Directvisor: Virtualization for Bare-metal Cloud

Kevin Cheng
Binghamton University
New York, USA
tcheng8@binghamton.edu

Spoorti Doddamani
Binghamton University
New York, USA
sdoddam1@binghamton.edu

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

Yongheng Li
Binghamton University
New York, USA
yli241@binghamton.edu

Kartik Gopalan
Binghamton University
New York, USA
kartik@binghamton.edu

Abstract

Bare-metal cloud platforms allow customers to rent remote physical servers and install their preferred operating systems and software to make the best of servers' raw hardware capabilities. However, this quest for bare-metal performance compromises cloud manageability. To avoid overheads, cloud operators cannot install traditional hypervisors that provide common manageability functions such as live migration and introspection. We aim to bridge this gap between performance, isolation, and manageability for bare-metal clouds. Traditional hypervisors are designed to limit and emulate hardware access by virtual machines (VM). In contrast, we propose Directvisor – a hypervisor that maximizes a VM's ability to directly access hardware for near-native performance, yet retains hardware control and manageability. Directvisor goes beyond traditional direct-assigned (pass-through) I/O devices by allowing VMs to directly control and receive hardware timer interrupts and inter-processor interrupts (IPIs) besides eliminating most VM exits. At the same time, Directvisor supports seamless (low-downtime) live migration and introspection for such VMs having direct hardware access.

CCS Concepts • **Software and its engineering → Virtual machines**; **Operating systems**;

Keywords Virtualization, Hypervisor, Virtual Machine, Bare-metal cloud, Live migration

VEE '20, March 17, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7554-2/20/03…$15.00
https://doi.org/10.1145/3381052.3381317

ACM Reference Format:
Kevin Cheng, Spoorti Doddamani, Tzi-Cker Chiueh, Yongheng Li, and Kartik Gopalan. 2020. Directvisor: Virtualization for Bare-metal Cloud. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3381052.3381317

1 Introduction

Conventional multi-tenant cloud services [14, 27, 33] enable customers to rent traditional system virtual machines (VM) [1, 5, 29] to scale up their IT operations to the cloud. However, commodity hypervisors used for virtualizing these platforms suffer from both performance overheads and isolation concerns arising from co-located workloads of other users. To address this concern, cloud operators have begun offering [12, 13] *bare-metal cloud service* which allows customers to rent dedicated remote physical machines. Bare-metal cloud customers are assured stronger isolation than multi-tenant clouds and bare-metal performance for critical workloads such as big data analytics and AI.

However, the quest for native performance and physical isolation compromises cloud manageability. Since cloud operators do not install hypervisors on bare-metal servers, they lose many essential manageability services provided by hypervisors, such as live migration [11, 25], high availability [15], patching [10], and introspection-based security [18, 20, 24, 42, 44]. In contrast, multi-tenant cloud providers compete to differentiate their offerings through rich hypervisor-level services. We aim to bridge this gap between performance, isolation, and manageability for bare-metal clouds.

We propose *Directvisor* to provide the best of both worlds: performance of bare-metal clouds and manageability of virtualized clouds. Directvisor runs one or more DirectVMs which are near-native VMs that directly access dedicated hardware. Traditional hypervisors are designed to limit and emulate hardware access by VMs. In contrast, Directvisor is designed to maximize a VM's ability to directly access hardware for near-native performance while retaining hardware control and manageability. During normal execution, the Directvisor allows a DirectVM to directly interact with processor and device hardware without hypervisor's intervention, as if the VM runs directly on the physical server. However, Directvisor maintains its ability to regain control over a DirectVM when needed, such as for live migration, introspection, high availability, and performance monitoring. Specifically, Directvisor makes the following novel contributions.

(1) Direct Interrupt Processing: Directvisor goes beyond traditional direct-assigned (pass-through) I/O devices to allow VMs to directly control and receive timer interrupts and inter-processor interrupts (IPIs) without interception or emulation by the hypervisor. This is accomplished through a novel use of processor-level support for virtualization [30, 49], directed I/O, and posted interrupts [2, 31]. Direct receiving of timer interrupts and IPIs in a VM greatly reduces the corresponding processing latencies, which is important for latency-critical applications. In contrast, existing approaches [3, 39, 45, 48] focus only on direct processing of device I/O interrupts. Additionally, Directvisor also eliminates the most common VM exits and nested paging overheads, besides ensuring inter-VM isolation and hypervisor transparency. Other than during startup and live migration, the Directvisor is not involved in a DirectVM's normal execution.

(2) Seamless Live Migration and Manageability: Directvisor supports seamless (low-downtime) live migration of a DirectVM by switching the VM from direct hardware access to emulated/para-virtual access at the source machine before migration and re-establishing direct access at the destination after migration. Unlike existing live migration [23, 26, 41, 50–52] approaches for VMs with pass-through I/O access, Directvisor does not require device-specific state capture and migration code, maintains liveness during device switchover, and does not require the hypervisor to trust the guest OS. Additionally, Directvisor supports other manageability functions of traditional clouds, such as VM introspection and checkpointing.

Our Directvisor prototype was implemented by modifying the KVM/QEMU virtualization platform and currently supports Linux guests in DirectVM. The rest of this paper describes the detailed design, implementation, and evaluation of Directvisor's virtualization support for DirectVM.

2 Background

In this section, we provide a brief background on processor support for direct I/O device access, Linux KVM/QEMU [1] virtualization platform, and the associated overheads.

VM Exits: I/O operations and interrupt processing using traditional VMs incur higher overheads than bare-metal execution because privileged I/O operations issued by a guest OS are typically trapped and emulated by the hypervisor. *VM exit* refers to this forced control transfer from the VM to the hypervisor for emulation of privileged operations. VM exits are also triggered by external device interrupts, local timer interrupts, and IPIs. Each VM exit is expensive since it requires saving the VM's execution context upon the exit, emulating the exit reason in the hypervisor, and finally restoring the VM's context before VM entry.

Direct Device Access: Intel VT-d [2] provides processor-level support, called IOMMU [7], for direct and safe access to hardware I/O devices by unprivileged VMs running in non-root mode, which is the processor privilege with which VMs execute. Virtual function I/O (VFIO) [45] is a Linux software framework that enables user-space device drivers to interact with I/O devices directly without involving the Linux kernel. In the KVM/QEMU platform, a VM runs as part of a user-space process called QEMU [6]; specifically, guest VCPUs run as non-root mode threads within QEMU. QEMU uses VFIO to configure a VM to directly access an I/O device without emulation by either KVM or QEMU. In contrast, in a *para-virtual* [47] I/O architecture, the hypervisor emulates a simplified virtual I/O device, called virtio [43], which provides worse I/O performance than VFIO.

Timer Interrupts and IPIs: A CPU may experience two types of interrupts: *external* and *local* interrupts. External interrupts originate from external I/O devices, such as a network card and disk. Local interrupts originate within processor hardware, such as timer interrupts and IPI. A local APIC (Advanced Programmable Interrupt Controller) associated with each CPU core delivers both types of interrupts.

Posted Interrupts: Normally, when a CPU running a VM's virtual CPU (VCPU) receives an interrupt, a VM exit is triggered. The hypervisor then processes the interrupt and, if necessary, emulates the hardware interrupt by delivering virtual interrupts to one or more VMs. The *Posted Interrupt* mechanism [2, 31] is a processor-level hardware support that allows a VM to directly receive external interrupts from directly assigned I/O devices without triggering a VM exit to the hypervisor. In this case, the IOMMU and local APIC hardware convert the external interrupt into a special interrupt, called *Posted Interrupt Notification* (or PIN) vector, which do not cause VM exits. Because the external interrupts "pretend" to be a PIN interrupt, a VM can receive them directly without any VM exits.

3 Directvisor Overview

Figure 1 shows the high-level architecture of Directvisor, which runs as a thin hypervisor on each physical server and partitions the physical hardware among one or more DirectVMs. A small Guest Agent (a self-contained loadable kernel module) coordinates with Directvisor to enable or disable direct hardware access by the guest on demand, such as before or after live migration. We begin by describing a few basic optimizations included in Directvisor which provide the foundations for its more advanced features of direct interrupt delivery and seamless live migration.

