This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

ES2: Building an Efficient and Responsive Event Path for I/O Virtualization

Xiaokang Hu, Jian Li, *Member, IEEE,* Ruhui Ma, *Member, IEEE* and Haibing Guan, *Member, IEEE*

Abstract—Hypervisor intervention in the virtual I/O event path is a main performance bottleneck for I/O virtualization because of the incurred costly VM exits. The shortcomings of prior software solutions against virtual interrupt delivery, a major source of VM exits, promoted the emergence of the hardware-based Posted-Interrupt (PI) technology. PI can provide non-exit interrupt delivery without compromising any virtualization benefit. However, it only acts on the half of the event path, i.e., the interrupt path, while guests' I/O requests may also trigger a large amount of VM exits. Additionally, PI may still suffer a severe latency from the vCPU scheduling while delivering interrupts. Aiming at an optimal event path, we propose ES2 to simultaneously improve bidirectional I/O event delivery between guests and their devices. On the basis of PI, ES2 introduces hybrid I/O handling scheme for efficient I/O request delivery and intelligent interrupt redirection for enhanced I/O responsiveness. It does not require any modification to guest OS. We demonstrate that ES2 greatly reduces I/O-related VM exits with the exit handling time (EHT) below 2.5% for TCP streams and 0.1% for UDP streams, increases guest throughput by 1.9x for Memcached and 1.6x for Nginx, and keeps guest latency at a low level.

Index Terms—I/O virtualization, Hypervisor, VM exits, Virtual interrupts, I/O requests, I/O performance.

1 Introduction

Input/output (I/O) virtualization is one of the architectural foundations in today's cloud infrastructures to consolidate multiple logical I/O connections into a single physical link [2], [3], [4]. With the rapid growth of data center IP traffic (3-fold over the five years from 2016 to 2021, as forecasted by Cisco Global Cloud Index [5]), it has been a key issue to guarantee the performance of I/O virtualization in cloud-based data centers [4], [6], [7].

The communication between a guest virtual machine (VM) and its I/O devices involves a data path for data movement and an event path for notification delivery. Previous efforts, such as shared memory I/O ring [8], [9], zero copy transmit/receive [10], [11], [12] and virtual address translation for DMA [13], [14], have made the performance overhead associated with the data path largely negligible. The remaining challenges mainly lie in the event path: frequent hypervisor interventions trigger costly VM exits (i.e., guest/host context switches for trap-and-emulate [15], [16]) and lead to dramatical performance degradation [17], [18], [19], [20], [21]. Concretely, I/O transactions with a virtual (or paravirtual) device [22] incur three types of VM exits continuously: the first one caused by guests' I/O request and the other two caused by virtual interrupt delivery and completion, respectively.

Many prior studies identified virtual interrupts as the major cause of VM exits and leveraged interrupt moderation [23], [24] or substitution [17], [25], [26], [27] to reduce exits. However, doing so is far from trivial, likely impeding latency or causing wasted CPU cycles [19], [28]. Recently, a novel software-based approach was proposed in ELI [19] and DID [21] to eliminate the interrupt-related VM exits without decreasing the number of interrupts. Its main idea is to deprivilege virtual interrupt delivery by directly employing physical interrupt controllers to serve guests. However, this direct use of physical resources inevitably compromises some important virtualization features (e.g., the multiplexing of physical CPU cores) and introduces potential security issues.

The shortcomings of these software solutions promoted the emergence of a hardware-assisted technology called Posted-Interrupt (PI) [15], which opens a new era for interrupt delivery. By posting virtual interrupts in the hardware-based virtual interrupt controllers, PI provides non-exit interrupt delivery and completion without compromising any virtualization benefit. Despite its usefulness, PI still has a distance to go before reaching an optimal virtual I/O event path. First, PI only acts on the half of the event path, i.e., the interrupt path, while guests' I/O requests may also incur frequent hypervisor interventions and become another major source of VM exits, but often neglected. Second, although PI inherently supports the multiplexing of physical CPU cores, it may still suffer a severe latency from the virtual CPU (vCPU) scheduling while delivering interrupts, leading to the degradation of guests' I/O responsiveness.

Aiming at an optimal virtual I/O event path, we propose ES2, an Efficient and reSponsive Event System to simultaneously improve bidirectional I/O event delivery between guests and their devices. ES2 first takes the PI technology as a basis to provide non-exit interrupt delivery and completion. Then, to throttle the VM exits triggered by guests' I/O requests, ES2 takes inspiration from the Linux NAPI mechanism [29], [30] and adopts a hybrid scheme to efficiently deliver I/O requests. This hybrid I/O handling scheme performs proper switches between the existing exit-based notification mode and a newly-added non-exit polling

This article extends a prior conference version [1] that appeared in the 46th International Conference on Parallel Processing (ICPP '17).

- *X. Hu, R. Ma and H. Guan are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. E-mail: {hxkcmp, ruhuima, hbguan}@sjtu.edu.cn*
- *J. Li is with the School of Software, Shanghai Jiao Tong University, China. E-mail: li-jian@sjtu.edu.cn*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

mode, reaping the strengths of both notification and polling. Two kinds of mode switch algorithms are designed: (1) a generic algorithm named *perceptive mode switch*, which determines the mode to use according to the real-time I/O load; (2) a specific algorithm named *optimistic mode switch*, which is more effective in reducing VM exits for the widely-deployed query-reply type applications (e.g., database or web server). Furthermore, to enhance guests' I/O responsiveness, ES2 bridges the gap between the PI processing and the vCPU scheduler, optimizing the PI mechanism by intelligently selecting the most appropriate vCPU as the new destination of virtual interrupts.

The design of ES2 is based on the paravirtual I/O model [4], which achieves a good balance between performance and flexibility and has become today's most popular I/O virtualization technology [20], [31], [32]. We have implemented the ES2 prototype with the KVM hypervisor [33] and its paravirtual I/O standard named virtio [9]. Particularly, in consideration of the scalability of ES2 (i.e., in the case of multiple co-resident VMs with simultaneous I/O activities), we combined a number of virtio back-end I/O threads into one for fine-tuned scheduling. For another important virtual I/O model, i.e., SR-IOV [34], the VM exits triggered by guests' I/O requests can be avoided owing to the direct device assignment [14]. However, PI for SR-IOV may also suffer from the vCPU scheduling, so the interrupt redirection scheme of ES2 can be applied to it.

The performance advantages of ES2 have been validated via extensive experiments that covered VM exit rate, network throughput and I/O responsiveness. Both micro and macro benchmarks were employed. In order to evaluate the scalability of ES2, we varied the number of the co-resident tested VMs with simultaneous I/O activities. The experimental results show that ES2 greatly reduces the VM exit rate, keeping the exit handling time (EHT) for I/O processing below 2.5% for TCP streams and 0.1% for UDP streams. Additionally, with the deployment of ES2, guest throughput is increased by 1.9x for Memcached and 1.6x for Nginx, and guest latency is kept at a low level.

In summary, this work makes the following contributions:

1) We analyze the shortcomings of software solutions against interrupt-related VM exits and prove the necessity of the hardware-assisted PI technology.
2) We propose ES2 to build a virtual I/O event path with minimum VM exits and lowest latency. It does not require any modification to guest OS.
3) Two kinds of mode switch algorithms are designed for hybrid I/O handling scheme to efficiently deliver guests' I/O requests.
4) We show the practical ES2 implementation on the KVM hypervisor and evaluate its effectiveness and scalability with extensive experiments.

The rest of the paper is organized as follows. Section 2 describes the background of I/O event architecture. Section 3 discusses the related work and the remaining challenges. In section 4 and 5, we present the detailed design and implementation of ES2, respectively. In section 6, we evaluate ES2 using both micro and macro benchmarks. Section 7 gives an analysis of the ES2 overhead. Section 8 discusses the applicability of ES2 to SR-IOV. In section 9, we summarize this paper and draw a conclusion.

2 Background

This section first introduces the x86 bare-metal I/O event architecture and then highlights the challenges with virtual I/O event delivery.

2.1 x86 I/O Event Architecture

In x86 bare-metal environments, I/O events can be grouped into two categories: (1) I/O instructions executed by the system software to issue I/O requests; (2) hardware interrupts generated by devices to signal the completion of I/O operations. An I/O request is delivered to a device by accessing its registers, through either memory-mapped I/O (MMIO) or port-mapped I/O (PMIO) [35]. The delivery of an interrupt is performed by the per-core local Advanced Programmable Interrupt Controller (APIC) [36], whose primary function is to receive interrupts from internal or external sources and send them to the CPU core for handling.

A local APIC contains a series of registers to maintain the interrupt state, such as Interrupt Request Register (IRR), Interrupt Service Register (ISR) and End Of Interrupt (EOI) register. When an interrupt with vector v arrives, it first prompts the local APIC to set the v-th bit of the IRR (i.e., IRR[v] = 1). Once the local APIC delivers this interrupt to the corresponding core, IRR[v] is cleared and ISR[v] is set, indicating the interrupt with vector v is currently in service. The CPU core receiving this interrupt suspends the current executing code and uses an in-memory table, named Interrupt Descriptor Table (IDT), to invoke the appropriate handler (here is the handler with index v). When the interrupt handler finishes, it writes the EOI register to signal the completion of the interrupt processing with vector v. This action automatically triggers the local APIC to clear ISR[v] and deliver the next pending interrupt in IRR with the highest priority.

In summary, I/O event delivery in x86 bare-metal environments involves **three key operations**: (1) the execution of an I/O instruction to issue an I/O request; (2) the interrupt delivery performed by the local APIC hardware; (3) the EOI write operation (i.e., interrupt completion) from the interrupt handler.

2.2 Challenges with Virtual I/O Event Delivery

In virtualized environments, an additional software layer between the host's hardware and guest OSes, called hypervisor, complicates the delivery of I/O events. Since the above-mentioned three key operations are **privileged** ones, they inevitably incur hypervisor interventions (i.e., VM exits for trap-and-emulate) to guarantee the correct running of the virtualized system.

