updates RVI by the maximum of previous RVI and highest index of bits set in PIR, before it clears PIR. Seventh, the processor evaluates the pending virtual interrupt. Eighth, the processor delivers the virtual interrupt.

3.3.2 Shared-PID DTID

The longer the guest stays on its CPU, the more local timer interrupts it receives. The goal is to let the guest have its dedicated CPUs. Our design does not only directly deliver the timer interrupts to the guest, but also take one step further by allowing the guest update its next timer event directly.

The local timer interrupt is delivered to the guest and results in two scenarios. First, the timer interrupt is meant for

the guest. It induces the VM exit and the control is transferred back to the host. The host handles the timer interrupt and injects the virtual timer interrupt to the guest. When the guest receives the virtual timer interrupt, it services the timer interrupt and set up the next timer event by updating the LAPIC timer initial count register through the x2APIC interface. This triggers the MSR-write VM exit and the control is transfer to the host. The host helps the guest to set up its next timer by registering the `hrtimer` object of guest next timer event. Second, the timer interrupt is not meant for the guest. It induces the VM exit and transfer the control back to the host. The host processes the timer interrupt but does not inject the virtual timer interrupt. Nonetheless, if the timer interrupt is not meant for the guest, the guest should not pay the price.

The first task is to transform the local timer interrupt into the posted interrupt, which is directly delivered to the guest by the VT-d hardware. It requires two actions. The timer-interrupt bit of posted-interrupt request needs to be set, before the posted-interrupt notification is delivered to the guest core. Since the guest is responsible for its own timer interrupt, the guest should set the bit in the PIR. However, such a PIR structure is embedded in the posted-interrupt descriptor and protected by the host. The host needs to share the PIR with the guest by isolating the entire PID to a shared page. If the guest messes up setting the proper bits in the PID through the EPT, it does not affect the host normal operations. In our design, the shared PID page is accessible by three entities: host, guest and virtualization hardware. The second task is to allow the guest control the timer initial count register of LAPIC timer. With the hardware-assisted APIC virtualization, this is achieved by updating the MSR bitmap of VM control structure. The KVM intercept of TMICT MSR update is disabled. When the guest configures the TMICT, the change is written to the register directly without triggering the VM exit. The third task is to configure the LAPIC timer chip to deliver the posted-interrupt notification instead of the actual timer interrupt. In summary, we reach our goal of guest having dedicated CPUs by disabling the HLT- and timer-related VM exits.

Using the shared PID has the draw back. It induces the spurious timer interrupts causing additional interrupt processing in the guest. Since the guest sets the PIR timer-interrupt bit before its next timer event, the it induces the spurious timer interrupts. Such a fake timer interrupt is induced in two cases. First, the guest experiences the spurious timer interrupt when performing I/O-bound activities with the assigned device. Let's take the assigned network card for an example. Both the bits of timer interrupt and network-device interrupt are set in the PIR. Based on the Intel architecture, the timer interrupt has a higher priority than the network interrupt does. The timer interrupt is delivered before the network interrupt. Although the guest should have only processed the network interrupt, it first processes the timer and then network interrupt. Second, upon the VM entry, the PIR is synced to the virtual interrupt-request register because of the KVM implementation. One of time points to evaluate the virtual interrupt delivery is at the VM entry time. If the PIR timer-interrupt bit is present during the copy, the fake virtual timer interrupt is delivered into the guest after the VM entry. If the arrival of virtual timer interrupt is earlier than the expected expiration, the guest ignores it and processes the next interrupt. Thus, the CPU overhead is reduced in comparison with the full timer interrupt processing.