Direct I/O Device Access: As with traditional hypervisors, Directvisor uses VT-d and posted-interrupt mechanisms

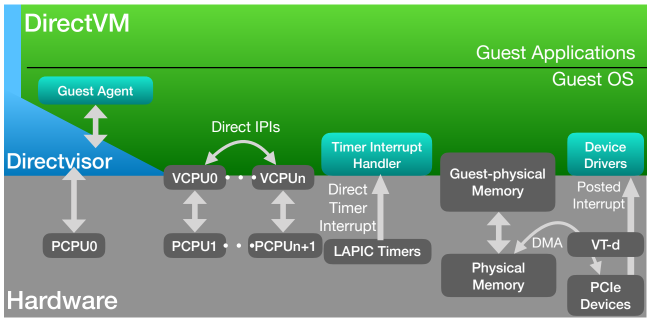


Figure 1. The Directvisor runs as a thin shim layer to provide a DirectVM with dedicated CPU, memory, timer, and I/O access. All interrupts – timers, IPIs, and devices – are directly delivered to the VM without hypervisor's emulation.

to directly assign dedicated (pass-through) I/O devices to a DirectVM and removes itself from both the I/O data path and interrupt delivery path. For physical devices, such as modern network cards, which support multiple virtual functions (VF), Directvisor uses existing mechanisms, such as VFIO, to directly assign a virtual function for each DirectVM.

Dedicated Physical CPUs: Directvisor also assigns a dedicated physical CPU for each virtual CPU of a DirectVM. Doing so not only avoids contention between the hypervisor and guest tasks but also eliminates the need to re-route interrupts from the hypervisor to the guest. Directvisor reserves one physical CPU (CPU 0) for itself where it also handles any external interrupts that are not meant for any DirectVM. For hosting multiple DirectVMs, Directvisor performs NUMA-aware assignment of dedicated physical CPUs among different guests.

Dedicated Physical Memory: Directvisor pre-allocates physical memory for each DirectVM to avoid second-level (EPT) page faults during guest execution. Directvisor also configures a DirectVM to use large pages (via transparent hugepages in KVM/QEMU) so that memory accesses by the guest generate fewer TLB misses due to EPT translations.

Disabling Unnecessary VM Exits: To allow direct control over timer and IPI hardware in the local APIC, Directvisor disables VM exits due to reads and writes by the guest on certain model-specific registers (MSR), specifically the initial count register, divide configuration register, and interrupt command register. Directvisor also implements an optimization to prevent any idle virtual CPUs of a DirectVM from exiting to hypervisor mode. Normally, when a CPU idles, the OS executes a HLT instruction to idle (or halt) the CPU in low-power mode. For a VM's virtual CPU, the HLT instruction would normally trigger a VM exit to the hypervisor, which emulates the HLT instruction by blocking the idle virtual CPU and optionally polling for new events. Besides increasing the latency of reviving the blocked virtual CPU, we observed that frequent HLT-triggered VM exits also increase CPU utilization due to event polling by hypervisor

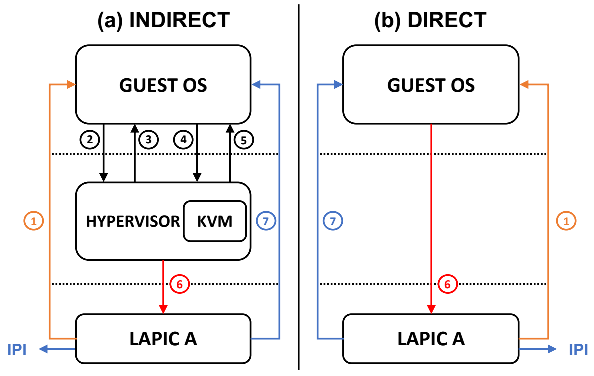


Figure 2. Indirect vs. direct timer interrupts and IPIs.

(see Section 9). Directvisor eliminates both overheads by disabling VM exits when a guest's virtual CPU executes the HLT instruction; thus, the physical CPU directly idles in guest mode without any emulation or polling overheads.

4 Direct Timer Interrupts and IPIs

We next describe how Directvisor enables a VM to directly control and receive local timer interrupts and IPIs. Consider Figure 2 which compares the relative costs for indirect and direct delivery of timer interrupts and IPIs to a VM. Traditional indirect processing of timer interrupts and IPIs is expensive due to several VM exits and entries. A timer interrupt (1) triggers a VM exit (2). The hypervisor injects a virtual timer interrupt into the guest upon VM entry (3). The guest processes the interrupt and attempts to program the next timer interrupt, which triggers another VM exit (4) and VM entry (5). The hypervisor emulates the guest's request by programming the local APIC (6). For IPIs, a guest's request to send an IPI from one virtual CPU to another is still trapped into the hypervisor (4, 5, 6). However, if the processor supports posted interrupts, then the IPI is delivered to the target virtual CPU without any VM exits (7). As shown in Figure 2b, Directvisor completely removes itself from the guest's interrupt processing path by configuring the DirectVM to use the posted interrupt mechanism for both local timers and IPIs. Next we describe these two mechanisms in more detail.

4.1 Direct Timer Interrupts

Figure 3 show how a DirectVM configures the timer in the local APIC to directly receive and handle its timer interrupts. To allow a DirectVM to directly control and receive local timer interrupts, the Guest Agent coordinates with the hypervisor. Directvisor passes the control of timer to the guest through two configuration steps. First, Directvisor configures the local APIC to deliver a posted interrupt notification vector instead of a regular timer interrupt vector when the timer expires (1). Next, the hypervisor executes a VM entry operation and resumes the guest's execution. If the timer fires hereafter, the CPU remains in the guest mode. Second,

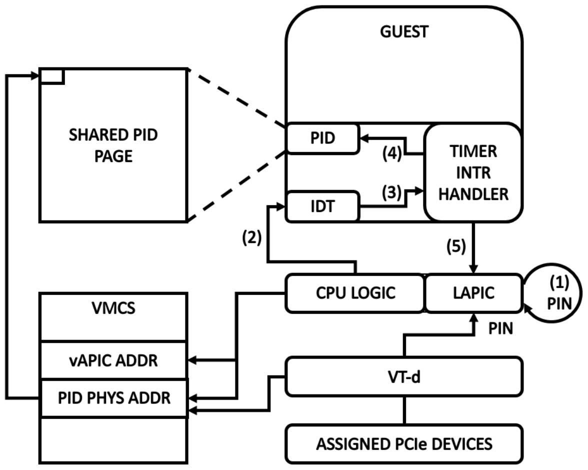


Figure 3. Direct timer interrupts: Local APIC delivers a posted interrupt notification instead of a timer interrupt vector.

Directvisor configures the DirectVM to program the timer directly by disabling VM exits due to the read and write operations to the local APIC's initial count register and divide configuration register (5). Once a timeout value is written to the initial count register, the timer starts ticking. Now, the rest is how to deliver a physical timer interrupt disguised as a guest's virtual timer interrupt without the hypervisor's help. The key is to let the guest use the posted interrupt mechanism. The hypervisor shares a posted interrupt descriptor with the guest. After the guest directly receives and processes a virtual timer interrupt via posted interrupt notification, it marks the timer interrupt bit in the shared posted interrupt request bitmap and programs the next timer value directly in the local APIC registers (2-5).

One side-effect of using posted interrupt notification is that a spurious timer interrupt can be delivered unexpectedly early if there is a VM entry, a posted interrupt from a device, or an IPI. Specifically, such spurious timer interrupts hurt the performance of high-speed and high-bandwidth devices. For instance, considering a guest that uses a pass-through network device, when a network interrupt must be delivered to guest as a posted interrupt notification, it may happen that timer interrupt bits may also be marked as pending in the shared posted interrupt request bitmap. Since both timer and network interrupt vectors are pending, the CPU has to decide whether it should deliver the timer or network interrupt first. Since the timer interrupt has a higher priority than the network interrupt, the CPU chooses to deliver the timer interrupt first. However, the timer interrupt thus delivered is too early, that is, spurious for the guest's actual timer event and must be ignored. Further more, too many spurious timer interrupts can prevent a high performance network interface from achieving peak throughput due to high CPU utilization.