A VM exit is a transition between the currently running VM and the hypervisor which must exercise system control for a particular reason [15], [16]. This kind of guest/host context switch takes hundreds or thousands of cycles [37] and may cause serious cache pollution. Consequently, the frequent VM exits incurred in the virtual I/O event path significantly impede guests' I/O performance.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

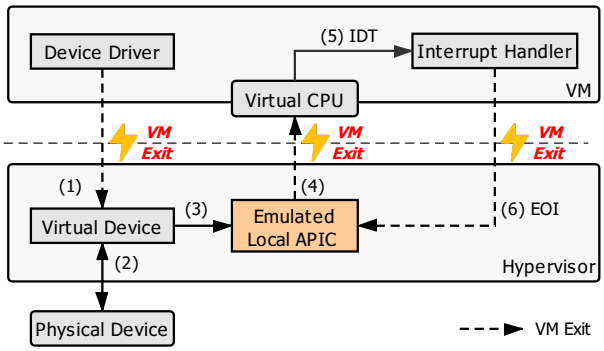


Fig. 1. I/O transactions with a virtual device

For I/O transactions with a virtual or paravirtual device [22], **three** types of VM exits are continuously triggered, as illustrated in Fig. 1:

- When the guest driver issues an I/O request (1), the **first** type of VM exit is triggered. The hypervisor handles this exit by sending a notification to the virtual device, and then resumes the guest's execution with a VM entry. When the virtual device completes the requested I/O operation through the sharing mechanism of the physical device (2), it may generate a virtual interrupt to inform the VM (3). Or, when the virtual device receives an ingress packet from the physical device (2), it may also notify the VM with a virtual interrupt (3).
- The virtual interrupt state is maintained by the per-vCPU emulated local APIC (provided by the hypervisor). If the destination vCPU is currently running, the emulated local APIC cannot directly deliver the virtual interrupt, but kicks it first with an inter-processor interrupt (IPI), triggering the **second** type of VM exit, and then performs interrupt injection during the next VM entry (4).
- Once the destination vCPU receives the injected interrupt, it invokes the corresponding interrupt handler specified in the guest IDT (5). Finally, the guest's EOI write operation triggers the **third** type of VM exit (6), prompting the hypervisor to update the corresponding registers in the emulated local APIC.

The advent of the direct device assignment [14] and SR-IOV [34] allow a VM to directly access its assigned device, thereby avoiding the exits incurred by I/O requests, but the interrupt-related VM exits (i.e., the second and third types of VM exits) still remain.

3 Related Work and Motivation

This section discusses the software solutions against VM exits in the virtual I/O environment, proves the necessity of the hardware-assisted PI technology and analyzes the remaining challenges.

3.1 Software Solutions against VM Exits

Many prior studies identified virtual interrupts as the major cause of VM exits and proposed a number of software approaches to reduce the interrupt-related VM exits. These approaches can be classified into two categories: (1) decreasing the number of virtual interrupts with either interrupt moderation or substitution; (2) retaining all interrupts yet eliminating the incurred VM exits.

Interrupt moderation: Dong et al. [23] used interrupt moderation to reduce network overhead in a paravirtual environment. Ahmad et al. [24] proposed a virtual interrupt coalescing (vIC) scheme for virtual SCSI hardware controllers to enhance I/O performance. It is true that fewer interrupts mean fewer VM exits, but doing so is far from trivial and may impede both latency and throughput [19], [28].

Interrupt substitution: Liu et al. [17] proposed Virtualization Polling Engine (VPE), which disables device interrupts and takes advantage of dedicated VPE polling threads to access I/O devices. Guan et al. [25] proposed a smart Event-Based Polling model (sEBP), which leverages existing system events to trigger a regular packet polling and thus eliminates virtual network interrupts. Some fast packet processing frameworks (e.g., Netmap [26] and DPDK [27]) adopt the poll mode driver to replace the interrupt-based driver so as to accelerate high-speed networking applications. However, it is hard to control the frequency of polling, likely leading to excess I/O latency or wasted CPU cycles [19].

Actually, device interrupt is crucial for the system software because it provides the ability to asynchronously perceive external I/O activities. And modern network devices have already enabled sophisticated interrupt coalescing from the hardware or firmware level [38]. Therefore, the above interrupt moderation or substitution approaches likely exert negative influence on the I/O sensitivity. A better solution is to retain all interrupts yet eliminate the incurred VM exits, as discussed below.

Deprivileging of interrupt delivery: ELI [19] and DID [21] presented a novel software-based approach to eliminate the interrupt-related VM exits without decreasing the number of interrupts. Its main idea is to deprivilege virtual interrupt delivery and completion by directly using physical local APICs to serve guests, thus avoiding interventions from the hypervisor. Specifically, a control bit in Virtual Machine Control Structure (VMCS) called External Interrupt Exiting (EIE) is cleared [15]. It means that when an interrupt arrives at a CPU core in guest mode, the physical local APIC directly delivers it to the currently running vCPU through the guest IDT, instead of initiating a VM exit. In addition, the physical EOI register of the local APIC is exposed to the guest, making the virtual interrupt completion become a non-virtualized operation without incurring a VM exit.

However, this kind of approach compromises some important virtualization features and introduces potential security issues. **First**, each vCPU is required to occupy a dedicated core to exclusively manipulate the physical local APIC, implying no support for the multiplexing of physical CPU cores. Assuming that vCPU A and B from different VMs run on the same core. If vCPU A is descheduled while handling an interrupt without having written the EOI register yet, the next running vCPU B may lose interruptibility since the local APIC believes a certain interrupt is still in service. Or, if vCPU A is descheduled with some pending

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

interrupts in the IRR, the local APIC may misdeliver these interrupts to the next running vCPU B. **Second**, a vCPU running on a core cannot simply migrate to other cores, disabling workload balancing. As a vCPU's interrupt state is stored in the physical local APIC, the migration of a vCPU among different cores may also cause the loss of interruptibility or interrupt misdelivery. **Finally**, the exposure of physical resources may provide malicious guests with opportunities to exert influences over other guests, or worse, the entire system.

Other types of VM Exits: Besides the interrupt-related VM exits, there are also some prior works targeting other types of exits. Ole Agesen et al. [39] proposed to identify clusters of instructions that would normally cause multiple exits and translate them together to exit only once for the whole block. This kind of technique increases the efficiency of running a virtual machine when the main exit reason is in the guest code, but it cannot alleviate the overhead for exits triggered by external interrupts [19]. ELVIS [20] used dedicated cores in the host to poll multiple guests' I/O with a fine-grained I/O scheduling, eliminating the VM exits triggered by guests' I/O requests. However, this kind of polling saturates the dedicated core even when the I/O load is at a very low level [40].

3.2 Hardware Era: Posted-Interrupt

The shortcomings of prior software solutions against interrupt-related VM exits promoted the emergence of a hardware-assisted technology called Posted-Interrupt (PI) [15], which opens a new era for virtual interrupt delivery. PI circumvents the limitation of traditional x86 architecture with the help of the per-vCPU hardware-based virtual-APIC (vAPIC) page. It can avoid interventions from the hypervisor to provide non-exit interrupt delivery and no-exit interrupt completion, without compromising any virtualization benefit.

PI processing is enabled by setting the Process Posted Interrupts (PPI) control bit in VMCS. It consists of five main steps, as depicted in Fig. 2. When the hypervisor needs to deliver a virtual interrupt to the currently running vCPU A on the core H, it first posts the interrupt information in the vCPU A's PI descriptor (step 1). Then it sends an inter-processor interrupt (IPI) with *posted-interrupt notification vector* to the core H (step 2). This special physical vector does not cause a VM exit as it would normally due to an external interrupt, but triggers the hardware to synchronize the interrupt information from the vCPU A's PI descriptor to its vAPIC page (step 3). Then the vAPIC page automatically delivers the pending interrupt indicated by the virtual IRR to the currently running vCPU A (step 4), without a VM exit. When the guest's interrupt handler finishes, the EOI write operation does not trigger a VM exit either, but prompts the hardware to update the corresponding virtual registers in the vCPU A's vAPIC page (step 5).

3.3 Gap from an Optimal Event Path

Despite the usefulness of PI, it still has a distance to go before reaching an optimal virtual I/O event path. The remaining challenges gives us opportunities for further enhancements.

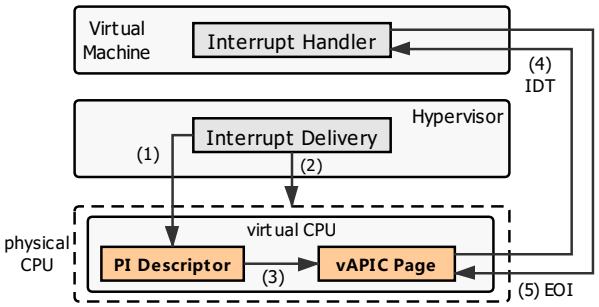


Fig. 2. Posted-Interrupt processing

Another major source of VM exits: PI is effective in eliminating VM exits, but it can only act on half of the virtual I/O event path, i.e., the interrupt path. As mentioned above, guests' I/O requests may also incur interventions from the hypervisor, which are actually another major source of VM exits, but often neglected.

TABLE 1
Breakdown of VM exit causes in the case of TCP Sending

VM Exit Causes	Interrupt Delivery	Interrupt Completion	Guest's I/O Request
w/o PI (Exits/s)	5575	30736	73310
w/o PI (%)	5.0%	27.7%	66.0%
with PI (Exits/s)	0	0	84403
with PI (%)	0.0%	0.0%	99.1%

Table 1 shows a breakdown of VM exit causes when a tested VM with a paravirtual network device is sending 512-byte TCP streams to the client server. In the configuration without PI, the number of total VM exits is up to 111,155 per second, with 32.7% (5.0% + 27.7%) triggered by interrupt delivery and completion, and 66.0% triggered by the guest's I/O requests. Interrupt delivery incurs less VM exits than interrupt completion, as the destination vCPU is likely in exit mode (considering the 73,310 VM exits caused by I/O requests) when an interrupt needs to be injected. We can see that PI eliminates the interrupt-related VM exits but could do nothing for the large amount (more than half) of VM exits triggered by the guest's I/O requests. Moreover, the elimination of interrupt-related VM exits brings a higher throughput to the guest, leading to a 15% increase in the number of guest's I/O requests and the resulting exits, from 73,310 to 84,403 per second. As a result, the large amount of VM exits triggered by guests' I/O request becomes the major challenge.