3.4 Seamless VM Migration

The direct device assignment makes it difficult to migrate the guest to its destination [10]. After the VM migration, it is possible that the previously-assigned devices may not be available at the destination. Even if the assigned device is available, the internal state of device may not be readable or still on its way to the destination. The host at the destination has a hard time to passthrough the device without the device-specific knowledge. Moreover, some devices have the unique hardware information that cannot be transferred, such as the MAC address of network interface card. In the case of guest-controlled timer, it depends on the VT-x availability at the destination. Our design takes the approach of alternating the usage of passthrough and respective virtual device with the acceptable service downtime or number of missed time interrupts. We assume that the devices and hardware supports are available. For the network activity, the network traffic is switch from the assigned to virtual network device, before the migration starts. The network traffic is switched back after the guest starts up at the destination. For the local timer interrupt, the direct timer interrupt delivery is switched back to the indirect delivery with the help of `hrtimer` object and TMICT WRMSR VM exit is enabled, before the migration. The changes are reverted after the guest starts up at the destination.

3.4.1 NIC Bonding Between the Assigned and Virtual NIC

In this section we describe the mechanism of migrating a guest with direct NIC assignment. As shown in Figure 4 the guest is configured with virtio network device backed by the vhost driver and a passthrough network interface card [10]. Using SR-IOV [3] the physical NIC is presented as virtual NIC through virtual functions. For the purpose of simplicity, we assume that the guest has one assigned network device. The prototype overcomes the challenge of migrating NIC assigned guest by the following strategy. It uses the Ethernet bonding driver to direct the network traffic between the assigned and virtual NIC. The migration procedure is divided into two parts. During regular operation of guest, the assigned NIC is used for higher network bandwidth. Before the migration, the host uses the bonding driver and shifts the network traffic from the assigned NIC to the virtual NIC. The source

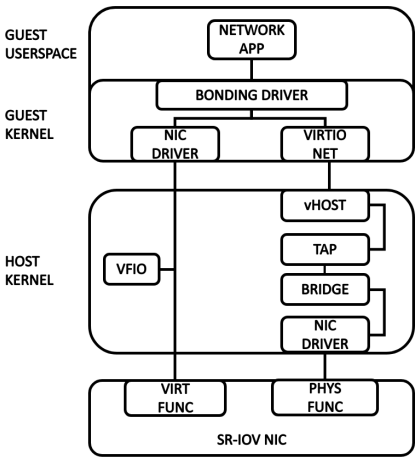

Figure 4: NIC bonding

host takes the control of the assigned NIC through hot unplug event of the assigned NIC and starts the migration. After the guest resumes at the destination, the destination host transfers the control back to the guest using hot plug event of the assigned NIC and switches the network traffic from virtual NIC to assigned NIC.

Linux provides bonding driver to present multiple network interfaces into a single logical interface. The modes of bonding driver define the behavior of the bonded interfaces. To maintain higher network performance during regular operation of guest, we configure the bonding driver in active-backup mode where the assigned NIC is chosen as the active interface and the virtual NIC as a backup-slave interface. In active-backup mode only one of the interfaces is active at any time. When the active interface fails, one of the slave interfaces becomes active. The bonding driver always takes the MAC of the active interface. On failure of the active interface, the bonding driver takes the MAC address of the next to be slave interface. The change in MAC address is notified by broadcasting ARP packets to avoid the network packets loss in guest. Before the migration is initiated, on hot unplug command, the outgoing network traffic is redirected to the virtual NIC interface by the bonding driver and the incoming traffic is shifted to the virtual NIC by broadcasting ARP packets. Once the assigned NIC is hot unplugged, QEMU issues migrate command. After the migration is completed, once the guest resumes on the destination, the NIC device is hot plugged.