To solve the problem of spurious timer interrupts, a DirectVM guest keeps track of both the arrival and the expected expiration times of each timer interrupt. If the arrival time is earlier than the expected expiration time, the guest acknowledges the timer interrupt and does not proceed with regular timer interrupt handling. Then, it restores the cleared timer interrupt bit in the posted interrupt request bitmap. When the actual timer expires, the corresponding posted interrupt notification triggers the correct timer interrupt in the guest.

4.2 Direct IPIs

Traditional hypervisors trap and emulate the delivery of IPIs between guest virtual CPUs. In contrast, Directvisor allows virtual CPUs within a DirectVM to directly send IPIs to each other without emulation by the hypervisor. This is possible because Directvisor dedicates a physical CPU to each guest virtual CPU, as discussed in Section 3. In addition, each CPU has a per-CPU local APIC identifier, or physical APIC ID, which is necessary to specify the physical CPU to which an IPI must be delivered. Directvisor exports the physical APIC IDs of all physical CPUs assigned to a guest via its Guest Agent. Guest virtual CPUs then use the posted interrupt mechanism to send IPIs to each other without VM exits, similar to the direct timer interrupt mechanism described in Section 4.1. The key difference is that, the hypervisor disables VM exits when a guest virtual CPU sends an IPI by writing to the per-CPU interrupt command register. For example, consider that a virtual CPU A wants to send an IPI to virtual CPU B. Virtual CPU A first sets the desired bit in B's posted interrupt request bitmap. Then A writes the physical APIC ID of B's physical CPU and the posted interrupt notification request to its interrupt command register without incurring a VM exit. Finally, physical CPU A delivers the posted interrupt notification which triggers the IPI delivery on CPU B.

5 Seamless Live Migration

Directvisor supports seamless live migration of DirectVMs, which is one of the key features of server virtualization. Live migration of traditional VMs having pass-through device access is hard for several reasons. First, migration of low-level I/O device state requires the destination node to have an identical instance of the I/O device. Even with identical devices, some of the internal device-specific states may not be readable by the hypervisor, which makes it difficult to restore such state at the destination. Directvisor overcomes these problems by switching a DirectVM from direct hardware access (namely timer, IPI, and device access) to emulated/para-virtual access at the source machine before migration and switching the VM back to direct access at the destination after migration. Unlike existing approaches [23, 26, 41, 50–52] for VMs with pass-through I/O access, Directvisor does not require device-specific state capture and migration code, does not pause guest execution when switching between

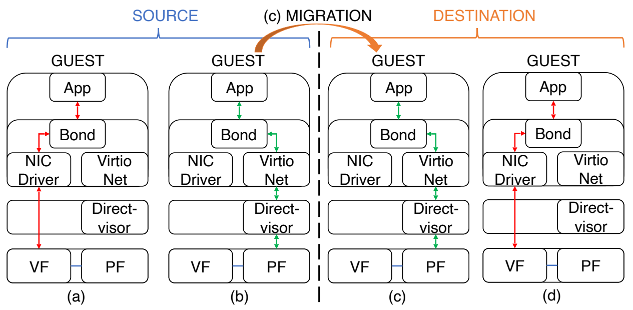


Figure 4. Seamless live migration via bonding of pass-through and para-virtual network interfaces in a guest. From (a) to (b), the guest agent transfers the network traffic from the pass-through to para-virtual NIC before hot-unplugging the NIC. From (c) to (d), the guest agent transfers the network traffic back from para-virtual to hot-plugged NIC.

direct and emulated modes, and does not require the hypervisor to trust the guest OS. Next, we describe the details of network interface migration followed by timer and IPI state migration.

5.1 Pass-through NIC Migration

Consider a bare-metal server equipped with a Single-Root I/O Virtualization [19] (SR-IOV) capable network device, which provides multiple network interfaces – one physical function (PF) and multiple virtual functions (VF) – each of which has its own Ethernet address. Directvisor itself uses the physical function whereas each guest is assigned one virtual function as a pass-through network device. Additionally, each guest is also configured with a para-virtual network interface that is emulated via the Directvisor. The Guest Agent in each DirectVM configures a bonding driver [46] that combines the pass-through and the para-virtual interfaces into one bonded interface which the guest uses for communication.

Figure 4 demonstrates how a DirectVM's network traffic is switched between the pass-through and para-virtual network interfaces before and after the live migration. The bonding driver eliminates the need to have an identical network adapter at the source and the destination. During the normal guest execution, the bonding driver uses the pass-through interface as the active slave and the para-virtual network device as the backup slave. To prepare for live migration, Directvisor asks the Guest Agent to switch the guest's network traffic to the para-virtual device. Directvisor then hot-unplugs the pass-through network interface from the DirectVM as part of temporarily disabling all direct hardware access for the guest. Directvisor then live migrates the DirectVM (via the QEMU manager process) along with the state of its para-virtual network device. Once the migration completes, the DirectVM resumes on the destination server and continues using the para-virtual network device. Concurrently, the Directvisor at the destination hot-plugs a new pass-through virtual function into the DirectVM, instructs the Guest Agent to bond the new pass-through device with the existing para-virtual device, and restores the new pass-through device as the active slave to carry its network traffic.

The most important performance metric for live VM migration is the service disruption time. In a typical live migration [11, 25], a VM experiences a brief downtime when its execution state is transferred from source to destination. In addition, for VMs using pass-through devices, the hot-unplug and hot-plug operations during migration can introduce additional network downtimes. Directvisor eliminates these additional network downtimes by proactively controlling the network traffic switch over between para-virtual and pass-through devices, so that the network liveness is not affected by the hot-plug and hot-unplug operations.

In KVM/QEMU implementation, we also found that the hot-plug operation forces the guest to pause execution, causing significant additional downtime. In particular, the hot-plug operation at the destination consists of three steps. (a) QEMU prepares a software object to represent the pass-through network device. (b) QEMU populates the software object with parameter values extracted from the PCIe configuration space of pass-through device. (c) QEMU resets the software object to set up the base-address registers and interrupt forwarding information. These three steps occur in the QEMU's main event loop which forcibly pauses the running guest. To minimize such a service disruption, we modified QEMU to perform the first and second steps at the destination during the transfer of guest's dirtied pages. QEMU performs the third step after resuming the guest at the destination, thus completely eliminating this additional downtime.

5.2 Timer and IPI Hardware State Migration

Recall that for direct local timer and IPI access during normal operations, Directvisor configures a DirectVM to directly access (i.e., without VM exits) the posted-interrupt request bitmap, the multiplication and shift factors of calibrated clock provided by the hypervisor, the initial count register (to set the timer), and the interrupt command register (to send an IPI). Before live migration, Directvisor disables direct timer and IPI access for the DirectVM by coordinating with the Guest Agent. Specifically, it stops sharing the posted-interrupt descriptor, enables VM exits for updating the initial count and interrupt command registers, configures local vectors to deliver physical timer interrupts and IPIs instead of posted interrupts, and restores the guest's adjusted view of multiplication and shift factors. Once the guest switches to using virtualized timers and IPI, the Directvisor can live migrate the guest to the destination. After the migration completes, Directvisor reenables direct timer and IPI control for DirectVM by restoring direct access to the above state. This switchover between direct and virtualized modes of access is largely transparent to the guest, other than the coordination between the Directvisor and the Guest Agent. Also, the switchover takes negligible time and is performed live as the guest continues running.

6 Isolation and Security Considerations

Directvisor does not trust guests or assume fully cooperative guests. Traditional EPT and IOMMU-based isolation remains unchanged for DirectVM. Directvisor always controls the boot processor and its local APIC. Guests cannot access or modify other guests' local APIC state because physical CPUs are dedicated to individual DirectVMs. Actions performed by Guest Agents are local to the respective guests and do not affect other co-located guests.