A solution to eliminate this type of VM exits is to poll guests' I/O requests in the host with dedicated cores, as proposed in ELVIS [20], but this kind of polling saturates the dedicated core even when the I/O load is at a very low level [40]. It cannot adapt well to the workloads with varied I/O traffic and likely leads to wasted CPU cycles due to polling.

Degradation of I/O responsiveness: Benefiting from the per-vCPU hardware-based interrupt controller (i.e., vAPIC page), PI retains all virtualization features and inherently supports the multiplexing of physical CPU cores. However,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

Both PI and traditional interrupt injection (using software-emulated interrupt controller) determine the destination vCPU of a virtual interrupt according to the guest's affinity setting, without awareness of the vCPU scheduling status. Considering the multiplexing of physical CPU cores, this kind of semantic gap likely leads to a severe event processing latency while delivering interrupts, thus degrading guests' I/O responsiveness [41], [42], [43], [44].

For example, when a virtual interrupt is generated, its destination is set to vCPU *A* according to the guest's affinity setting. However, If physical CPU cores are multiplexed (i.e., multiple vCPUs may share one core), the destination vCPU *A* may have been descheduled while PI delivers the virtual interrupt to it. Then, the delivered interrupt cannot be processed until vCPU *A* is rescheduled. In this situation, the scheduling delay of vCPU *A* will inevitably be introduced into the guest's event processing, causing excess I/O latency.

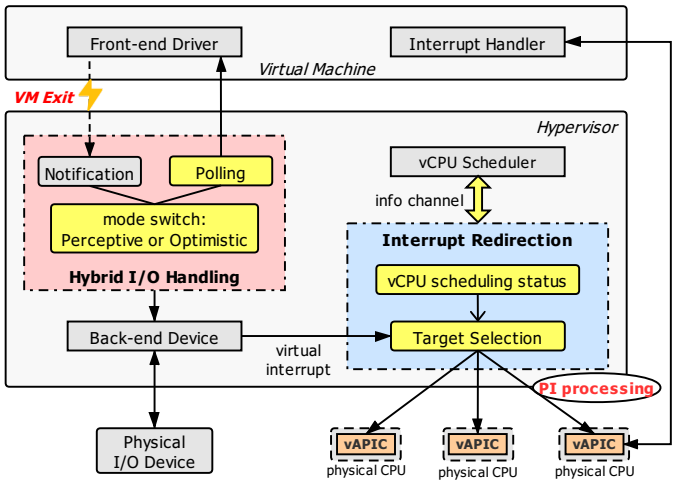


Fig. 3. Overall architecture of ES2

4 ES2 Design

Aiming at an optimal virtual I/O event path, we propose ES2, an Efficient and reSponsive Event System for I/O virtualization. We take the PI technology as a basis and make efforts to resolve the remaining challenges. The design of ES2 is based on the paravirtual I/O model, which achieves a good balance between performance and flexibility.

4.1 Overview

The overall architecture of ES2 is shown in Fig. 3, including three main components: Hybrid I/O Handling, Interrupt Redirection and PI Processing. ES2 adopts the *hybrid I/O handling* scheme to efficiently deliver the I/O requests from the guest's front-end driver. This hybrid scheme performs proper switches between the existing exit-based notification mode and a newly-added non-exit polling mode, reaping the strengths of both notification and polling. We design two kinds of mode switch algorithms: a generic algorithm named *perceptive mode switch* and a specific algorithm named *optimistic mode switch*. When the back-end device completes an I/O operation and generates a virtual interrupt, ES2 intercepts this interrupt and performs *intelligent interrupt redirection* before it is delivered through the *PI processing*. Specifically, an information channel is established to bridge the gap between the PI processing and the vCPU scheduler. According to the real-time vCPU scheduling status acquired from this channel, ES2 optimizes the PI mechanism by intelligently selecting the most appropriate vCPU (i.e., the currently running or first running vCPU) as the new destination of virtual interrupts, thus reducing the guest's event processing delay as much as possible.

4.2 Hybrid I/O Handling

The main idea of the hybrid I/O handling is to combine the existing exit-based notification mode and a newly-added non-exit polling mode. We first analyze these two modes to clarify our design motivation and then present the designed mode switch algorithms: a generic *perceptive mode switch* algorithm and a specific *optimistic mode switch* algorithm for the query-reply type applications.

4.2.1 Notification vs. Polling

In paravirtual I/O, the virtual device is divided into a front-end driver in the guest and a back-end device in the host, which communicate with each other through shared virtual queues. Each virtual queue corresponds to a handler in the host, scheduled by a back-end I/O thread.

Notification mode: When the front-end issues an I/O request, it first places the request (e.g., the data to be transmitted) into a TX (Transmit) virtual queue and then notifies the back-end with a VM exit. The hypervisor handles this VM exit to wake up the corresponding I/O thread and the latter further schedules the TX handler to process pending I/O requests. This existing exit-based notification mode works well when the egress I/O traffic is low. However, as the I/O load increases, the incurred large amount of VM exits may becomes the bottleneck. In our measurement, there are more than 80,000 VM exits per second triggered by I/O requests when a tested VM is sending TCP streams.

Polling mode: Using polling to replace notification is a straightforward method to eliminate the VM exits triggered by guest's I/O requests. Specifically, the notification mechanism of the front-end driver can be disabled and the back-end I/O thread is responsible to actively poll the TX virtual queue for I/O requests. Since the back-end I/O thread cannot predict whether there are pending works in the TX virtual queue, a typical implementation is to make back-end I/O threads, running on a number of dedicated cores, to continuously poll TX virtual queues until no new work is detected for a period of time [20]. However, this kind of side-core polling saturates the dedicated cores even when the I/O load is at a very low level (e.g., simple ping benchmark) [40]. It likely causes wasted CPU cycles and cannot adapt well to the workloads with varied I/O traffic.

Hybrid scheme: To efficiently deliver guests' I/O requests, ES2 takes inspiration from the Linux NAPI mechanism [30] and adopts a hybrid scheme to reap the strengths of both notification and polling, i.e., not only throttling the number of VM exits due to notification but also reducing the wasted CPU cycles due to polling. The key of this hybrid I/O handling is the proper switching between the existing notification mode the newly-added polling mode.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

Algorithm 1 Perceptive mode switch algorithm

```
1: procedure TX_HANDLER
2: notification:                                   ▷ Label
3:     sleeping in notification mode
4:     waked up by an I/O request
5: schedule:                                       ▷ Label
6:     waiting to be scheduled
7:     scheduled by the back-end I/O thread
8:     if notify_enabled then
9:         disable_notify                          ▷ mode switch
10:    end if
11:    load ← 0
12:    while the TX virtual queue is not empty do
13:        retrieving one I/O request from this queue
14:        load ← load +1
15:        if load >= quota then
16:            goto schedule                       ▷ Wait for next turn
17:        end if
18:    end while
19:    enable_notify                               ▷ mode switch
20:    goto notification
21: end procedure
```

Algorithm 2 Optimistic mode switch algorithm

```
1: procedure RX_HANDLER
2:     if a new packet in RX-queue is captured then
3:         disable_notify(TX-queue)                ▷ mode switch
4:         TX-queue.is_poll ← TRUE
5:         TX-queue.poll_count ← 0
6:         add the handler for TX-queue into scheduling list
7:     end if
8: end procedure
9:
10: procedure TX_HANDLER
11:    if TX-queue.is_poll then
12:        polling I/O requests from TX-queue
13:        TX-queue.poll_count ← TX-queue.poll_count + 1
14:        if TX-queue.poll_count > max_poll_count then
15:            TX-queue.is_poll ← FALSE
16:            enable_notify                       ▷ mode switch
17:            return
18:        end if
19:        waiting to be scheduled again.
20:    end if
21: end procedure
```

We design two kinds of algorithms for it, named perceptive mode switch and optimistic mode switch, respectively. Our evaluation demonstrates that the proposed hybrid I/O handling only has a small performance gap from the pure side-core polling.

4.2.2 Perceptive Mode Switch

This is a generic mode switch algorithm that determines the appropriate mode to use according to the real-time I/O load of the TX (Transmit) virtual queue. In the case of low I/O load, the notification mode is a better choice because a small number of VM exits are tolerable and preferable, compared to the wasted CPU cycles due to polling. However, if the TX virtual queue experiences high load of I/O activities, the polling mode is more appropriate as it can achieve better performance by avoiding VM exits. Since this perceptive algorithm performs prompt mode switches according to the real-time I/O load, it can adapt well to the workloads with varied I/O traffic.

The perceptive mode switch algorithm leverages a *quota* parameter to sensitively control whether the TX handler runs in the polling mode or not, as shown in Algorithm 1. At first, the handler for the TX virtual queue sleeps in the notification mode. When the handler is waked up by the guest's I/O request and gets scheduled, it directly disables the guest's notification mechanism for this virtual queue and enters the polling mode. ES2 monitors the real-time *load* when the handler polls I/O requests from this virtual queue. If the *load* reaches a predefined *quota* before the queue is empty, it is evident that the guest is experiencing a high load of I/O activities, so the handler remains in the polling mode. At this time, this handler will be descheduled and waits for its next turn, which maintains fairness among handlers and gives this virtual queue an opportunity to accumulate requests. If the guest does not have enough built-up I/O requests to fill the *quota* (i.e., $load < quota$ when the queue is empty), it means the current I/O load is at a tolerable low level. Then, the handler re-enables the guest's notification mechanism and goes back to the notification mode. We can see that once the real-time I/O load varies, this algorithm can perceive that and promptly switch the mode.

4.2.3 Optimistic Mode Switch

Although the perceptive mode switch algorithm is a generic algorithm that can be applied to all applications, it actually provides a relatively low performance enhancement for the widely-deployed query-reply type applications (e.g., database or web server), as shown in our evaluation (section 6.3.2). Additionally, an observation for the query-reply type application is that almost all the I/O requests are issued after it receives a new packet, which could be a packet for query or new connection. In other words, for the query-reply type applications, the initiation of the guest's I/O request (i.e., a packet for reply or connection establishment) is predictable to some extent. The above two reasons motivates us to design an optimized mode switch algorithm named *optimistic mode switch* for the query-reply type applications.

