CoINT: Proactive Coordinator for Avoiding Interruptability Holder Preemption Problem in VSMP Environment

Wang Zhang†§, Xiaokang Hu†§, Jian Li†‡ and Haibing Guan†
†School of Software Engineering
‡Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
Email: {miwa1997, hxkcmp, li-jian, hbguan}@sjtu.edu.cn

Abstract—In a Virtual Symmetric Multiprocessing (VSMP) environment, the behavior of hypervisor scheduler can significantly influence a guest’s I/O responsiveness. The interrupt remapping mechanism, which can leverage multiple virtual CPUs in the VSMP guest to process I/O events, is known to be an efficient and prevalent solution to improve the I/O performance. However, in this paper we identify a novel challenge called the “Interruptability Holder Preemption” (IHP) problem in interrupt remapping mechanism. To solve this problem, we propose CoINT, a gasket coordinator residing in the hypervisor, to substantially enhance the network I/O performance by empowering the hypervisor to be proactively aware of the interruptability information of the guest’s network device. CoINT completely eliminates the “Interruptability Holder Preemption” problem and largely reduces I/O interrupt processing delay caused by hypervisor scheduler. We implement CoINT in KVM hypervisor and evaluate its efficiency and responsiveness using both macro and micro-level benchmarks. The results show that CoINT can improve the netperf throughput up to 3x compared with native KVM, and up to 1.3x compared with traditional interrupt remapping of hypervisor-level solution, in sacrifice of an negligible and reasonable overhead in the hypervisor.

I. INTRODUCTION

Input/Output (I/O) virtualization has become the determinant of system performance and efficiency in virtualization infrastructure [1]–[8], which consolidates multiple virtual machines (VM) instances to share the same underlying physical server resource by means of the hypervisor or Virtual Machine Manager (VMM). The virtual interrupt delivering latency and the associated interrupt processing consumption can significantly influence the I/O throughput and responsiveness of the VM. This effect has been widely recognized as a closely relevance to the behavior of the hypervisor scheduler [9], [10].

The Virtual Symmetric Multiprocessing (VSMP), can furnish a VM with multiple virtual processor, thus providing a higher parallelization and resource utilization for processing the concurrent user requests, as well as the more powerful capacity to achieve higher I/O throughput and responsiveness. Consequently, VSMP has become prevalent in enterprise-class applications such as trading systems, databases and web servers. Moreover, some hardware-based schemes, such as Message Signal Interrupt (MSI) and Posted-Interrupt (PI) mechanisms, are proposed to effectively deliver the interrupts from network device into the specified VM so that the interrupt can be processed by the virtual CPU (vCPU) as soon as possible. However, the VMM scheduling policy, for purpose of multiplexing the physical CPU (pCPU) resource among different vCPUs in co-hosted VSMP guests, complicates the procedure of virtual interrupt processing and results in great fluctuations and unpredictability of the network I/O.

Numerous methods have been proposed to improve the I/O responsiveness and throughput in VSMP environment, including a) side-core approaches, b) scheduling-related methods and c) interrupt remapping schemes. The side-core approaches [11]–[13] leverage a dedicated core to serve the I/O processing of the VM, and the scheduling-related methods [14]–[18] reduces the time slice of the hypervisor scheduler to decrease the waiting time of each vCPU in the runqueue. Apparently, these two methods inevitably incur high resource consumption and introduce the extra unnecessary context-switch overhead. These effects limit their feasibility of wide deploying in most close-sourced commercial OSes under a cloud environment. Interrupt remapping schemes can avoid the deficiencies of high resource cost, and use some wise strategy to redirect the interrupt processing workload among the vCPUs of a VSMP guest [19]–[21]. This is an attractive, efficient and prevalent mechanism to improve the I/O performance under VSMP environment.

However, this paper first reveals a novel challenge that can easily invalidates the interrupt remapping mechanism in VSMP environment, called the Interruptability Holder Preemption (IHP) problem. The IHP issue is caused by the lack of coordination between the hypervisor scheduler and the network device driver in the guest, since the hypervisor scheduler is mostly agnostic to the interruptability of the

§Co-equal First Author †Corresponding Author
guest’s network device.