We observed that on hot plug event of NIC device on the destination host, the network service in guest drops until for 0.3seconds. Further, we investigate the reason for the network packet loss in the guest. The hot plug mechanism of assigned NIC consists of the following three steps. First, QEMU prepares a software object that represents the passthrough NIC. It then realizes the QEMU software object by getting a copy of configuarion space from the NIC device. Finally, it resets the software NIC object and setup the BAR and interrupt forwarding. The first and last step happen in QEMU main event loop during which the guest remains paused. As a result, the guest experiences downtime during hot plug operation. In IHO , to mitigate the downtime due to hot plug operation on the destination host, the first two steps are executed during migration. During the first phase of pre-copy live migration [2], all the memory pages are transferred to the destination over the network. The dirty memory pages are then transferred in multiple iterative rounds. The VCPU and I/O state of guest are transferred in the final phase to resume the guest on destination. Step one and two are executed on the destination host during migration. Consequently, the network service in guest does not get affected as it runs on the source host. QEMU allows to setup and realize the software object during migration. However, the BAR and interrupts can be setup only after resuming the guest. Hence, we eliminate the downtime caused during the setup of software NIC object phase.

3.4.2 Migration with Shared-PID DTID

The direct timer interrupt delivery depends the following factors. First, the host shares the posted-interrupt descriptor with the guest. When the guest udpates the timer-interrupt bit of posted-interrupt request through EPT, it does not trigger the EPT violations and the local timer interrupt is delivered as the posted interrupt. Second, the guest directly configures the timer initial count register without a WRMSR VM exit. This is achieved by updating the MSR bitmap of VM control structure. Third, the guest needs to correctly compute its next timer events with the host-calibrated LAPIC timer. To compute the next timer event from the nano-seconds to the clock cycles, the multiplication and shift factor of the calibrated timer are required. The host needs to convey such information to the guest. The guest configures the TMICT with the correct timer event in clock cycles.

Before the migration starts, both host and guest need to tear down the DTID. In the host, it enables the TMICT WRMSR VM exit by updating the MSR bitmap of VMCS and configures the LAPIC time to fire the timer interrupt rather than the posted-interrupt notification. It needs to notify the guest to tear down its DTID. Upon receving the notification, the guest requests the host to unmap the shared PID and restores the timer multiplication and shift back in order to correctly compute the timer duration. Such a timer duration is conveyed to the host `hrtimer` object when the host handles TMICT WRMSR VM exit. After the guest starts at the destination, both the guest and host rebuild the DTID. The process of rebuilding the DTID is the reverse of aforementioned steps.

4 Implementation

In addition to the existing supports from the VT-x and VT-d, Linux and its modules, QEMU and KVM, we modify the kernel, KVM and QEMU and reach our goal of baremetal virtual