The main idea of the new mode switch algorithm is to optimistically poll I/O requests from the TX virtual queue a certain number of times once a new ingress packet to the guest is captured. Compared to the generic perceptive mode switch algorithm, this optimistic algorithm may incur a little overhead because of the possible ineffective polling operations, but it is more effective in reducing VM exits and provides a fairly better I/O performance for the query-reply type applications. This has been proved by our experiments in terms of both VM exit rate and network throughput (see section 6 for detail).

Algorithm 2 shows the detailed algorithm of the optimistic mode switch. Once a new packet in the RX (Receive) virtual queue is captured, the guest's notification mechanism for its TX virtual queue is disabled. The polling flag (*is_poll*) for the TX virtual queue is set to TRUE and the corresponding handler is added to the scheduling list of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

back-end I/O thread, with the *poll_count* value reset to zero. When the handler for the TX virtual queue is scheduled and finds that the polling flag *is_poll* has been set, it polls I/O requests for the virtual queue and increases the *poll_count* by one. If *poll_count* is bigger than a predefined threshold, the handler re-enables the guest's notification mechanism and goes back to the notification mode. If not, the handler remains in the polling mode and it will be descheduled to wait for its next turn.

4.3 Intelligent Interrupt Redirection

As described before, PI may suffer a severe latency from the vCPU scheduling while delivering interrupts, causing excess I/O latency in the guest. To enhance the guest's I/O responsiveness, ES2 optimizes the PI mechanism by a strategy named intelligent interrupt redirection, which is based on the following observation. The multiplexing of physical CPU cores typically occurs in the environment of symmetric multiprocessor (SMP [45]) VMs, and it is likely that an SMP VM has a vCPU that is currently running or will soon start to run again. We can keep searching for this kind of vCPU and redirecting virtual interrupts to it to reduce the guest's event processing delay.

There have been some studies (e.g., vbalance [42] and hbalance [44]) that adopt the interrupt redirection strategy to redirect interrupts from preempted vCPUs to running ones in a balanced manner, i.e., with the concentration on interrupt load balancing. In contrast, the goal of the intelligent interrupt redirection in ES2 is to select the most appropriate vCPU as the new destination of virtual interrupts to reduce the guest's event processing delay as much as possible.

As shown in Fig. 3, an information channel is established between the vCPU scheduler and the PI processing to acquire the real-time scheduling status of all vCPUs. The status of a vCPU is defined as *online* if it is currently running on a core, and defined as *offline* if not. ES2 maintains online/offline vCPU lists for all the VMs. Each time the scheduler switches the running vCPU, ES2 gets notified and updates the related lists.

Destination vCPU selection: If there are multiple vCPU candidates in the online list when an interrupt arrives, all of them can immediately receive interrupts delivered by PI without any latency. In this situation, ES2 selects the destination vCPU in consideration of both workload balancing and cache affinity. ES2 records the number of processed interrupts for each vCPU, and selects a vCPU with the lightest workload among all the candidates as the destination. Once a vCPU is selected to process a virtual interrupt, ES2 keeps redirecting the subsequent interrupts to it until it is descheduled by the vCPU scheduler, thus achieving better cache affinity.

Although an SMP VM has multiple vCPUs, it is still likely that no vCPU is online when an interrupt arrives. In this situation, ES2 selects an offline vCPU that will be the first one to regain its online status, i.e. the first running vCPU. An approximate prediction is leveraged here to quickly make the selection. It takes the philosophy that the longer the time period a vCPU remains offline, the higher the probability it has to become online again. The offline vCPU list is sorted to indicate the descheduling sequence of all vCPUs. Each time a certain vCPU is descheduled, it is removed from the online list and added to the tail of the offline list. In this manner, ES2 makes its prediction by simply returning the vCPU at the head of the offline list (i.e., the vCPU with the longest offline time) as the destination.

Effectiveness discussion: The interrupt redirection has no effect for the target VM with only one vCPU. However, for this kind of uniprocessor (UP) VM, a typical configuration is to pin the only vCPU to a dedicated core. Since the vCPU is almost always online, it can immediately receive interrupts delivered by PI without any latency. The design goal of the interrupt redirection is to serve the SMP VMs. According to the reported statistics [46], when there are two four-vCPU VMs in a four-core host, the probability of vCPU-stacking (i.e., at least two sibling vCPUs from the same VM run on the same core) is more than 40%. Once vCPU-stacking happens, it means at least two sibling vCPUs have completely different online time, which can be leveraged by the interrupt redirection scheme to select an appropriate vCPU destination. Even if we pin the sibling vCPUs of an SMP VM to different CPU cores (i.e. only sharing a core with vCPUs from other VMs) to avoid vCPU-stacking, as each physical core owns an independent scheduling queue, it is very likely that those sibling vCPUs have differences in online/offline time. Therefore, the intelligent interrupt redirection still has the opportunity to optimize the event delivery.

5 ES2 Implementation

The design of ES2 can be implemented on different hypervisors. In this section, we present the detailed implementation based on the KVM [33] hypervisor and its paravirtual I/O standard named virtio [9]. The involved line of codes (LOC) for the ES2 prototype is about 600, involving the modifications to three paravirtual I/O related files (*vhost.h*, *vhost.c* and *net.c*) and three KVM related files (i.e., *kvm_host.h*, *kvm_main.c* and *irq_comm.c*).

5.1 Hybrid I/O Handling

The virtio standard in KVM offers a user-space implementation and an in-kernel implementation for the back-end device [47]. We validated the hybrid I/O handling scheme on the in-kernel back-end implementation for paravirtual network device, called *vhost-net*, as it performs significantly better than the user-space alternative [20], [47].

The virtio standard provides *flags* and *avail_event* fields for the back-end device to temporarily suppress notifications from the guest when the host is servicing a particular virtqueue [48]. By manipulating these fields, ES2 can permanently disable the notification mechanism in the polling mode and thus avoid the VM exits triggered by the guest's I/O requests.

Combining of I/O threads: The shared buffer between the front-end driver and back-end *vhost-net* consists of two virtqueues: one for packet transmission (TX) and the other for packet reception (RX). KVM allocates a kernel I/O thread for each *vhost-net* device to schedule its TX handler (for the TX virtqueue) and RX handler (for the RX virtqueue). In ES2, multiple back-end I/O threads are combined into one joint thread for the following two reasons.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

First, the fairness among different *vhost-net* devices can be maintained well in one thread, instead of relying on the hypervisor scheduler. Second, the context switches among different back-end I/O threads can be avoided.

Note that reducing the number of I/O threads may decrease the maximum CPU time (i.e., the number of CPU cores) that these I/O threads can use. To address this issue, we restrict the number of back-end I/O threads that are combined into one, denoted by *combining_level*, with the goal to avoid the lack of computing resources. If the number of back-end I/O threads (e.g., 6) is bigger than the value of *combining_level* (e.g., 4), they are combined to multiple joint I/O threads to be able to use more CPU resources. The default *combining_level* is set to 4 and in our evaluation with high I/O load, one CPU core is sufficient to run one joint thread (i.e., 4 combined back-end I/O threads). Additionally, we made the *combining_level* value configurable through a module parameter to meet different requirements. Users can set this parameter and pin the joint threads to different cores to control the use of CPU resources.

5.2 Perceptive Mode Switch Implementation

When multiple TX handlers in the joint I/O thread reach the *quota* threshold, the round-robin scheduling gives those TX virtqueues an opportunity to accumulate I/O requests. However, if a TX handler reaches the *quota* threshold and it is the only handler in the scheduling list, the corresponding TX virtqueue has no time to accumulate the guest's I/O requests and possibly returns back to the notification mode at its next turn. In this situation, ES2 lets the I/O thread perform an interruptable sleep for $10\mu s$ (an empirical value) before scheduling the only handler. This optimization increases the probability that the only TX handler stays in the polling mode and further reduces the VM exits incurred by I/O requests. Since the sleeping of the joint I/O thread is interruptable, once a new handler, either a TX handler or a RX handler, is added to the scheduling list, the joint thread is waked up immediately to minimize I/O latency.

The *quota* parameter is the key of the perceptive mode switch algorithm, but how to determine an appropriate value for it is not simple. A value too high may render ineffective polling while a value too low may lead to frequent switches among different handlers. The NAPI mechanism has a similar parameter called *weight*, and it is usually determined by empirical methods. Analogously, we used experiments to select the proper *quota* value, as presented in our previous work [1]. we recommend a quota value of no more than 4 for TCP streams and a quota value of no more than 16 for UDP streams. Also, to facilitate modifications of the *quota* value, we added a module parameter named *poll_quota*, which allows the *quota* value to be set dynamically when loading the *vhost-net* module.

5.3 Optimistic Mode Switch Implementation

Once a RX handler receives a new packet, the corresponding TX handler is marked as a polling handler and added into the scheduling list of the joint I/O thread. The regular handlers (i.e., the TX/RX handlers triggered by VM exits) are scheduled only if there is new pending work, while the polling handlers are scheduled because we optimistically believe there may be new work in the TX virtqueues. The max polling time (i.e., *poll_count*) is set to 1000. In order to reduce the wasted CPU cycles due to possible ineffective polling operations, if all the active (i.e., needed to scheduled) handlers are polling handlers, after they have been scheduled for once, ES2 lets the joint I/O thread perform an interruptable sleep for $10\mu s$ (an empirical value) before scheduling them again. Similarly, once a new handler is added to the scheduling list, the joint thread is waked up immediately to minimize I/O latency.

A challenge here is that the sleeping scheme needs to be disabled if there is at least one regular handler is active. To this end, ES2 introduces a new polling list into the joint I/O thread, besides the existing scheduling list. At first, all the active handlers are added to the scheduling list. When a polling handler is scheduled to run and needs to be rescheduled, it is added to the polling list. Once a regular handler is identified and scheduled, all the handlers in the polling list are moved to the tail of the scheduling list. At some time, if the scheduling list is empty while the polling list is not empty, it means that all the active handlers are polling handlers and they have been scheduled for once, so the joint I/O thread sleeps for $10\mu s$. After that, all the polling handlers are moved from the polling list to the scheduling list and the loop continues.

5.4 Tracing vCPU Scheduling Status

In KVM, a vCPU is implemented as a normal thread and scheduled by the Completely Fair Scheduler (CFS) [49]. ES2 monitors the scheduling status of all vCPUs. If a vCPU thread is currently running, it is put into the online vCPU list of the given VM. When an online vCPU thread is descheduled, it is removed from the online list and added to the offline list. The accesses to these per-VM lists must be carefully synchronized, because multiple sibling vCPUs on different cores may change their scheduling status concurrently.