Concretely, after receiving a virtual interrupt that notifies incoming packets, a vCPU executing the interrupt handler will disable the interruptability of the guest’s network device. Then, the network device will leverage the NAPI mechanism to process the subsequent I/O, thus mitigating costly interrupt processing overhead [22], [23]. At this time, the specific vCPU, running the virtual interrupt handler, becomes the interruptability holder before it re-enables the interruptability of the guest’s network device. It is worth noting that the interruptability holder (IH) is highly likely to be descheduled by the hypervisor scheduler and the IHP issue occurs. The IHP problem results that the subsequent network I/O events from the physical NIC cannot be delivered to the guest, and then the available packets in the shared memory cannot be processed timely even there are available pCPU cycles for other vCPUs of the VSMP guest. The IHP problem decreases the efficiency of the interrupt remapping mechanism, and makes the guest suffer from an inappropriate scheduling policy, with high I/O latency and degraded I/O throughput being involved.

Traditionally, intelligent interrupt redirection methods over the interrupt remapping mechanisms have been proposed both in guest-level and in hypervisor-level, such as vBalance [19], hBalance [20] and vINT [21]. The main idea behind these schemes are similar, which is migrating the I/O interrupts from a descheduled vCPU to a running vCPU in the VSMP guest to shorten the latency of I/O interrupt processing. Although none of these previous work has concerned the IHP issue, all of them have drawn the conclusion in common that the interrupt remapping scheme residing in guest OS can easily obtain the vCPU running status, but inevitably involves heavy modifications to the guest OS and is not applicable to the most close-sourced commercial OSes. By contrast, a hypervisor-level solution, such as hBalance and vINT, makes all decisions for selecting the interrupt targeted vCPU in the hypervisor and is completely transparent to the guest OS. Therefore, a hypervisor-level solution is more feasible, effective and efficient, while it is still challenging to solve the IHP issue.

To this end, CoINT is proposed based on the hypervisor-level interrupt remapping framework, which coordinates the hypervisor scheduler to be aware of the interruptability information of guest’s network device in a VM-agnostic manner and schedule vCPUs of the VSMP guest in a proactive manner. As a gasket coordinator residing in the hypervisor, CoINT can effectively avoid the IHP issue of the interrupt remapping mechanism and significantly enhance the network I/O performance for the VSMP guests. The experimental evaluations show that CoINT can improve the netperf throughput up to 3x compared with native KVM, and 1.3x compared with traditional interrupt remapping of hypervisor-level solution, in sacrifice of an negligible and reasonable overhead in the hypervisor.

II. BACKGROUND AND CHALLENGE

This section first introduces the I/O processing workflow for the VSMP guest in the paravirtual I/O model. Second, it depicts the interrupt remapping mechanism in the hypervisor. At last, it elaborates the Interruptability Holder Preemption issue in detail.

A. I/O Interrupt Processing in Paravirtual I/O

On bare-metal x86 servers, A Local APIC (LAPIC) is attached to each CPU core to receive and handle interrupts. There are a series of registers in the LAPIC to trace the interrupt status, such as the Interrupt Request Register (IRR) and the End Of Interrupt (EOI) register which records the pending of an interrupt and the completion of the interrupt processing, respectively. After a CPU core receives an interrupt, the Interrupt Descriptor Table (IDT) is introduced to decide and invoke the appropriate interrupt handler. In order to comply to the requirements of virtualization, a) the hypervisor emulates the LAPIC registers for each vCPU to forward the virtual interrupts and (b) a guest IDT is introduced to distribute the virtual interrupts.

To illustrate the I/O processing workflow in VSMP guest, we take paravirtual I/O model as example. In the paravirtual I/O, a "split-driver" mechanism is implemented for the I/O devices, where a frontend driver appearing as a normal device runs in the guest OS and communicates with a backend driver running in hypervisor by means of the shared memory. As shown in Figure 1. The I/O processing procedure can be decomposed into eight steps:

**Step 1**: Once the backend device driver receives packets from software network bridge [24] or virtual switch [25], it copies them into the Receive Buffers in shared memory and injects a virtual interrupt to notify the guest.