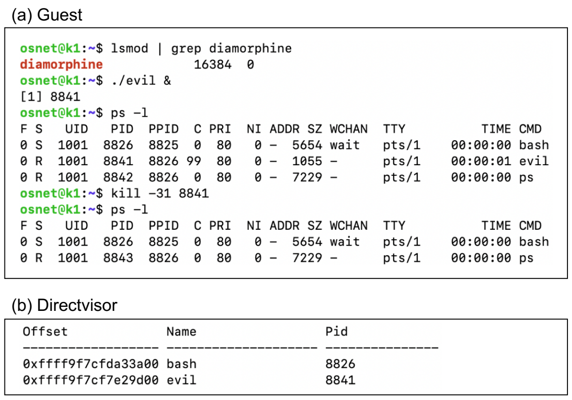
Control over the timers and IPI hardware is important for a hypervisor to maintain its overall control over physical CPUs. Although Directvisor permits a DirectVM to control timer and IPI hardware on its assigned CPUs, it can reclaim control over these resources whenever needed by simply switching the guest to emulated mode. One example is during VM live migration, discussed in Section 5. Before live migration, Directvisor reclaims control over local timers and IPI hardware by enabling previously disabled VM exits and reconfiguring the APIC to deliver the physical interrupts that cause VM exits. If for any reason a Guest Agent becomes non-responsive, the Directvisor can still forcibly switch the corresponding DirectVM to emulated access mode without guest consent.

One potential isolation concern could be if a malicious guest sends unwanted direct IPIs to other guests or to the hypervisor by guessing their physical APIC IDs. This situation can be detected in a couple of ways and handled by turning off direct access for the offending guest. First approach is that the IPI receiver could simply monitor the rate of incoming IPIs and inform the hypervisor of unusual IPI activity via the Guest Agent, which could then temporarily switch all guests to emulated IPI delivery mode to identify the offending guest. A second, potentially simpler, approach is to verify that each received IPI is from a valid sender, that is, one of guest's own virtual CPUs, by recording the sender virtual CPU's identifty in a guest-private memory, which the receiver virtual CPU verifies before acting on an IPI.

7 Other Use Cases

Besides live migration, here we discuss other illustrative use cases that would benefit from Directvisor.

Low-Latency Applications: Real-time applications, which have strict timing constraints, can benefit from Directvisor because a DirectVM processes interrupts with lower latency than traditional VMs. Specifically, the DirectVM (a) directly controls the local APIC hardware (through model-specific registers) without VM exits to schedule timer interrupt and IPI delivery, (b) directly receives all interrupts (device, timers, and IPIs) without VM exits, and (c) remains free of timing jitters due to resource contention by dedicated CPU and memory. In Section 9.1, we show that a DirectVM provides close to bare-metal timer event and IPI processing latencies which are an order of magnitude lower than with a traditional (Vanilla) VM. Likewise, Table 3 shows that round-trip packet latency on a 40Gbps network link is also close to bare-metal latency due to use of posted interrupt delivery.


(a) Guest

```
osnet@k1:~$ lsmod | grep diamorphine
diamorphine            16384  0
osnet@k1:~$ ./evil &
[1] 8841
osnet@k1:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1001  8826  8825  0  80   0 -  5654 wait   pts/1    00:00:00 bash
0 R  1001  8841  8826 99  80   0 -  1055 -      pts/1    00:00:01 evil
0 R  1001  8842  8826  0  80   0 -  7229 -      pts/1    00:00:00 ps
osnet@k1:~$ kill -31 8841
osnet@k1:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1001  8826  8825  0  80   0 -  5654 wait   pts/1    00:00:00 bash
0 R  1001  8843  8826  0  80   0 -  7229 -      pts/1    00:00:00 ps
```

(b) Directvisor

```
Offset             Name                 Pid
------------------ -------------------- ---------------
0xffff9f7cfda33a00 bash                 8826
0xffff9f7cf7e29d00 evil                 8841
```

Figure 5. The Volatility introspection tool in Directvisor can identify a hidden rootkit "evil" in DirectVM.

VM Introspection: VM introspection refers to a hypervisor ability to transparently examine a VM's memory content for manageability functions, such as malware detection and performance monitoring. Here we demonstrate that VM introspection can be performed on a DirectVM. Figure 5(a) shows that we run a malware inside a DirectVM called Diamorphine [37]. The malware hides a target process "evil" which simply burns CPU cycles. Figure 5(b) showes that Directvisor uses the Volatility [22] introspection tool to examine the VM's memory and identified the process "evil" from the guest process list.

VM-based Container Runtimes: VM-based container runtimes such as Kata containers [21], gVisor [34], and Firecracker [40], allow customers to run traditional containers such as Dockers [28] using system VMs on multi-tenant cloud platforms. However, stronger isolation of VMs comes at the expense of giving up bare-metal performance for containers. Although these VM-based container runtimes are optimized to boot quickly, the underlying traditional hypervisors reintroduce all the runtime virtualization overheads, such as VM exits and I/O device emulation. Furthermore, these VM-based runtimes do not support seamless live migration of containers because they also reintroduce many semantic dependencies between the container and the host OS which are difficult to decouple. Directvisor can be used to run such VM-based container runtimes using DirectVMs to achieve bare-metal performance like native containers, while being fully live migratable and manageable like VMs.

8 Implementation-specific Details

Directvisor is implemented by modifying the KVM/QEMU virtualization platform (Linux kernel 4.10.1 and QEMU 2.9.1).

Small Footprint: During runtime, Directvisor maintains a small execution footprint, currently one (shareable) CPU core and around 100MB of RAM, with room for further reductions.

Guest's Code Changes: The special Linux kernel module is installed in an unmodified Linux guest. The module helps to enable or disable the direct timer and IPI delivery. The guest's kernel module has 704 lines of code. Of these, 177 and 193 lines support direct timer and IPI delivery, respectively. The rest deals with managing *HLT*-triggered VM exits, direct writes to the initial count and interrupt command registers, initialization and termination of bonding driver, and other supporting functions.

Virtual-to-Physical CPU Assignment: Directvisor keeps the boot CPU or CPU0 for itself and dedicates the rest to DirectVMs. Each guest virtual CPU is pinned to a dedicated physical CPU core. Directvisor exports the local APIC IDs of a guest's physical CPUs to its Guest Agent, which uses these IDs for direct IPI delivery among guest virtual CPUs.

Disabling *HLT* Exiting and Updating the MSR Bitmap: To disable *HLT*-triggered VM exits, Directvisor clears the *HLT*-exiting bit of processor-based VM-execution control in the VM control structure. Directvisor also disables VM exits due the writes to the initial count register and interrupt command register of the local APIC.

Guest Bonding Driver: The Guest Agent configures the bonding driver to use the pass-through network interface and para-virtual interface as the active slave and backup slave, respectively. Linux's bonding driver provides a fail-over option to change the MAC address of the bonded interface and to broadcast gratuitous ARP packets to advertise its new MAC address when its active slave fails. The Guest Agent configures the bonding driver with this option to cut down the network downtime due to the hot-unplug operations.

9 Performance Evaluation

We evaluate the performance of Directvisor on two physical servers having 10 core Intel Xeon E4 v4 2.2 GHz CPU, 32GB memory, and two network interfaces – a 40Gbps Mellanox ConnectX-3 Pro network card and a 1Gbps Intel Ethernet Connection I217-LM network card. Both the hypervisor and the guest use Linux kernel 4.10.1. Directvisor is implemented by modifying the KVM and QEMU 2.9.1. Directvisor configures the guest with 1–9 virtual CPUs and 28GB of RAM, besides para-virtual and one pass-through network interface. The Guest Agent configures the bonding driver in the active-backup mode with fail-over MAC option.

We used a *Bare-metal server* configuration and five guest configurations to compare the effectiveness of Directvisor.

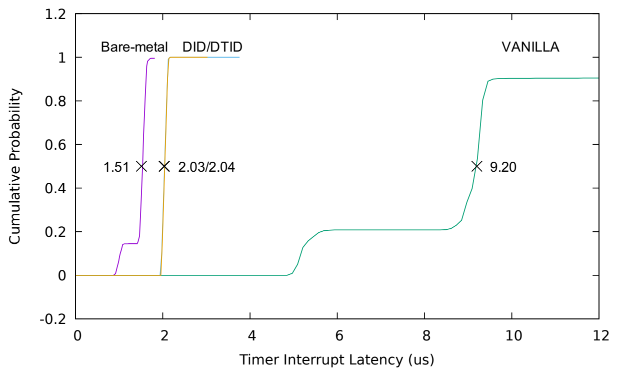


Figure 6. Comparison of timer interrupt latencies.