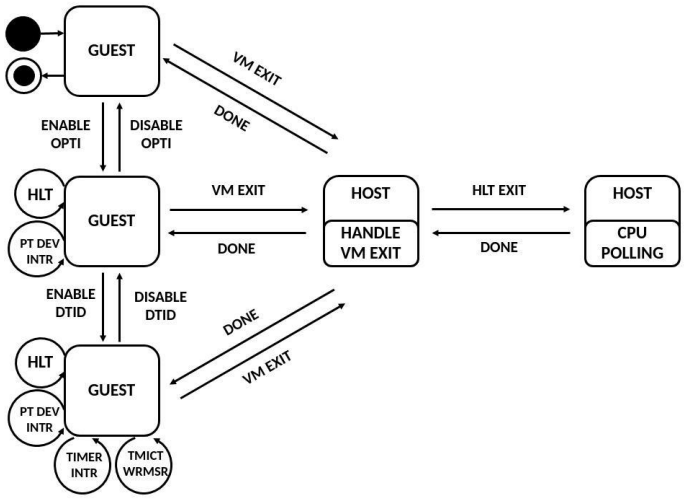

Figure 5: CPU state diagram

machine. The following features are provided. First, the guest has its own dedicated CPU resources and PCIe devices. The guest's cores are isolated, so no other host processes or threads compete the CPU resources with the guest. Each virtual processor is pinned to the isolated core in a one-to-one fashion. It is important to move the host out of the way, when the guest is idle. The VM exit due to the HTL instruction is disabled. Second, the interrupts from the assigned devices and LAPIC timer are handled directly by the guest without the hypervisor intervention. Our approach utilizes the posted-interrupt mechanism to deliver the local timer interrupt. It has the requirement that the guest needs to set the PIR timer-interrupt bit before the arrival of posted-interrupt notification. While the host protects the posted-interrupt descriptor, it shares the only the PID from the associated VMCS. When the DTID is enabled in the host and guest, it introduces spurious timer interrupts. The guest ignores the fake timer interrupt, when the timer interrupt arrives earlier than the expected. Third, the guest updates the LAPIC TMICT directly. This is achieved by updating the MSR bitmap of running guest. As a result, when the guest writes its next period to the LAPIC TMICT, it does not trigger a VM exit and avoids the overhead of interrupt processing and complexity of `hrtimer` subsystem in the host. Fourth, when switching the traffic from the passthrough to the virtual network device, the network service down time is reduced by the Ethernet bonding driver. Furthermore, the network service down time is eliminated when hot plugging the assigned network device to the running guest.

4.1 Disable HLT Exiting and Update the MSR Bitmap

To disable HLT-related VM exit, the HLT-exiting bit of processor-based VM-execution control of VMCS is cleared. Our implementation uses the existing KVM function to set or unset the HLT-exiting bit. The function is `vmcs_write64`. To update the MSR bitmap, we disable the VM exit due to TMICT WRMSR. The function is `vmx_disable_intercept_msr_x2apic`. We uses this function as the prototype and enable the TMICT WRMSR VM exits, while consulting the MSR bitmap from the Intel Software Developer Manual.

4.2 Direct Timer Interrupt Delivery

The core of DTID implementation relies on the following components. First, the PID is shared among the host, guest and VT-d. Before our modification, the PID is embedded within the `vcpu_vmx` structure. If we simply shares the page where the PID is located, we also allows the guest access to the sensitive or privilege information in the host and thus, put the host in a greater risk. Our implementation isolates the PID into its own page and modifies the way how some KVM functions access the PID. The PID is located in the beginning of the page and accessed as a pointer. During the vCPU creation, the PID page is allocated. When KVM sets up the VMCS for each vCPU, the physical address of PID is stored in the field of posted-interrupt descriptor address. As a result, the PID is accessible from both KVM and VT-d. Before the guest access the shared PID page, it needs to tell the host where to map the shared page. The guest issues the hypercall and passes the target guest physical address to the host. Once the host receives the GPA, the host updates the QEMU page table entry and extended page table entry to the physical location of shared page and the reference count of shared page. To have the DTID reversible, the implementation saves the host physical address of target GPA. When the DTID is torn down, our implementation reverts the PTE and EPTE back to the saved HPA and updates the reference counts of shared page. Second, the TMICT WRMSR VM exit is disabled, so the guest is able to update its TMICT without the additional cost. Third, the host needs to inform the gust the multiplication and shift factor of calibrated LAPIC timer. When the guest programs the LAPIC TMICT, it needs the right factor to convert the duration in time to the number of clock cycles. Fourth, we implement the screening algorithm in the guest timer interrupt handlers, `smp_apic_timer_interrupt` and `local_apic_timer_interrupt`. If the timer interrupt arrives than the expected expiration, it is the spurious interrupt. Then, the guest ignores it by skipping the regular processing of timer interrupt. Fifth, the guest updates the PIR timer-interrupt bit, whenever it receives the timer interrupt. From the Intel Software Developer Manual [5], it is the general requirement

to use the locked read-modify-write instruction to access the PID. If the guest uses the atomic test-and-set instruction to set the PIR bit, it prevents the KVM and VT-d from accessing the PID during the time of operation. Such an atomic instruction causes the processor's lock signal which ensures the exclusive usage of shared PID and lock the cache line. Because of the induced spurious interrupts from the guest I/O activities, the guest locks the shared PID so often and prohibits the VT-d to set the proper bit. Thus, the guest I/O performance is degraded. Since the guest only updates the PIR timer-interrupt bit, it is not necessary to lock it. In our implementation, we choose to use the non-atomic operation to set the bit. This effectively reduces the chance of lock contention.