**Step 2**: The virtual interrupt is then taken over by emulated LAPIC of the interrupt targeted vCPU.

**Step 3**: The corresponding interrupt handler in the guest IDT is invoked.

**Step 4**: The interrupt handler launches an EOI writing operation to notify the emulated LAPIC that the current interrupt has been served.

**Step 5**: The interrupt handler starts to process the network events of the guest.

**Step 6**: The interruptability of the frontend driver is first disabled by the interrupt targeted vCPU (IH), and the guest
will no longer receive any interrupts until the interruptability is re-enabled.

**Step 7:** A `softirq` is then raised in guest to perform the `poll()` function to enter the polling mode for packet receiving from the shared ring buffer between frontend and backend.

**Step 8:** Once packet receiving completes or a predefined polling threshold is reached, the frontend driver exits the polling mode and the interruptability is re-enabled by the IH.

Notice that the whole procedure of **Step 5-8** is known as polling mode with the NAPI mechanism for the guest’s I/O processing. A data field named *budget* is introduced to control the switch between the polling mode and the interrupt mode of the guest’s frontend network device. In **Step 6**, the `poll()` disables the interruptability of the frontend driver, and the interruptability is re-enabled in **Step 8** if the number of the packets processed in a given time-slice is less than the *budget*, or all the packets in the shared memory have been processed. Consequently, the interruptability of the frontend driver is disabled by the IH for a long time, which may introduce a challenge when it comes to the interrupt remapping mechanism.

### B. Scheduling over Interrupt Remapping

A VSMP guest possesses multiple vCPUs that multiplex on physical pCPU according to a global scheduling policy of the hypervisor scheduler. The status of a vCPU can be defined as *available* if it is currently running on a pCPU; otherwise the status is defined as *unavailable*. The time-sharing mechanism of the hypervisor scheduler constitutes the fact that the vCPU responsible for the I/O event may be descheduled/suspended by the hypervisor based on its scheduling policy. Consequently, a virtual interrupt delivered to an *unavailable* vCPU suffers from a prolonged latency, since it cannot be served in time until the targeted vCPU is rescheduled by the hypervisor scheduler.

To address this issue, the intelligent scheduling methods for interrupt remapping are proposed, such as vBalance [19], hBalance [20] and vINT [21]. The main idea behind these schemes are similar, which is redirecting the virtual interrupts from an *unavailable* vCPU to an *available* vCPU in the VSMP guest. This not only reduces I/O processing delay which is useful for “latency-sensitive” applications but also improves the I/O throughput which benefits “throughput-sensitive” applications. Since each vCPU has an opportunity to serve and process interrupts, the interrupt remapping mechanism can take full advantage of all the available vCPU resources for interrupt processing, which is very helpful in a VSMP guest with multiple vCPUs.

All of the above methods only try to choose an *available* vCPU for a better I/O responsiveness and throughput. However, the hypervisor scheduler is completely unaware of the interruptability information of guest’s network device, which may incur an novel IHP issue stated in the next subsection.

### C. Interruptability Holder Preemption (IHP) Challenge

As we mentioned before, the current high-performance network drivers employ polling mode with NAPI mechanism to avoid heavy interrupt processing overhead. In the NAPI mechanism, the interruptability of the guest’s network device is managed by the IH (vCPU), which is further subject to the global scheduling policy in the hypervisor. This introduces the IHP issue into the hypervisor-level interrupt remapping mechanism.