From the perspective of CFS, there is no difference between a vCPU thread and an ordinary thread, so we cannot trace the vCPU scheduling status by ascertaining whether the scheduled or descheduled thread is a vCPU or not. Instead, we turn to the two preemption notifiers provided by KVM, called *kvm_sched_in* and *kvm_sched_out* respectively. The former is invoked when a vCPU thread is about to be scheduled, and the latter is invoked immediately after a vCPU thread is descheduled. By leveraging these two notifiers, ES2 can collect and update the vCPU scheduling status for each VM.

5.5 Interrupt Redirection

Guest devices in KVM are implemented as standard PCI devices with the Message Signaled Interrupt (MSI) architecture or its extension MSI-X [50]. The destination vCPU ID of a virtual interrupt is specified in the MSI/MSI-X address, determined by the guest's interrupt affinity setting. ES2 does not reprograms the interrupt configuration at the sources, as this way is non-atomic, often complex and dependent on interrupt source characteristics [14]. Instead, ES2 intercepts MSI/MSI-X type virtual interrupts in a key

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

function called *kvm_arch_set_irq_inatomic*, and modifies the destination vCPU to the selected candidate.

In Linux, a device interrupt can be delivered by either the lowest priority delivery mode or the fixed delivery mode [36], depending on the selection of the local APIC driver. An interrupt using the lowest priority delivery mode can be recognized by all CPU cores, so there is no problem with the validity of the interrupt redirection. But this may not be true for the fixed delivery mode. Fortunately, the number of vCPUs in a VM is usually no more than eight, in which case the guest OS (Linux) typically selects *apic_default* or *apic_flat* as the local APIC driver and adopts the lowest priority delivery mode. For the VM with more than eight vCPUs, further guest configuration (e.g., manually setting the corresponding interrupt affinity to multiple cores in the guest) or guest OS modification is necessary to enable the interrupt redirection.

In addition, it is important to identify device interrupts here, because in an SMP VM, not all interrupts are allowed to be redirected. Some kinds of interrupts (e.g., timer interrupt) are generated for specific vCPUs and redirecting them to other vCPUs may cause the guest OS to crash. We notice that Linux adopts a strict interrupt vector allocation strategy. By taking advantage of the vector range distribution, ES2 can distinguish device interrupts from the others and perform the correct redirection.

6 Evaluation

In this section, we evaluate the effectiveness and scalability of the ES2 implementation.

6.1 Experimental Setup

We established an experiment testbed with one tested server and one client server that were connected back-to-back via Intel® XL710 40GbE NICs. Both servers were equipped with an 8-core Intel® Xeon® E5-4610 v2 CPU (hyper-threading disabled) and 32GB RAM. We ran Ubuntu with longterm Linux kernel 4.4 on both servers and all the VMs. ES2 was installed on the tested server with qemu-kvm 2.0 and the client server acted as a traffic generator. Each VM in the tested server was provisioned with one paravirtual network device using the vhost-net kernel module. The Maximum Transmission Unit (MTU) was set to its default size of 1500 bytes.

Four configurations: To evaluate all aspects of ES2, the following configurations are compared:

- *PI*: Vanilla Linux kernel 4.4 (i.e., KVM 4.4) with PI enabled.
- *PI+H*: adding the Hybrid I/O Handling scheme based on the PI configuration. Specifically, *PI+H(P)* denotes the use of the perceptive mode switch algorithm and *PI+H(O)* denotes the use of the optimistic mode switch algorithm.
- *PI+R*: adding the Intelligent Interrupt Redirection scheme based on the PI configuration.
- *PI+H+R*: adding both of the above two schemes on the basis of the PI configuration, i.e., the full ES2.

In recent Linux kernel, PI is enabled by default if the CPU supports the APIC virtualization feature, so we take "PI enabled" as the baseline configuration. We do not directly compare ES2 with previous software approaches, such as ELI or DID, because the baseline PI configuration has equivalent effect on eliminating VM exits and does not compromise any virtualization benefit.

Workload: The performance advantage of the Hybrid I/O Handling scheme depends on the workload. Under heavy workload, it outperforms the PI baseline (i.e., the exit-based notification scheme) due to the large reduction of VM exits. However, under light workload with only a few VM exits, it performs similarly compared to the PI baseline. Therefore, in our evaluation, ES2 is mainly validated using heavy workload: for both micro and macro benchmarks, the client traffic generator continuously sends packets or makes requests to the tested server. The detailed characteristics of the input traffic will be described in terms of each experiment. For real-life traffic that lies between the case of light workload and the tested case of heavy workload, the performance advantage of the Hybrid I/O Handling scheme also lies between them.

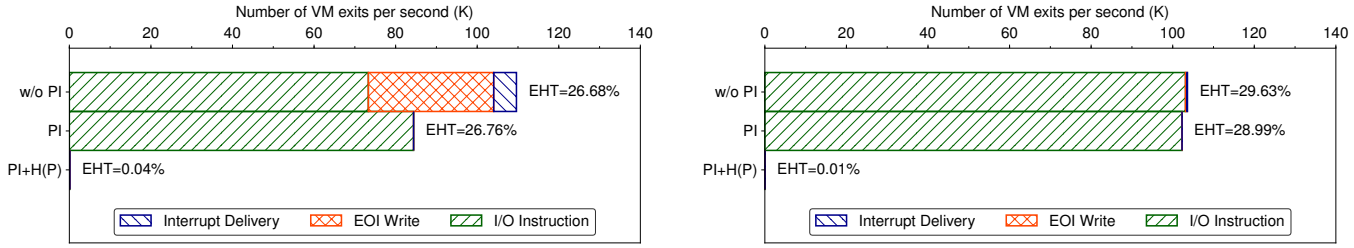
6.2 Reduction of VM Exit Rate

To quantify the ability of ES2 to eliminate VM exits, we collected the VM exit statistics of a running VM by leveraging the *perf-kvm* utility [51]. The tested VM was configured with 1 vCPU and 1GB RAM.

Based on experimental results, we have concluded three most-frequent exit causes involved in the virtual I/O event delivery: (1) External Interrupt: arrival of an external interrupt; (2) APIC Access: attempt from the guest to access the local APIC, such as the EOI write operation; (3) I/O Instruction: attempt from the guest to issue an I/O request. An External Interrupt exit may be caused by virtual interrupt delivery (which issues an IPI) or other kinds of external interrupts (e.g., timer interrupt). By analyzing the interrupt vector, we classify the IPI-triggered exits into the category named *Interrupt Delivery*. Similarly, an APIC Access exit may be caused by accessing the EOI register or other kinds of APIC registers. By analyzing the access address, we classify the exits triggered by accessing 00B0H (corresponding to the EOI register [36]) into the category named *EOI Write*. In the evaluation, we focus on these three VM exit causes: *Interrupt Delivery*, *EOI Write* and *I/O Instruction*, which are directly related to the virtual I/O event path.

We first evaluated the sending of TCP and UDP streams. The Netperf benchmark [52] was configured in the tested VM to send 512-byte TCP or UDP packets to the client server. The perceptive mode switch algorithm was adopted here in the hybrid I/O handling scheme. As shown in Fig. 4a, while the tested VM is sending TCP streams, the total number of VM exits per second in the configuration without PI is nearly 110K, with a 26.68% Exit Handling Time (EHT), which is the percentage of CPU time used to handle these VM exits. About 70K VM exits are related to *I/O Instruction* due to the continuous sending of TCP packets. The *Interrupt Delivery* exits are mainly triggered by the virtual interrupts for the ingress ACK packets. Note that *Interrupt Delivery* incurs obviously less VM exits than *EOI Write*. The reason is that when an interrupt needs to be injected, the destination vCPU is likely in exit mode due to the large

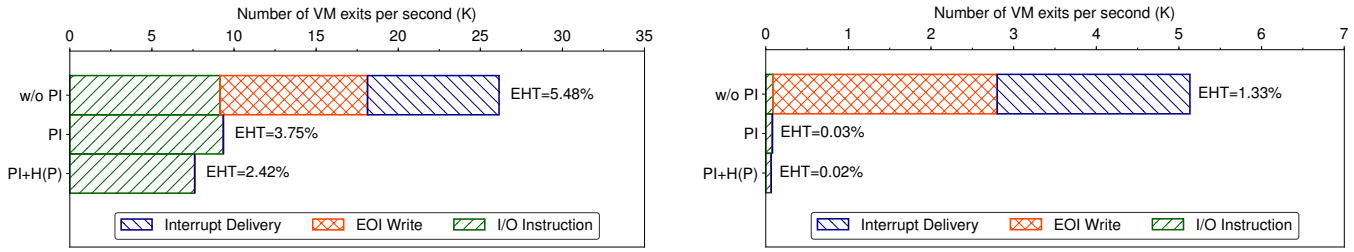
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing



(a) Sending TCP streams

(b) Sending UDP streams

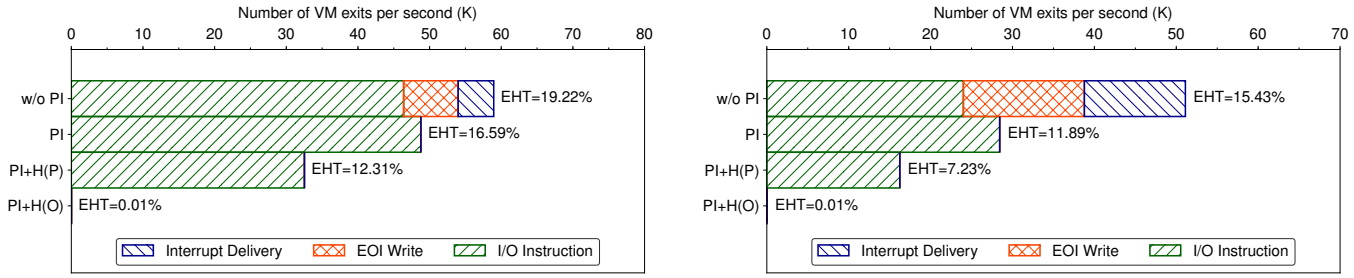
Fig. 4. Breakdown of three related VM exit causes for a VM sending 512-byte TCP or UDP streams under configurations: *w/o PI*, *PI* and *PI+H*.