4.3 Migration of guest with passthrough NIC

For the migration of guest with NIC passthrough device, we setup the guest with both virtual NIC and assigned NIC. The two interfaces are enslaved to the bonding driver in active-backup mode. The active-backup mode provides fault-tolerance by switching the network traffic from failed active interface to slave interface. In active-backup mode, the MAC address of bond interface is the same as the MAC address of active interface. On failure of active interface, the MAC address of bond interface has to be changed to the MAC address of the slave interface. On changing the MAC address of the bond interface, the ARP packets have to be broadcast through newly setup bond interface to switch-over the traffic from active to slave interface. The bonding driver provides `fail_over_mac` option to change the MAC address and broadcast the ARP packets on fail-over of active interface. In IHO the bonding driver is configured with `fail_over_mac` set to one. This results in minimum network downtime during hot unplug operation which is performed before initiating the migration process.

The NIC device control is transferred from host to guest (hot plug) using QEMU command `device_add` and the control is transferred back (hot unplug) to host using `device_del`. In IHO the hot unplug operation or `device_del` command is further broken down into three commands `setup_nic`, `realize_vfio_nic` and `reset_nic_device`. `setup_nic` command sets up the software NIC object, `realize_vfio_nic` configures the software NIC object and `reset_nic_device` resets the device by setting the BARs and redirecting the interrrupts. The commands `setup_nic` and `realize_vfio_nic` are executed on the destination host during the migration. The destination host executes `reset_nic_device` command on migration completion.

5 Performance Evaluation

In this section we present our solutions using macro and microbenchmarks.

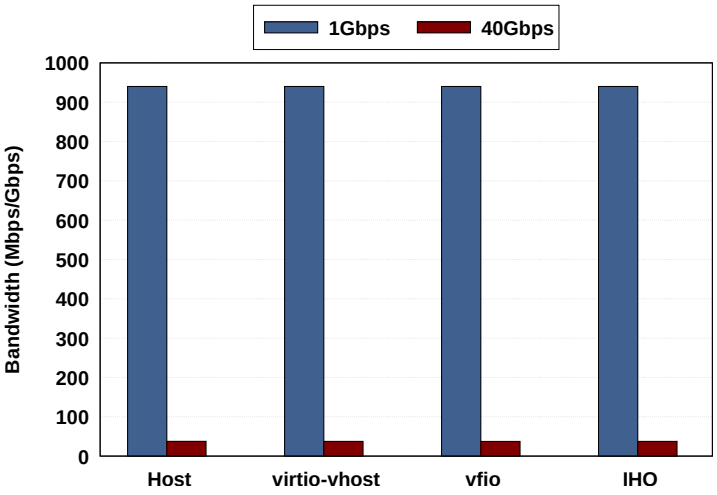

Figure 6: Place holder for iperf performance

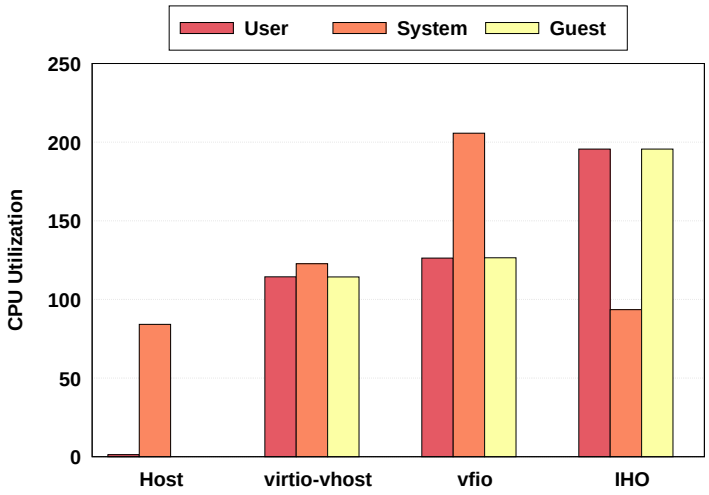

Figure 7: Place holder for iperf CPU utiliztion

5.1 Evaluation Testbed and Methodology

The experiments are run on a 10-core Intel Xeon CPU of 2.2GHz, 32GB memory and 40Gbps Mellanox ConnectX-3 Pro network interface server machine. The guest is configured with –GB memory and –VCPUs. The bonding driver in guest operates in active-backup mode.