Table 1 compares the five guest configurations in terms of their VM exit behaviors.

- **Bare-metal**: A machine without virtualization.
- **VHOST guest**: uses a para-virtual network device with VHOST as the backend.
- **VFIO/VANILLA guest**: uses pass-through network device to directly handle network I/O and interrupts.
- **OPTI guest**: VFIO + disabled VM exits due to *HLT* instruction and VMX preemption timer.
- **DTID guest**: OPTI + direct timer interrupt delivery.
- **DID guest**: DTID + direct IPI delivery.

In terms of measurement workloads and tools, we measured network throughput using *iperf 2.0.5* [32], network latency using *ping* [36], and CPU utilization using *atopsar 2.3.0* [4]. We also used Perf 4.10.1 [38] to measure the number of VM exits. To evaluate the direct timer and IPI latencies, we developed an in-kernel Cyclictest benchmark inspired by a similar userspace Cyclictest [16] benchmark. Third, we investigated the performance of CPU, memory and cache first by Princeton Application Repository for Shared-Memory Computers. We used the PARSEC benchmark [8] to evaluate the CPU, memory, and cache performance of various workloads. We used SPEC CPU 2017 [9] benchmark to evaluate CPU-intensive workloads. Finally, we measured the network downtime for our seamless VM live migration by the in-house UDP packet generator and tcpdump [35].

9.1 Direct Interrupt Delivery Efficiency

We first compare both the timer and IPI latency across various guest configurations. The timer interrupt latency is the timing difference between the expected and actual wakeup time when a thread sets a timer. We ran the experiment with a sleep time of 200 μs for 10 million iterations. Unidirectional IPI latency is the difference between the IPI sending and receiving times. To measure IPI latencies, we registered our own special IPI vector in the host or guest kernels. When a CPU received the special IPI, it recorded the timestamp. We sent IPIs between two CPUs with an interval of 200 μs for 10 million iterations.

Guest Configuration \ Optimization	VM Exit Happens When						
	EPT Fault	Access NIC	NIC Interrupt	HLT Instruction	VMX Preemption Timer	Timer Interrupt	Inter-Processor Interrupt
VHOST	Yes	Yes	Yes	Yes	Yes	Yes	Yes
VFIO/VANILLA	*No*	*No*	*No*	Yes	Yes	Yes	Yes
OPTI	*No*	*No*	*No*	*No*	*No*	Yes	Yes
DTID	*No*	*No*	*No*	*No*	*No*	*No*	Yes
DID	*No*	*No*	*No*	*No*	*No*	*No*	*No*

Table 1. Different virtualization configurations for a guest.

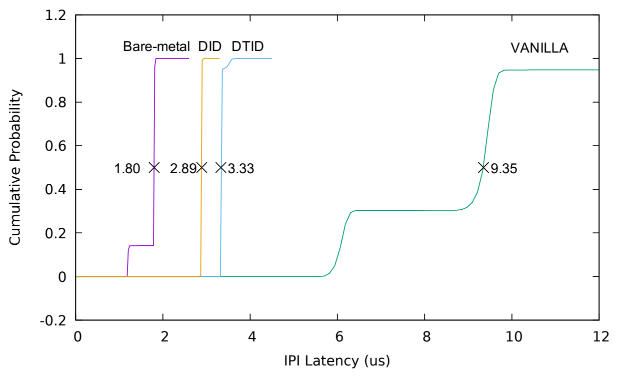


Figure 7. Comparison of timer interrupt latencies.

Figure 6 shows that the median timer-interrupt latency for the Bare-metal, DID, DTID and VANILLA guests were 1.51, 2.03, 2.04 and 9.20 μs, respectively. Both the DTID and DID guests came close to the Bare-metal performance with an additional 0.52 μs latency. Figure 7 shows that the median latency of IPIs for Bare-metal, DID, DTID and VANILLA guest were 1.80, 2.89, 3.33 and 9.35, μs respectively. The DID guest had a 1 μs extra latency than the Bare-metal case.

	Bandwidth (Gbps)	CPU Core	Network Intr. (Hz)	Timer Intr. (Hz)
Bare-metal	37.39	0	118592	256
		1	0	348
DTID Guest	37.35	0	110856	255
		1	0	348

Table 2. Removal of spurious timer interrupts, when the DTID guest performs the network-intensive workload.

Since both the network and timer interrupt delivery use the posted-interrupt mechanism, network interrupts can trigger spurious (early) timer interrupts which are filtered out by the Guest Agent. The effect of spurious timer interrupts is particularly notable when a DTID guest ran network intensive workload over the InfiniBand link. In our experiment, the CPU 0 processed both the network and timer interrupts whereas the CPU 1 ran iperf [32] threads. The timers fired at a frequency of 250Hz. In Table 2, the interrupt profile of DTID guest matched the Bare-metal's profile. If the DTID guest did not use the filtering mechanism, its CPU 0 would have experienced an additional 100,000 or more timer interrupts. Although the DTID guest incurred an extra filtering overhead for every network interrupt, this overhead had negligible impact on its network throughput.

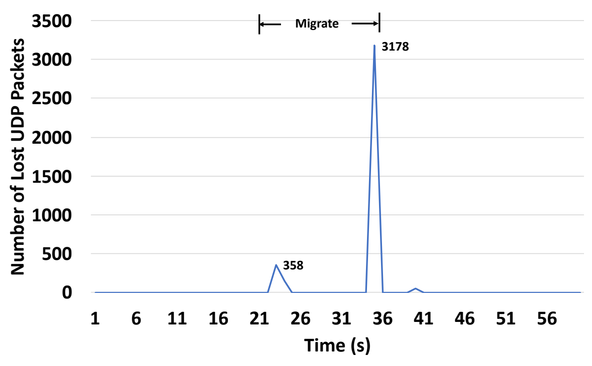
9.2 Live Migration Performance

Next, we show that Directvisor can live migrate a DirectVM while maintaining liveness comparable to traditional pre-copy migration for regular VMs. Figure 8 compares the number of packets lost for four guest configurations where the guest receives 1KB UDP packets at the rate of around 30K packets/sec while the guest was being migrated from a source machine to a destination machine over the 1Gbps interface. We chose 30K packets/sec because higher incoming packet rates adversly interfered with VM migration; both para-virtual network interface and live migration are managed by the same QEMU process, causing CPU contention within QEMU. The four guest configurations being compared are as follows:

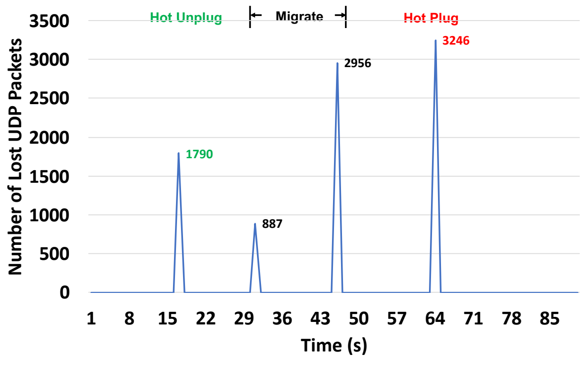
- *Vhost-only:* in which the guest uses only a para-virtual (Vhost) network interface.
- *Reactive Switchover:* in which guest uses a bonding driver. The active pass-through slave is hot-unplugged from the guest at the source machine to reactively trigger the bonding driver to switch traffic to its para-virtual slave interface. Likewise, the pass-through slave is hot-plugged back at destination to reactively trigger a switch back at the destination.
- *Managed Switchover:* in which the incoming traffic is proactively switched over at source before the hot-unplug operation and switched back at destination after the hot-plug operation.
- *Optimized Switchover:* which improves upon Managed Switch by overlapping part of the hot-plug operation at the destination with memory transfer from the source.

Figure 8a for the Vhost-only configuration shows that, for traditional live migration, packets are lost mainly during the stop-and-copy phase when the VM's execution state is transferred from source to destination, besides minor losses during the preparation at source node. Our goal is to match this liveness when migrating a DirectVM.