(a) Receiving TCP streams

(b) Receiving UDP streams

Fig. 5. Breakdown of three related VM exit causes for a VM receiving 512-byte TCP or UDP streams under configurations: *w/o PI*, *PI* and *PI+H*.



(a) Memcached

(b) Nginx

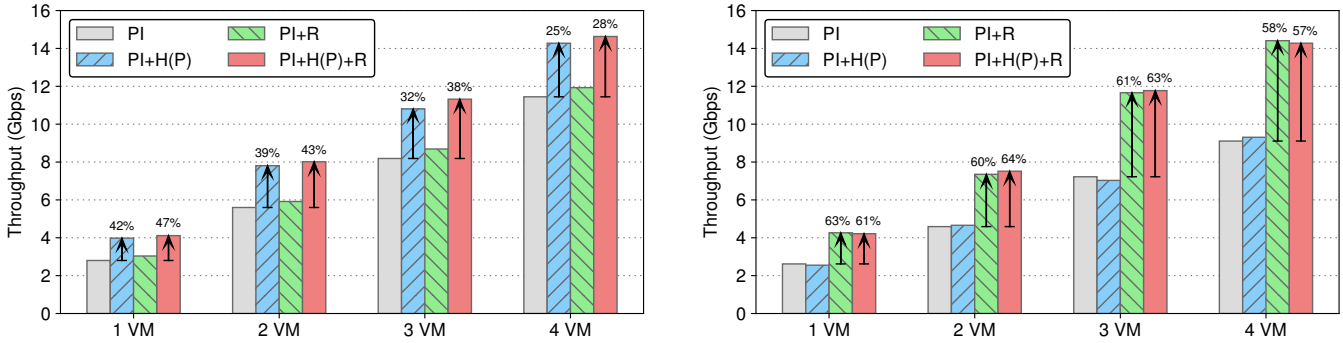
Fig. 6. Breakdown of three related VM exit causes for a VM running the query-reply type application under configurations: *w/o PI*, *PI* and *PI+H*.

amount of *I/O Instruction* exits. PI successfully eliminates the interrupt-related VM exits, which means more time can be used to send TCP packets, bringing a small increase in the amount of *I/O Instruction* exits. After the hybrid I/O handling scheme is added, the *I/O Instruction* exits are effectively throttled with a remaining number of 1K and the EHT is reduced to 0.04%. In the UDP case, as shown in Fig. 4b, the unidirectional high I/O load brings a nearly 100K *I/O Instruction* exits, with only few interrupt-related VM exits. The perceptive mode switch algorithm in the hybrid I/O handling scheme keeps the number of remaining *I/O Instruction* exits under 100 per second and reduces the EHT from 29.63% to 0.01%.

Fig. 5 shows the experiment results of the receiving of TCP and UDP streams. Still, the perceptive mode switch algorithm was adopted in the hybrid I/O handling scheme. While the tested VM is receiving TCP streams, each of the three causes contributes about a third of the VM exits in the w/o PI configuration. The *Interrupt Delivery* and *EOI Write* exits are related to the ingress TCP packets, and the *I/O Instruction* exits are triggered by the egress ACK packets. PI eliminates the former two kinds of VM exits while the *I/O Instruction* exits cannot be greatly reduced by the perceptive mode switch algorithm because ACK packets are sent only at a certain interval. In the UDP case, almost no I/O instruction exit is triggered because of the unidirectionality of UDP traffic. The PI and PI+H(P) configuration keep the EHT below 0.03%.

In order to evaluate the effectiveness of the optimistic mode switch algorithm for the query-reply type application, Memcached and Nginx were configured to run in the tested VM to handle concurrent requests from the client server. Memcached [53] is a distributed memory caching system that speeds up dynamic database-driven websites by caching data and objects in RAM. Nginx [54] is a popular high-performance web server used by a large number of top websites. The breakdown of the three types of VM exits are shown in Fig. 6. We can see that Memcached has more than 40K *I/O Instruction* exits and Nginx has more than 20K *I/O Instruction* exits in the w/o PI configuration. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing



(a) Sending TCP streams

(b) Receiving TCP streams

Fig. 7. Throughput of Netperf sending or receiving TCP streams in multiple tested VMs while four VMs are time-sharing four physical cores

interrupt-related VM exits can be successfully eliminated by the PI, bringing a small increase in the amount of the *I/O Instruction* exits. The perceptive mode switch algorithm (i.e., the PI+H(P) configuration) can reduce a number of *I/O Instruction* exits in the case of concurrent queries, with the EHT down to 12.31% for Memcached and 7.23% for Nginx. By contrast, the optimistic mode switch algorithm (i.e., the PI+H(O) configuration) is much more effective in reducing the *I/O Instruction* exits. The remaining *I/O Instruction* exits is kept under 50 per second, with the EHT down to 0.01% for both Memcached and Nginx.

As shown by the experimental results, the proposed hybrid I/O handling has already eliminated nearly all I/O instruction exits under high I/O load. It means that there is only little room for the pure side-core polling (which can entirely eliminate I/O instruction exits) to achieve additional performance enhancement. In other words, the hybrid I/O handling has a small performance gap from the pure polling. When I/O activities are not intensive (i.e., only a small number of triggered VM exits), the hybrid I/O handling scheme performs similarly compared to the exit-based notification scheme. Its main advantage is to save CPU cycles in that situation. In contrast, the pure polling uses up the dedicated cores even if the VMs are experiencing the simple ping benchmark.

6.3 Throughput Enhancement

This section evaluates the throughput enhancement of ES2 using both micro and macro benchmarks. To simulate the multiplexing of physical CPU cores, four VMs, each with four vCPUs and 1GB RAM, were created in the tested server to time-share four physical cores. We pinned the four sibling vCPUs of the same VM to different physical cores to avoid vCPU-stacking [46]. All the back-end I/O threads for the paravirtual network devices ran on a separated core, except for the case of Netperf TCP receiving, where two I/O cores were used to avoid the resource bottleneck. We also ran lowest-priority CPU burn scripts in each VM to trigger the vCPU scheduling in the hypervisor.

In order to validate the scalability of ES2, we varied the number of the tested VMs with simultaneous I/O activities from one to four. For example, when there are two tested VMs, the client server generates simultaneous I/O traffic to these two tested VMs, and the other two VMs work as the background VMs.

6.3.1 Micro Benchmark Testing

We configured Netperf [55] benchmark in multiple tested VMs to send or receive 512-byte TCP streams simultaneously to or from the client server. For each tested VM, four concurrent Netperf threads were used to fully load its four vCPUs. The perceptive mode switch algorithm was adopted in the hybrid I/O handling scheme. The experiment results are shown in Fig. 7.

Netperf TCP sending: When there is only one VM sending TCP streams, as shown in Fig. 7a, the hybrid I/O handling scheme (i.e., PI+H(P) configuration) improves the network throughput by 42% over the PI baseline, due to the effective reduction of *I/O Instruction* exits (see Fig. 4a). The intelligent interrupt redirection (i.e., PI+R configuration) only provides a slight throughput increase because only a few virtual interrupts signaling ingress ACK packets can be redirected. Finally, the full ES2 (i.e., PI+H(P)+R configuration) achieves a 47% throughput enhancement.

As the number of tested VMs increases from one to four, we can see that the throughput improvement brought by the hybrid I/O handling scheme gradually decreases from 42% to 25%, with the following reason. Since all the back-end I/O threads ran on the same core, when there are four VMs being tested in the PI configuration, the round-robin scheduling (performed by CFS) of the four I/O threads gives each VM a period of time to accumulate TX packets in the shared memory. It means the TX handler can process more packets in one shot and the number of *I/O Instruction* exits is reduced. In our measurement, the average number of *I/O Instruction* exits for each tested VM drops from 23,000 to 13,000 per second in the PI configuration when the number of tested VMs increases from 1 to 4. As fewer *I/O Instruction* exits are triggered in each tested VM, the benefit of the hybrid I/O handling scheme is cut down.

Netperf TCP receiving: As shown in Fig. 7b, the hybrid I/O handling scheme does not show obvious effect, as only a small number of *I/O instruction* exits are triggered by the sending of ACK packets and the sending is at a low rate. Notably, the intelligent interrupt redirection brings a significant increase on the network throughput, up to 63% compared to the PI configuration when there is only one

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

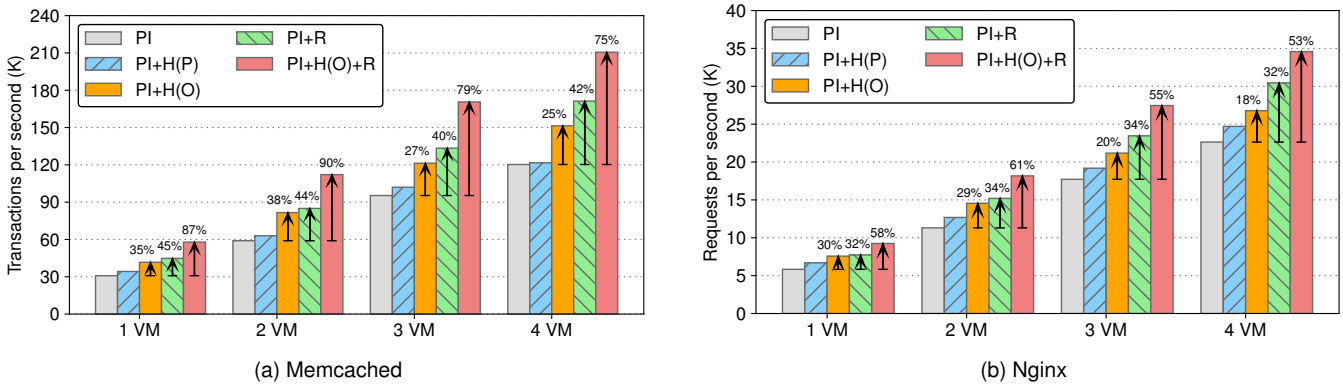


Fig. 8. Throughput of two kinds of macro workloads in multiple tested VMs while four VMs are time-sharing four physical cores

tested VM. This is because the virtual interrupts signaling ingress TCP packets are redirected to the most appropriate vCPU, thus reducing the tested VM's event processing delay. As the number of tested VMs increases from one to four, the throughput improvement brought by the intelligent interrupt redirection keeps about 60%. It proves the good scalability of the intelligent interrupt redirection.