Figure 2 shows an example of the IHP challenge. Consider a VSMP guest possesses two vCPUs (vCPU0 and vCPU1), the vCPU0 disables the interruptability of the network device at time *T0* (**Step 6** in Figure 1) and becomes an IH. If it is preempted by other threads at time *T1*, the guest’s network I/O events will be suppressed until it regains the pCPU resource at time *T2* and re-enables the interruptability of the network device at time *T3*. This problem arises even though the vCPU1 is *available* (running on pCPU1) in *Tpreempt* (during time *T1* and *T2*). As a result, the interrupt remapping mechanism loses its efficiency in *Tpreempt* and an extra delay is imposed on the guest’s I/O latency. The IHP issue seriously disobeys the principle of the interrupt remapping mechanism, which is - “taking full advantage of the available vCPU resources to serve and process the VSMP guest’s I/O events”.

![Fig. 2. “Interruptability Holder Preemption” Challenge](image-url)

The root cause of the IHP issue is that the hypervisor scheduler is mostly agnostic to the interruptability of the network device in the guest. The loss of coordination between the hypervisor scheduler and guest’s network device leads to a situation where the I/O event processing of the guest, blocks on a specific vCPU (IH) which decreases the efficiency of the interrupt remapping mechanism. As a result, the guest still suffers from an inappropriate scheduling policy, which reflects both in high I/O latency and in degraded I/O throughput.

Therefore, more attention must be paid to make the hypervisor be aware of the interruptability of guest’s virtual device, and conduct scheduling decisions more precisely for the vCPUs of the VSMP guest. So the subsequent virtual interrupts can be successfully migrated from the *unavailable* vCPU to the *available* vCPU by means of interrupt remapping mechanism and the scheduling delay can be completely removed from the network I/O processing in the guest.
CoINT is a coordinator residing in the hypervisor. It is designed to wisely and elaborately schedule vCPUs with the awareness of the NAPI mechanism in the *paravirtual I/O* model, and can significantly enhance the network I/O performance under the VSMP environment. The basic idea of CoINT is intuitive - it empowers the hypervisor scheduler to be proactively and independently aware of the interruptability information of the guest’s network device. The proactivity of CoINT embodies in that it checks whether a vCPU is an IH or not before the descheduling, instead of waiting until the context switch happens and re-scheduling the IH, thus solving the IHP issue adequately and effectively.

### III. **CoINT Design**

As shown in Figure 3, CoINT consists of two major modules: *Interruptability Holder Detecting Module (IHD)* and *Proactive Interruptability-Aware Scheduling Module (PIAS)*. The IHD Module monitors and records the IH information in advance, thus providing necessary information for scheduler to keep awareness of the device’s interruptability. The PIAS Module performs the *deferred transitory scheduling* for the IH and the *instant transitory scheduling* for the non-IH, also taking the overall scheduling fairness into consideration, hence significantly removing the IHP issue from the guest’s I/O processing.

We describe these two modules in details below.

### B. **Interruptability Holder Detecting Module**

The *paravirtual I/O* model provides a mechanism which allows the frontend driver in guest and the backend driver in hypervisor to share a region of memory for the effective I/O processing. More specifically, the frontend driver negotiates a control flag, which is stored in the shared memory, with the backend driver to disable or enable the interruptability status.

As we discussed above, the vCPU in the guest OS takes the initiative to disable or enable the interruptability of the frontend driver, and there is no notification to the hypervisor after the interruptability information is transformed. So, a straightforward solution for the *Interruptability Holder Detecting Module* is to setup a semantic channel which enables the guest OS to notify hypervisor the first-hand IH information. However, this kind of semantic channel will involve not only costly hyper-calls caused by frequent guest-hypervisor switches but also multiple sys-calls with heavy modifications to the guest OS. Thus, for the most close-sourced commercial OSes, in-guest solution is less realistic and practical.

To be guest independent, this module primarily concerns one question: how to effectively detect the IH information in the hypervisor, with the requirements of accuracy and timeliness?

CoINT achieves the accurate IH detecting by using an on-demand approach: The interrupt targeted vCPU is detected as an IH when the hypervisor injects a virtual interrupt into the guest.