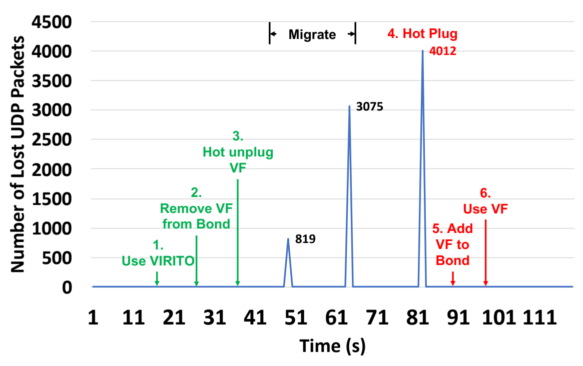
Figure 8b for the Reactive Switchover configuration shows that there are two additional events that induce significant packet loss besides the stop-and-copy phase. First when the



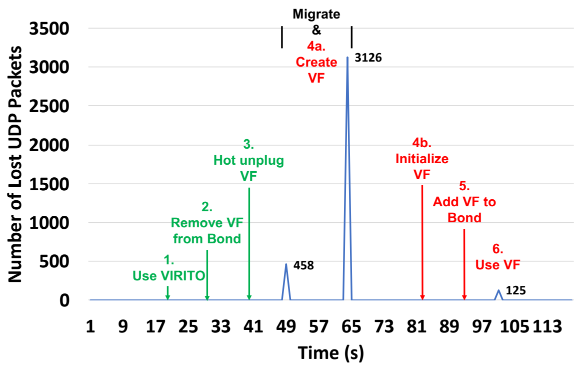
(a) Vhost-only.



(b) Reactive Switchover.



(c) Managed Switchover.



(d) Optimized Switchover.

Figure 8. Number of lost UDP packets during the live VM migration, when the guest receives UDP packets.

pass-through device is hot-unplugged at the source and second when it is hot-plugged at the destination. This is expected because bonding driver takes non-negligible time to detect the network state change and switch over the traffic.

Figure 8c for Managed Switchover configuration shows that switching over guest traffic in a managed fashion completely eliminates packet losses at the source during hot-unplug operation. However, packet losses remain at the destination during the hot-plug operation. We tracked down the cause of the latter packet losses to the fact that all guest virtual CPUs are paused by QEMU during the entirety of the hot-plug operation.

Figure 8d for Optimized Switchover shows that by splitting the relevant portions of hot-plug operation and overlapping them with memory copying phase of live migration, we can avoid pausing the guest again once it resumes at the source. Thus, all packet losses during the hot-plug phase are eliminated and the liveness matches traditional live migration of emulated guests.

Finally, we did not observe any missed timer interrupts when disabling and re-enabling the direct hardware access during migration. It took 450 μs to transform a VM from VANILLA to DID configuration, while it took 420 μs to transform the guest from DID back to VANILLA configuration.

9.3 Network I/O Performance

We now show that, with all the optimizations applied in the DID configuration, a DirectVM can match the network I/O performance of a Bare-metal server in terms of network throughput, latency, and CPU utilization, while eliminating almost all VM exits.

Throughput and Latency: In this experiment, we used two CPUs to try to saturate a 40 Gbps InfiniBand link for all configurations. To minimize CPU contention, one core handled network interrupts while another handled softirq requests and ran network workload. We used the third core to monitor CPU utilization, which did not affect the network performance. For the VHOST configuration, we used one additional core to run the VHOST worker thread in the host kernel. We experimented with two workloads. First, we used the iperf [32] tool to generate two uni-directional TCP streams to try to saturate the Infiniband link. We also used the ping [36] tool to exchange back-to-back ICMP echo-response messages between two machines.

Table 3 shows that all six configurations were able to saturate the bandwidth of outgoing traffic over InfiniBand. All but VHOST were able to saturate the bandwidth of incoming traffic over InfiniBand. The VHOST configuration needed to use three CPU cores to reach 34.36 Gbps for the incoming

Guest Configuration \ Performance	Host		Guest		Active CPUs	Network		
	User%	System%	User%	System%		Outbound (Gbps)	Inbound (Gbps)	Round-Trip Delay (μs)
Bare-metal	0.68	90.59	–	–	2	37.39	37.60	11
VHOST	79.32	172.74	0.31	35.19	3	37.61	34.36	24
VFIO	102.45	91.90	0.36	44.66	2	37.59	37.60	12
OPTI	199.58	0.42	0.85	99.97	2	37.55	37.58	12
DTID	199.58	0.42	0.89	90.55	2	37.59	37.60	12
DID	199.58	0.42	0.86	90.74	2	37.56	37.59	13

Table 3. Network throughput, latency, and CPU utilization for a network-intensive guest over 40 Gbps InfiniBand link.

Guest Configuration \ Exit Reason	EPT Fault	Physical Interrupt	HLT Instruction	PAUSE Instruction	Preemption Timer	MSR Write	Total
VHOST	13652	98	10454	104	145	1065	25518
VFIO	0	126	36672	19	151	1421	38389
OPTI	0	547	0	0	0	1446	1993
DTID	0	3	0	0	0	1147	1150
DID	0	3	0	0	0	0	3

Table 4. VM exits per second per virtual CPU when the guest sends TCP traffic over a 40Gbps InfiniBand link.

traffic. The round-trip delay if 24 μs in the VHOST configuration was approximately twice the latency of the other five configurations due to numerous VM exits.

CPU Utilization: All except the VHOST configuration are able to match the Bare-metal throughput and latency, their CPU utilizations differ in important ways. Table 3 shows the CPU utilization measured from both the host's and guest's perspective when the guest saturated the InfiniBand link. The CPU utilization is computed as a simple sum of corresponding individual utilization values across all active CPUs, leading to some values greater than 100%. *Host User* and *Host System* represents the CPU utilization in the user mode and system mode, respectively, as measured in the host (hypervisor). *Guest User* and *Guest System* represent the CPU utilization in the user and system modes, respectively, as measured in the guest.

When interpreting the *Host User* value, one should remember that it includes the *Guest User* and *Guest System* values also; from the hypervisor's perspective, any CPU time spent by the guest in non-root mode is counted against QEMU process' time usage. Thus, for OPTI, DTID, and DID configurations which disable HLT-triggered VM exits, the *Host User* time is misleadingly inflated because any CPU idle time in non-root (guest) mode will be appear as busy time in *Host User* mode.

From the table, we observe that the DTID and DID guests came very close to matching the Bare-metal CPU utilization, as expected. The OPTI guest consumed an additional 10% CPU *Guest System* time. The VFIO guest consumed less *Guest System* time but more *Host System* CPU time due to HLT-triggered VM exits. The VHOST guest spent a majority of time in *Host System* since network I/O was processed as part of the VHOST worker thread.

VM Exits: We also measured the VM exits for the two active virtual CPUs using the perf [38] tool, when the guest transmitted TCP traffic over InfiniBand. Table 4 shows that in the DTID configuration, we eliminated timer-related VM exits by enabling the direct timer-interrupt delivery. However, the DTID guest still experiences VM exits due to IPIs. Such a VM exit happens when a DTID guest receives IPIs from the Directvisor or other virtual CPUs. In the DID configurationa we used the posted-interrupt mechanism to deliver direct IPIs and eliminate all VM exits achieving close to bare-metal performance.

Among other configurations, the VHOST guest suffered the most VM exits mainly due to HLT-triggered VM exits and EPT faults.

In contrast, the VFIO guest did not trigger any VM exits when accessing its assigned network devices or receiving a network interrupt. It still experienced more *HLT*-triggered VM exits than the VHOST case. This behavior depended on the amount of time that the guest was idled on its CPUs. For the VHOST guest, the hypervisor used the guest's CPU time to handle the EPT faults and emulate *HLT* instructions. On the other hand, the VFIO guest actually had a longer idle time than the VHOST guest. When the VFIO guest occupied CPUs longer and waited for its next network I/O, it idled more often and issued *HLT*.