6.3.2 Macro Benchmark Testing

We evaluated the throughput of two kinds of macro workloads: Memcached [53] and Nginx [54], both of which are query-reply type applications. The tested VMs were configured as either Memcached servers or Nginx servers. The perceptive mode switch algorithm and the optimistic mode switch algorithm were adopted respectively in the hybrid I/O handling scheme for comparison. The experiment results are shown in Fig. 8.

Memcached: We configured Memaslap [56], a load generation and benchmark tool for Memcached, to run on the client server, making 256 concurrent requests from 16 threads with a get/set ratio of 9:1. For each tested VM, one Memaslap process was launched. It is evident from the results shown in Fig. 8a that the optimistic mode switch algorithm (i.e., PI+H(O) configuration) behaves obviously better than the perceptive mode switch algorithm (i.e., PI+H(P) configuration). When there is only one tested VM, the PI+H(P) configuration only gives a throughput enhancement of 12% while the PI+H(O) configuration gives a throughput enhancement of 35%. When the number of tested VMs becomes four, the PI+H(P) configuration only shows a very slight throughput increase and while the PI+H(O) configuration still improves the throughput by 25%. These comparison results prove that the optimistic mode switch algorithm is more effective in reducing VM exits and brings a better I/O performance for the query-reply type applications.

We can also see that the increase in the number of tested VMs leads to a drop of the throughput improvement brought by these two mode switch algorithms. The reason is the same as that for the micro benchmark testing. When there are multiple VMs being tested in the PI configuration, the round-robin scheduling (performed by CFS) of the multiple I/O threads (running on the same I/O core) gives each VM a period of time to accumulate TX packets in the shared memory. As a result, the number of *I/O Instruction* exits is reduced and the benefit of the hybrid I/O handling scheme is cut down.

The intelligent interrupt redirection scheme (i.e., PI+R configuration) brings a stable more than 1.4x transactions per second over the PI baseline when the the number of tested VMs increases from one to four. This is because the virtual interrupts can be processed much more quickly. And this result shows the good scalability of the intelligent interrupt redirection scheme. The combination of the hybrid I/O handling scheme and the intelligent interrupt redirection scheme (i.e., PI+H(O)+R configuration) finally improves the throughput by 75%-90%.

Nginx: In each tested VM, four Nginx worker processes were launched on the four vCPUs to listen on different ports. We configured ApacheBench [57], a program for measuring the performance of HTTP web servers, to run on the client server, repeatedly requesting the default static page provided by Nginx. For each tested VM, four ApacheBench processes were launched to load different ports. As shown in Fig. 8b, the experiments results are similar with that of the Memcached testing. The optimistic mode switch algorithm is more effective in reducing VM exits and gives a higher requests per second (RPS) value, compared to the perceptive mode switch algorithm.

When there is only one tested VM, the PI+H(O) configuration and PI+R configuration provide 30% and 32% throughput improvement over the PI baseline respectively. The full ES2 finally improves the throughput by 58%. As the number of tested VMs increases from one to four, the throughput improvement brought by the hybrid I/O handling scheme drops to 18% and the final throughput enhancement provided by the full ES2 is 53%.

6.4 Responsiveness Improvement

This section evaluates the responsiveness improvement of ES2. The VM setting is the same as the throughput evaluation. Still, four VMs were created to time-share four physical cores, but there is only one tested VM. As the hybrid I/O handling scheme has no obvious effect on the responsiveness, here we only compare two configuration: PI and PI+R.

Round trip time: We used Ping with one second interval to measure the round trip time (RTT) from the client server

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

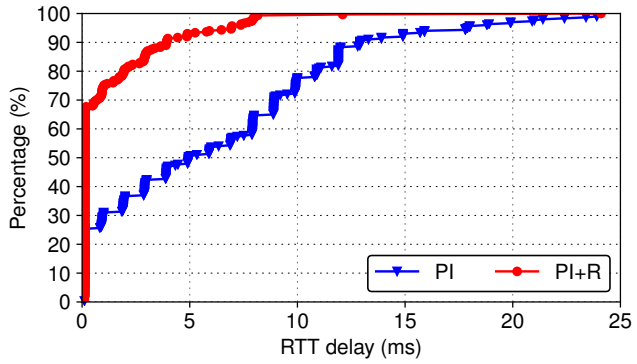


Fig. 9. Round trip time (RTT) evaluation with Ping

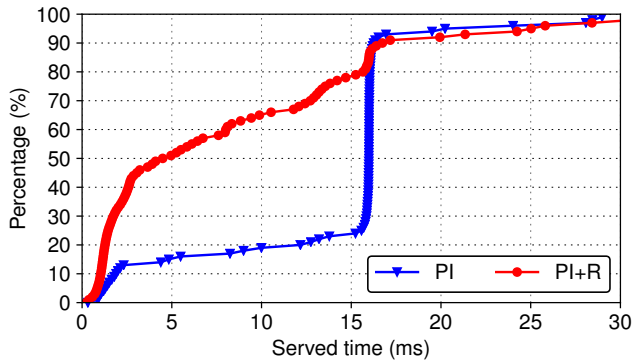


Fig. 10. HTTP served time evaluation with ApacheBench

to the tested VM. Fig. 9 shows the testing results of RTT delay. It can be seen that in the PI configuration, the RTT largely varies: half of the ping requests have a RTT above 5ms, with 24ms peak reached. After adding the intelligent interrupt redirection, nearly 70% ping requests have a RTT under 0.2ms and more than 90% ping requests have a RTT under 5ms. This proves that the original PI suffers from the degradation of I/O responsiveness. The intelligent interrupt redirection makes sure that the virtual interrupt can be delivered to the most appropriate vCPU and largely mitigates this problem.

HTTP served time: The tested VM was set up as an Apache server. The Apache HTTP Server [58] is the world's most used web server software. We configured one ApacheBench process in the client server to repeatedly request a small static page with a concurrency of 16. As shown in Fig. 10, no more than 30% HTTP requests can be served within 15ms in the PI configuration while nearly 80% HTTP requests can be served within 15ms in the PI+R configuration. A phenomenon is that in the PI configuration, about 90%-30%=60% HTTP requests are served with the same time: 16ms. This is because there is only one vCPU in the tested VM that can process interrupts and this vCPU shares the physical core with three other vCPUs from different VMs. If this vCPU is descheduled when an HTTP request arrives, this request can not be processed until it is rescheduled. Therefore, the 16ms means that the only vCPU is rescheduled and a large number of HTTP requests can be processed by it. The intelligent interrupt direction leverages all the vCPUs of the tested VM to process interrupts and always selects the appropriate vCPU as the new destination of arrived interrupts. That is the reason why it can reduce the average HTTP served time.

7 Overhead Analysis

The intelligent interrupt redirection involves two main operations. The first operation is the tracing of the vCPU scheduling status, which is performed when a vCPU thread is scheduled or descheduled. Compared to the heavy context switch, the overhead incurred by this operation is negligible. The second operation is the modification of the interrupt destination ID, which is also negligible in the overhead.

The perceptive mode switch algorithm in the hybrid I/O handling scheme performs prompt switches between the polling mode and the notification mode. Once the workload in one polling cycle does not reach the predefined *quota*, the handler returns to the notification mode immediately. It can be seen that it does not incur ineffective polling operations and the overhead is negligible.

The optimistic mode switch algorithm in the hybrid I/O handling is the major source of ES2 overhead. Once a new ingress packet is captured, the TX virtual queue is polled optimistically a certain number of times, which inevitably introduces ineffective polling operations. In order to reduce the wasted CPU cycles, a sleeping scheme is adopted for the back-end I/O thread, as described in section 5.3. We collected the average CPU usage of the back-end I/O thread in the Memcached Throughput Testing to evaluate the overhead incurred by the optimistic mode switch. In the case of one tested VM, the hybrid I/O handling along with the optimistic mode switch algorithm brings a 35% throughput enhancement over the PI baseline. At the same time , the CPU usage of the back-end I/O thread increases from 13.58% to 18.6%. So the overhead of the optimistic mode switch algorithm can be calculated by:

$$\frac{18.6 - 13.58 \times (1 + 35\%)}{13.58 \times (1 + 35\%)} = 1.45\%$$

In consideration of the 35% throughput enhancement, the incurred 1.45% overhead is acceptable.

8 Applicability of ES2 to SR-IOV

As mentioned in section 2.2, the advent of the direct device assignment [14] and SR-IOV [34] allow a VM to directly access its assigned device to issue I/O requests, without triggering any VM exit. However, the interrupt delivery still incurs hypervisor interventions. Each time an assigned device generates an interrupt, this interrupt is first intercepted by the hypervisor with a VM exit. The hypervisor handles this interrupt through the host IDT and then injects the converted virtual interrupt into the VM owning this device.

ES2 leverages the hardware-based PI technology (to be exact, called CPU-side PI) to eliminate the hypervisor intervention for virtual interrupt delivery from an I/O core (where the virtual device runs) to the running VM. A similar hardware-based technology called Vt-d PI [14] is designed for SR-IOV to avoid the VM exit when the assigned device

generates an interrupt into a running VM. For each interrupt source from any assigned device, the hypervisor allocates a posted-format Interrupt Remapping Table Entry (IRTE) for it. By tracing the logical processor where the destination vCPU resides, the Notification Destination (NDST) field will be automatically updated. When an assigned device generates a remappable interrupt according to the IRTE, this interrupt with a special notification vector is delivered to the logical processor specified by NDST. If the destination vCPU is currently running, the processor hardware processes this notification vector by transferring any posted interrupt to the vAPIC page and directly delivering it to the running vCPU through guest IDT, without any hypervisor intervention.

Still, Vt-d PI may suffer a severe latency from the vCPU scheduling while delivering interrupts as the destination vCPU may be de-scheduled when an interrupt from the assigned device arrives. Therefore, the intelligent interrupt redirection scheme of ES2 can be applied to Vt-d PI for optimization as well. By tracing the vCPU scheduling status, we can update the IRTE (NDST field) to redirect the interrupt to the most appropriate vCPU with the minimal event processing latency.