5.2 Performance of assigned NIC and CPU utilization

To demonstrate that IHO achieves near bare-metal performance with minimum CPU utilization and minimal hypervisor involvement, we measure the performance using macro benchmark in (a)Host (b)VM with virtio-net interface and (c)VFIO interface. Additionally, we also show that we eliminate the VM exits due to HLT instruction.

VM Exit Reasons	Number of VM Exits	
	Before opti	After opti
HLT		
MSR_WRITE		
EXTERNAL_INTERRUPT		
PREEMPTION_TIMER		
IO_INSTRUCTION		
EXCEPTION_NMI		
PAUSE_INSTRUCTION		
EPT_VIOLATION		
MSR_READ		
CPUID		
TOTAL		

Table 1: Place holder for VM Exit Reasons

We measure the TCP throughput in (a)-(c) using iperf benchmark []. In Figure **??**, we observe that the network bandwidth in host for a 1Gbps and 40Gbps NIC is 940Mbps and 37.4Gbps respectively. With virtio network front-end device and vhost back-end driver, the network bandwidth in guest for 1Gbps and 40Gbps is – and –. The guest achives near bare metal performance using configuration (b) and (c). However, the CPU utilization differs between (b) and (c). With virtio network device backed with vhost driver, the CPU utilization in system mode is – and – in guest mode. With NIC passthrough we observe that the CPU utilization is in system mode is – and – in guest mode. The CPU overhead with passthrough NIC is high compared to virtio network backed by vhost driver. The VCPUs execute HLT instruction when the guest is idle. The VCPUs burn CPU cycles when they poll for sometime before executing the HLT instruction. The VCPU polling mechanism keeps the CPU busy leading to higher CPU utilization.

As explained before, to reduce the CPU utilization in host, we disable the HLT exits by modifying the VMCS structure in KVM. To avoid other system processes to compete with guest, the VCPUs are pinned on isolated CPUs. As shown in **??**, we notice that disabling HLT exits alongwith dedicating cores to guest, reduces the CPU utilization in system mode and increases the CPU Utilization in guest mode indicating that the guest occupies the CPUs for most of the time. In **??** we show that the number of VMExits due to HLT instruction is zero.

5.3 Direct Interrupt Delivery Efficiency

5.4 Seamless Live Migration

Acknowledgments

This work is supported in part by the Industrial Technologies Research Institute, Taiwan, and the National Science Foundation, USA.

Availability

TODO: To be decided wether to include this section, since USENIX gives extra points for open-source papers.

The prototype source code for this work is available at https://github.com/osnetsvn/pci-passthrough.git

References

[1] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Bare-metal performance for virtual machines with exit-less interrupts. *Commun. ACM*, 59(1):108–116, December 2015.

[2] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[3] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, volume 2, 2008.

[4] Intel Corporation. *Intel Virtualization for Directed I/O – Architecture Specification*, November 2017.

[5] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2018.

[6] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. Look Mum, no VM Exits! (Almost). *ArXiv e-prints*, May 2017.

[7] Valentine Sinitsyn. Jailhouse. *Linux J.*, 2015(252), April 2015.

[8] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 1–15, New York, NY, USA, 2015. ACM.

[9] Alex Williamson. An introduction to pci device assignment with vfio. http://www.linux-kvm.org/images/5/54/01x04-Alex_Williamson-An_Introduction_to_PCI_Device_Assignment_with_VFIO.pdf.

[10] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live migration with pass-through device for linux vm. In *Proceedings of the Ottawa Linux Symposium*, pages 261–267, 2008.