The OPTI configuration further eliminated *HLT*-triggered VM exits but experienced 547 VM exits per second per virtual CPU due to the local interrupts, most of which can be attributed to timer interrupts. In our test, both the hypervisor and the guest set a timer resolution of 4ms or 250Hz,

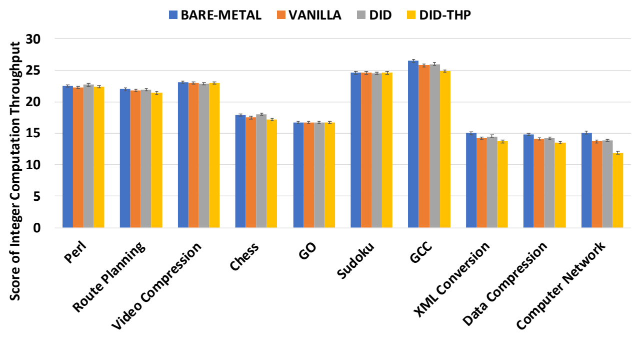


Figure 9. Comparison of integer computation throughput of SPEC CPU 2017 benchmark [9].

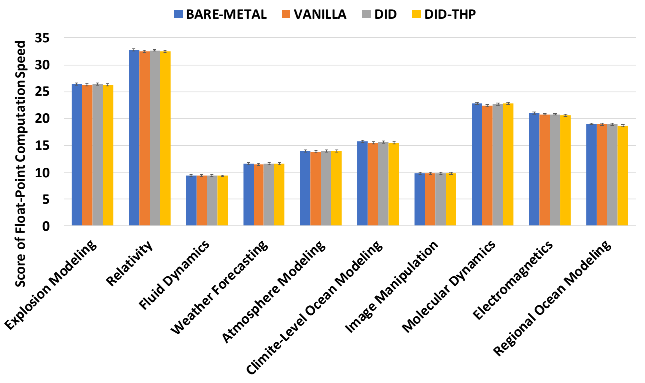


Figure 10. Comparison of floating point computation speed of SPEC CPU 2017 benchmark [9].

leading to at most 500 hardware timer interrupts every second. In addition, iperf [32] threads set up its own timers and rescheduling IPIs woke up iperf [32] threads.

In the VFIO configuration, the rate of VM exits per virtual CPU due to the local interrupts was substantially less than what we have observed in the OPTI configuration. Because of the large number of *HLT*-related VM exits, a timer interrupt could arrive at the CPU which was in the hypervisor's control. Such an interrupt did not contribute an additional VM exit.

9.4 Parallel Application Performance

We now evaluate how closely Directvisor matches the performance Bare-metal configuration when running parallel applications. We use two benchmarks for evaluation, namely SPEC CPU 2017 [9] and PARSEC [8]. SPEC CPU evaluates the performance of various CPU-intensive workloads. PARSEC evaluates various parallel workloads having diverse locality, synchronization, and memory access profiles. In these experiments, the DirectVM had eight dedicated cores and 27 GB RAM whereas Directvisor had the two cores and the remaining 5 GB RAM. The topology of L1/L2/L3 caches was exposed to the guests. The benchmark programs limited the number of active CPUs to eight. By default, QEMU/KVM

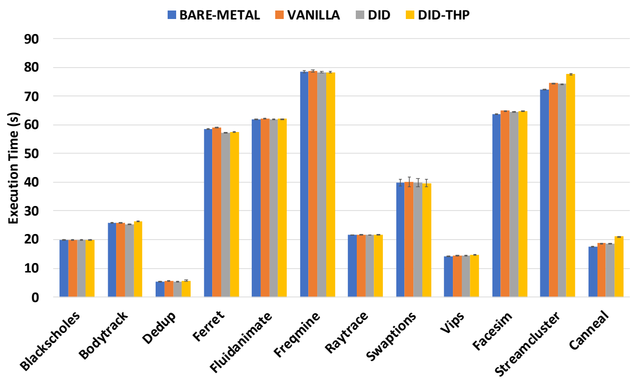


Figure 11. Comparison of PARSEC benchmark [8].

allocated the 2 MB huge pages for the guests using *madvise()* to reduce TLB miss overhead.

For SPEC CPU 2017, we ran the two suites: integer rate and floating-point speed. The integer rate suite ran eight concurrent copies of same program and measured the work done per unit time. The floating-point speed ran eight OpenMP threads [17] for each program and measured the amount of time to complete the work. Figure 9 and 10 show the performance of integer-rate and floating-point suite running in guests in comparison with the Bare-metal. Particularly, the DID-THP configuration disabled the support of transparent huge pages (THP) for the DID guest. For the integer-computation throughput, the DID guest's performance came close to Bare-metal's performance within the two standard deviations. One interesting case involved computer network simulation. The DID guest experienced 1.2 score reduction or 7.95% throughput reduction when compared to the Bare-metal case. The DID guest without the huge-page support had a worse throughput, which was 3.2 score reduction or 21.19% throughput reduction. However, enabling the huge-page support further closed the performance gap for GCC, XML conversion, and data compression. For the floating point workloads, the DID guest matched bare-metal performance.

Similarly, we ran the PARSEC benchmark [8] to study how well the DID guest compared to the Bare-metal case. We ran eight POSIX threads for each workload and measured the parallel execution time. Figure 11 shows the performance. The DID guest's performance was close to the Bare-metal's. The worst case for Canneal was about 5.85% slowdown compared to Bare-metal. Without hugepage support, the DID guest experienced 7.35% and 19.52% slowdown for Streamcluster and Canneal, respectively. There were two notable observations. First, enabling huge page support helped to reduce the performance difference between Bare-metal and DID guest down to less than 8%. We speculate that the remaining gap is due to sparse memory access pattern, resulting in more TLB misses which are more expensive for virtual machines. Second, sometimes the DID guest performed surprisingly better than the Bare-metal case. The reason could be that when the processor started to use the TLB for 2 MB huge pages, it saved the cost of walking the EPT. In comparison, the Bare-metal case still used 4 KB pages which had a much lower TLB coverage than using the 2 MB huge pages.

10 Related Work

To mitigate I/O-related virtualization costs, customers can provision VMs with direct device assignment [2, 31]. However, doing so has meant sacrificing the ability to live migrate VMs for load balancing and fault-tolerance due to the difficulty of migrating the state of physical I/O devices. Directvisor supports close-to-bare-metal performance while retaining the benefits of virtualized servers.

Live migration with direct device access: On-demand virtualization [26] presented an approach to insert a hypervisor that deprivileges the OS at the source server just before migration and re-privileges it at the destination after migration. However, this approach requires the hypervisor to fully trust the OS being migrated, the OS having full access to the hypervisor's memory and code. Such lack of isolation rules out the use of many hypervisor-level cloud services that require strong isolation, such as VM introspection and SLA enforcement. In contrast, a DirectVM cannot access or modify Directvisor's code or memory. Recent developments [23, 52] have proposed frameworks to support live migration of VMs using pass-through I/O devices. These approaches require guest device drivers to save and reconstruct device-specific I/O states for VM migration. Similar approaches [41, 50] allow a hypervisor to extract internal state of virtual functions from SR-IOV devices by monitoring and capturing writes to device-specific registers. However, these approaches require implementing device-specific state capture and reconstruction code in the VM by each device vendor besides tracking of each device's state changes by the hypervisor. In contrast, Directvisor's live migration mechanism is device agnostic; device drivers in a DirectVM do not need device-specific state capture and reconstruction code and the Directvisor does not need to intercept guest I/O state changes.

Another solution [51] closest to ours proposed using a bonding driver in the guest to connect a para-virtual and a pass-through network interface as a single network interface. Before live migration, the hypervisor hot-unplugs the pass-through interface and the guest automatically switches its network traffic to the para-virtual interface. After the migration, the hypervisor hot-plugs the pass-through interface at the destination and the guest switches the network traffic back to the pass-through device. We observed that the hot-unplug and hot-plug operations pause the guest execution long enough to introduce significant network downtime. Directvisor eliminates these disruptions by switching network traffic before and after the hot-unplug and hot-plug operations respectively besides addressing implementation-level inefficiences with the hot-plug operation in QEMU. Additionally, Directvisor also supports live migration of guests having direct control over local APIC timers and IPIs, which earlier approaches do not address.