9 Conclusion

This paper focused on the elimination of hypervisor interventions in the virtual I/O event path, which trigger frequent VM exits and lead to dramatic performance degradation. We proposed ES2, a comprehensive scheme that simultaneously improves bidirectional I/O event delivery between guests and their devices. ES2 eliminates the interrupt-related VM exits with the help of PI and efficiently delivers guests' I/O requests by the hybrid I/O handling scheme, where two kinds of mode switch algorithms are offered. Furthermore, ES2 adopts a strategy named intelligent interrupt redirection to optimize PI, enhancing guests' I/O responsiveness. We implemented the ES2 prototype with the KVM hypervisor and the virtio paravirtual I/O standard. It does not require any modification to guest OS or compromise any virtualization benefit. Extensive experiments have demonstrated that ES2 substantially improves the performance of I/O virtualization in terms of throughput and latency.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments. This work is supported in part by the National Natural Science Foundation of China (No. 61972245), the National Key Research and Development Program of China (No. 2016YFB1000502) and the National Science Fund for Distinguished Young Scholars (No. 61525204). Jian Li is the corresponding author.

References

[1] X. Hu, W. Zhang, J. Li, R. Ma, F. Wu, and H. Guan, "Es2: Aiming at an optimal virtual i/o event path," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2017, pp. 141–150.

[2] Wikipedia. (2016, Oct.) I/o virtualization. [Online]. Available: https://en.wikipedia.org/wiki/I/O_virtualization

[3] C. Waldspurger and M. Rosenblum, "I/o virtualization," *Communications of the ACM*, vol. 55, no. 1, pp. 66–73, 2012.

[4] F.-F. Zhou, R.-H. Ma, J. Li, L.-X. Chen, W.-D. Qiu, and H.-B. Guan, "Optimizations for high performance network virtualization," *Journal of Computer Science and Technology (JCST)*, vol. 31, no. 1, pp. 107–116, 2016.

[5] Cisco. (2018, Nov.) Cisco global cloud index: Forecast and methodology, 2016-2021 white paper. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html

[6] M. Bourguiba, K. Haddadou, I. El Korbi, and G. Pujolle, "Improving network i/o virtualization for cloud computing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 3, pp. 673–681, 2014.

[7] J. Li, S. Xue, W. Zhang, Z. Qi, and H. Guan, "When i/o interrupt becomes system bottleneck: Efficiency and scalability enhancement for sr-iov network virtualization," *IEEE Transactions on Cloud Computing (TCC)*, 2017.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," vol. 37, no. 5, pp. 164–177, 2003.

[9] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[10] H. R. Mohebbi, O. Kashefi, and M. Sharifi, "Zivm: A zero-copy inter-vm communication mechanism for cloud computing," *Computer and Information Science (CIS)*, vol. 4, no. 6, p. 18, 2011.

[11] LWN. (2011, Apr.) macvtap/vhost tx zero copy support. [Online]. Available: https://lwn.net/Articles/439531/

[12] K. Meth, M. Rapoport, J. Nider, and R. Ladelsky, "Zero-copy receive path in virtio," in *Proceedings of the ACM International Systems and Storage Conference (SYSTOR)*, 2017, p. 19.

[13] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 72, no. 11, pp. 1471–1480, 2012.

[14] Intel. Intel® virtualization technology for directed i/o architecture specification. [Online]. Available: https://software.intel.com/en-us/download/intel-virtualization-technology-for-directed-io-architecture-specification

[15] ——. (2018, Nov.) Intel® 64 and ia-32 architectures software developer's manual volume 3c: System programming guide, part 3. [Online]. Available: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-volume-3c-system-programming-guide-part-3

[16] D. Ott. (2009, June) Virtualization and performance: Understanding vm exits. [Online]. Available: https://software.intel.com/en-us/blogs/2009/06/25/virtualization-and-performance-understanding-vm-exits

[17] J. Liu and B. Abali, "Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2009, pp. 225–234.

[18] B. Zhang, X. Wang, R. Lai, L. Yang, Z. Wang, Y. Luo, and X. Li, "Evaluating and optimizing i/o virtualization in kernel-based virtual machine (kvm)," in *Proceedings of the International Conference on Network and Parallel Computing (NPC)*, 2010, pp. 220–231.

[19] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: bare-metal performance for i/o virtualization," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 411–422.

[20] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.

[21] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. 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)*, 2015, pp. 1–15.

[22] J. Herrmann, Y. Zimmerman, D. Parker, L. Novicha, J. East, and S. Radvan. Virtualized hardware devices. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_getting_started_guide/sec-virtualization_getting_started-products-virtualized-hardware-devices

[23] Y. Dong, D. Xu, Y. Zhang, and G. Liao, "Optimizing network i/o virtualization with efficient interrupt coalescing and virtual receive side scaling," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, 2011.

[24] I. Ahmad, A. Gulati, and A. Mashtizadeh, "vic: Interrupt coalescing for virtual machine storage device io," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[25] H. Guan, Y. Dong, K. Tian, and J. Li, "Sr-iov based network interrupt-free virtualization with event based polling," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 31, no. 12, pp. 2596–2609, 2013.

[26] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[27] Intel. (2019, Jan.) Data plane development kit. [Online]. Available: https://www.dpdk.org/

[28] M. Ben-Yehuda, M. Factor, E. Rom, A. Traeger, E. Borovik, and B.-A. Yassour, "Adding advanced storage controller functionality via low-overhead virtualization," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[29] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet." in *Proceedings of the 5th Annual Linux Showcase & Conference*, 2001.

[30] Wikipedia. (2018, Sep.) New api. [Online]. Available: https://en.wikipedia.org/wiki/New_API

[31] K. Kontodimas, P. Kokkinos, Y. Kuperman, A. Houbavlis, and E. Varvarigos, "Analysis and evaluation of scheduling policies for consolidated i/o operations," *Journal of Grid Computing*, vol. 15, no. 1, pp. 107–125, 2017.

[32] L. Zeng, Y. Wang, X. Fan, and C. Xu, "Raccoon: A novel network i/o allocation framework for workload-aware vm scheduling in virtual environments," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 9, pp. 2651–2662, 2017.

[33] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[34] Intel. Pci-sig sr-iov primer an introduction to sr-iov technology. [Online]. Available: https://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf

[35] Wikipedia. (2018, Sep.) Memory-mapped i/o. [Online]. Available: https://en.wikipedia.org/wiki/Memory-mapped_I/O

[36] Intel. (2018, Nov.) Intel® 64 and ia-32 architectures software developer's manual volume 3a: System programming guide, part 1. [Online]. Available: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-volume-3a-system-programming-guide-part-1

[37] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, pp. 2–13, 2006.

[38] Intel. Intel® ethernet controller x710/xxv710/xl710: Datasheet. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf

[39] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software techniques for avoiding hardware virtualization exits," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[40] Y. Kuperman, E. Moscovici, J. Nider, R. Ladelsky, A. Gordon, and D. Tsafrir, "Paravirtual remote i/o," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[41] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu, "vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2012, pp. 3–14.

[42] L. Cheng and C.-L. Wang, "vbalance: using interrupt load balance to improve i/o performance for smp virtual machines," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2012.

[43] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu, "vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core," in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013, pp. 243–254.

[44] L. Cheng and F. C. Lau, "Offloading interrupt load balancing from smp virtual machines to the hypervisor," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 11, pp. 3298–3310, 2016.

[45] Wikipedia. (2019, Jan.) Symmetric multiprocessing. [Online]. Available: https://en.wikipedia.org/wiki/Symmetric_multiprocessing

[46] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for smp vms?" in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2011.

[47] J. Stecklina, "Shrinking the hypervisor one subsystem at a time: a userspace packet switch for virtual machines," *ACM SIGPLAN Notices*, vol. 49, no. 7, pp. 189–200, 2014.

[48] OASIS. (2016, Mar.) Virtual i/o device (virtio) version 1.0. [Online]. Available: http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html

[49] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the linux scheduler," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 34–43, 2008.

[50] Wikipedia. (2019, Feb.) Message signaled interrupts. [Online]. Available: https://en.wikipedia.org/wiki/Message_Signaled_Interrupts

[51] A. Kivity, "Performance monitoring for kvm guests," in *Proceedings of the KVM Forum*, 2011.

[52] Wikipedia. (2018, Dec.) Netperf. [Online]. Available: https://en.wikipedia.org/wiki/Netperf

[53] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 2004, no. 124, p. 5, 2004.

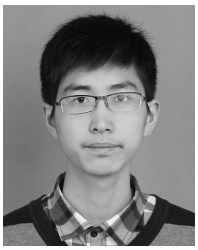
[54] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[55] R. Jones *et al.*, "Netperf: a network performance benchmark," *Information Networks Division, Hewlett-Packard Company*, 1996.

[56] M. Zhuang and B. Aker. memaslap - load testing and benchmarking a server. [Online]. Available: http://docs.libmemcached.org/bin/memaslap.html

[57] A. S. Foundation. ab - apache http server benchmarking tool. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[58] R. T. Fielding and G. Kaiser, "The apache http server project," *IEEE Internet Computing*, vol. 1, no. 4, pp. 88–90, 1997.



Xiaokang Hu is a Ph.D. candidate majoring in the computer science and technology at Shanghai Jiao Tong University. He obtained his B.S. degree in computer science and technology from Nanjing University, China. His research interests include virtualization, cloud computing, heterogeneous system and system security.

Jian Li is an Associate Professor in the School of Software at Shanghai Jiao Tong University. He obtained his Ph.D. in Computer Science from the Institut National Polytechnique de Lorraine (INPL) - Nancy, France in 2007. He is a member of ACM, IEEE and CCF. His research interests include virtualization, networking system and cloud computing.

RuHui Ma is an Associate Professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. He received his Ph.D. degree in computer science from Shanghai Jiao Tong University in 2011. His main research interests include virtual machines, computer architecture and compiling.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2969660, IEEE Transactions on Cloud Computing

Haibing Guan is a full professor in department of Computer Science at Shanghai Jiao Tong University, China. He received his Ph.D. degree in computer science from the Tongji University (China), in 1999. He is a member of ACM, IEEE and CCF. His current research interests include but are not limited to computer architecture, compiling, virtualization and hardware/software co-design.