Reducing virtualization overheads: A number of techniques have been proposed to reduce virtualization overheads in interrupt delivery by eliminating VM Exits to the hypervisor. ELI [3] and DID [48] presented techniques for direct delivery of I/O interrupts to the VM. Intel VT-d [2, 31] supports a posted interrupt delivery mechanism [39]. However, these techniques do not fully eliminate a hypervisor's role in the delivery of device interrupt. Specifically, the hypervisor must intercept and deliver device interrupts (using either virtual interrupts or IPIs) when the target VM's VCPU is not scheduled on the CPU that receives the device interrupt. Further, idle guest virtual CPUs waiting for I/O completion will trap to the hypervisor, where a halt-polling optimizations can inadvertently end up increasing CPU utilization. In contrast, Directvisor eliminates these overheads in device interrupt delivery by combining the use of Intel VT-d with optimizations to dedicate physical CPU cores to the guest and disable VM exits when VCPUs are idle.

Finally, the key distinction of Directvisor lies in direct local APIC timer and IPI access by the guest. Existing techniques do not eliminate hypervisor overheads when a VM interacts with its local APIC timer hardware and nor do they support direct delivery of timer interrupts and IPIs between virtual CPUs. Directvisor provides these features for a DirectVM through a novel use of posted interrupt mechanism and without VM exit overheads.

11 Conclusion

We presented Directvisor which provides virtualization support for bare-metal cloud platforms. Directvisor enables near-native performance for DirectVMs while retaining the manageability and isolation benefits of traditional clouds. Directvisor goes beyond traditional direct-assigned (pass-through) I/O to enable VMs to directly control and receive hardware timer interrupts and inter-processor interrupts (IPIs) besides eliminating most VM exits. At the same time, Directvisor retains the ability to perform seamless live migration of DirectVM besides providing other manageability functions such as the VM introspection.

Acknowledgments

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

References

[1] A. Kivity, Y. Kamay, K. Laor, U. Lublin, and A. Liguori. 2007. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the Ottawa Linux Symposium.* 225–230.

[2] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel Virtualization Technology for Directed I/O. *Intel technology journal* 10, 3 (2006).

[3] Nadav Amit, Abel Gordon, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2015. Bare-metal Performance for Virtual Machines with Exitless Interrupts. *Commun. ACM* 59, 1 (Dec. 2015), 108–116. https://doi.org/10.1145/2845648

[4] Atoptool.nl. 2019. atoptool: Various System Activity Reports. https://www.atoptool.nl/downloadatop.php

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Operating Systems Review* 37, 5 (2003), 164–177.

[6] Fabrice Bellard. 2005. QEMU, A fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[7] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert Van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. 2006. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLSâĂŹ06: The 2006 Ottawa Linux Symposium.* Citeseer, 71–86.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08).* ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[9] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18).* ACM, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[10] H. Chen, R. Chen, F. Zhang, B. Zang, and P.C. Yew. 2006. Live updating operating systems using virtualization. In *Proc. of ACM VEE.* Ottawa, Canada.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05).* USENIX Association, Berkeley, CA, USA, 273–286. http://dl.acm.org/citation.cfm?id=1251203.1251223

[12] Oracle Coporation. 2020. Oracle Cloud. https://cloud.oracle.com/compute/bare-metal/features

[13] International Business Machines Corporation. 2020. IBM SoftLayer. http://www.softlayer.com

[14] Microsoft Corporation. 2020. Microsoft Azure. https://azure.microsoft.com/en-us/

[15] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High availability via asynchronous virtual machine replication. In *Proc. of Networked Systems Design and Implementation.*

[16] cyclictest [n. d.]. Cyclictest: Finding Realtime Kernel Latency. http://people.redhat.com/williams/latency-howto/rt-latency-howto.txt.

[17] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[18] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *15th ACM conference on Computer and communications security (CCS).* 51–62.

[19] Yaozu Dong, Zhao Yu, and Greg Rose. 2008. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Workshop on I/O Virtualization*, Vol. 2.

[20] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of USENIX OSDI.* Boston, MA.

[21] OpenStack Foundation. 2020. Kata Containers. https://katacontainers.io

[22] Volatility Foundation. 2019. Volatility Framework - Volatile memory extraction utility framework. https://github.com/volatilityfoundation/volatility.git

[23] T. Fukai, T. Shinagawa, and K. Kato. 2018. 10.1109/TCC.2018.2848981. Live Migration in Bare-metal Clouds. *IEEE Transactions on Cloud Computing* (2018. 10.1109/TCC.2018.2848981).

[24] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection.. In *Network & Distributed Systems Security Symposium.*

[25] Michael Hines and Kartik Gopalan. March 2009. Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning. In *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Washington, DC.*

[26] Jaeseong Im, Jongyul Kim, Jonguk Kim, Seongwook Jin, and Seungryoul Maeng. 2017. On-demand Virtualization for Live Migration in Bare Metal Cloud. In *Proceedings of the Symposium on Cloud Computing (SOCC).* https://doi.org/10.1145/3127479.3129254

[27] Amazon.com Inc. 2020. Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2/

[28] Docker Inc. 2020. Docker. https://www.docker.com

[29] VMWare Inc. 2008. Architecture of VMWare ESXi. https://www.vmware.com/techpapers/2007/architecture-of-vmware-esxi-1009.html

[30] Wikimedia Foundation Inc. 2020. Intel virtualization (VT-x). https://en.wikipedia.org/wiki/X86_virtualization#Intel_virtualization

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

[32] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, Kaustubh Prabhu. 2020. iPerf: The Network Bandwidth Measurement Tool. https://iperf.fr/

[33] Google LLC. 2020. Google Cloud platform. https://cloud.google.com/

[34] Google LLC. 2020. gVisor. https://gvisor.dev

[35] Canonical Ltd. 2014. tcpdump: Dump traffic on the network. http://manpages.ubuntu.com/manpages/xenial/man8/tcpdump.8.html

[36] Canonical Ltd. 2015. iputils: Utility Tools for the Linux Networking. https://launchpad.net/ubuntu/+source/iputils

[37] Victor Ramos Mello. 2019. Diamorphine: LKM rootkit for Linux Kernels 2.6.x/3.x/4.x. https://github.com/m0nad/Diamorphine.git

[38] Ingo Molnar. 2017. perf: Linux Profiling with Performance Counters. https://github.com/torvalds/linux/tree/master/tools/perf/Documentation

[39] Jun Nakajima. 2012. Enabling Optimized Interrupt/APIC Virtualization in KVM. In *KVM Forum, Barcelona, Spain.*

[40] Gal Niv. 2019. Amazon Firecracker: Isolating Serverless Containers and Functions. https://blog.aquasec.com/amazon-firecracker-serverless-container-security

[41] Zhenhao Pan, Yaozu Dong, Yu Chen, Lei Zhang, and Zhijiao Zhang. 2012. CompSC: live migration with pass-through devices. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 109–120.

[42] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Recent Advances in Intrusion Detection.* 1–20.

[43] Rusty Russell. 2008. Virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[44] Sahil Suneja, Canturk Isci, Vasanth Bala, Eyal de Lara, and Todd Mummert. 2014. Non-intrusive, Out-of-band and Out-of-the-box Systems Monitoring in the Cloud. In *SIGMETRICS'14, Austin, TX, USA.*

[45] The kernel development community 2020. VFIO - Virtual Function I/O. https://www.kernel.org/doc/Documentation/vfio.txt

[46] Thomas Davis, Willy Tarreau, Constantine Gavrilov, Chad N. Tindel, Janice Girouard, Jay Vosburgh, Mitch Williams. 2011. Linux Ethernet Bonding Driver HOWTO. https://www.kernel.org/doc/Documentation/networking/bonding.txt

[47] Michael S. Tsirkin. 2009. vhost-net: A kernel-level virtio server. https://lwn.net/Articles/346267/

[48] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. 2015. 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).* ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2731186.2731189

[49] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.

[50] Xin Xu and Bhavesh Davda. 2016. Srvm: Hypervisor support for live migration with passthrough sr-iov network devices. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 65–77.

[51] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. 2008. Live Migration with Pass-through Device for Linux VM. In *Proceedings of the Ottawa Linux Symposium.* 261–267.

[52] Yan Zhao. 2019. QEMU VFIO live migration. https://patchwork.kernel.org/cover/10819495/