Figure 1 illustrates that the interruptability disabling of the guest’s frontend driver is closely tie to the virtual interrupt it receives. Once the frontend driver receives a virtual interrupt, it is the interrupt targeted vCPU executing the corresponding handler that instantly disables the interruptability of device and process subsequent I/O events in the polling mode. As a result, marking the interrupt targeted vCPU as guest’s IH has its theoretical correctness and can also achieve the requirements of the accuracy and effectiveness.

With this knowledge, once a virtual interrupt is redirected to the guest by the interrupt remapping framework in the hypervisor, CoINT gets the message and maintains a specialized data field associated with the guest to record the interrupt targeted vCPU, thus providing necessary information for *Proactive Interruptability-Aware Scheduling Module* to keep the device’s interruptability awareness.

### C. **Proactive Interruptability-Aware Scheduling Module**

Conventionally, the hypervisor scheduler only focuses on the fairly resource-sharing among the threads in the system and is mostly agnostic to the specific information of the VSMP guest, such as network device’s interruptability, which involves the above-mentioned IHP issue and decreases the efficiency of the interrupt remapping mechanism. In this module, we design a more detailed and wise vCPU scheduling method in the hypervisor to avoid the IHP issue.

**Deferred Transitory Scheduling.** With the IH information in *Interruptability Holder Detection Module*, CoINT proactively checks whether a vCPU should be descheduled. If a vCPU is an IH and the current interruptability of the frontend driver is disabled, CoINT gives this vCPU another running opportunity and marks it as a *favored IH*. This mechanism ensures that the IH can serve and process more available packets and then has a greater chance to re-enable interruptability of the frontend driver before being descheduled. But in case the current vCPU is not an IH or the current interruptability of the frontend driver is enabled, then it is descheduled as usual.

We expect the *deferred transitory scheduling* to achieve a better I/O performance than the native scheduling: It guaran-
tees that the IH has a higher running opportunity than the other vCPUs within a short time and endeavors to re-enable the interruptability of the guest’s frontend driver before being descheduled, thus significantly eliminating the IHP issue.

**Instant Transitory Scheduling.** One challenge of CoINT is to decide that, once a favored IH re-enables the interruptability of the guest’s network driver, how to timely inspect and deschedule it instantly to minimize the fairness impacts introduced by the deferred transitory scheduling.

Intuitively, we can use a polling approach in the hypervisor, where a thread running on a dedicated pCPU core regularly inspects the interruptability information, with a predefined frequency. The polling approach can ensure the hypervisor acquire then accurate interruptability information of guest’s network device. However, running it needs a well-designed frequency be set and at least one pCPU core be dedicated, which degrades the flexibility, manageability and resource utilization of the entire system. Therefore, the polling approach is sub-optimal under a real cloud environment.

Instead, CoINT settles this challenge by implementing a para-polling approach in a light-weight manner. More specifically, CoINT takes advantages of VM exits in Intel VT-x technology to timely check and inspect the interruptability information of guest’s frontend network device.

In virtualized system, the hypervisor deals with all VM exits with the specific exit reasons recorded in a common exit handler. Existing researches [3], [13] have evaluated that the VM exits of the virtualized system are with high frequency, which is typically over 10,000 exit/s under an I/O intensive workload. Therefore, the VM exits can be considered as an off-the-shelf polling mechanism [27] and it is beneficial to conduct the interruptability information checking in the exit handler.

In CoINT, once the guest’s exit handler detects that the interruptability of its network device is enabled, it firstly clears the recorded IH of the guest and then instantly invokes the scheduling function of the hypervisor scheduler to deschedule the vCPU if it is a favored IH, thus minimizing the impacts on other threads. Note that a considerable number of cycles are already spent on handling VM exits, so the cost of appending checking the interruptability information of guest’s network device is trivial and negligible. Based on the fact that the frequency of the VM exit is much larger than the scheduling frequency of the hypervisor, this kind of mechanism can prevent an IH occupying the pCPU resource too long as much as possible.

**Scheduling Fairness Discussion.** Notice that simply offering extra running opportunities to the IH may easily undermine and break the fairness of the hypervisor scheduler, and even cause the starvation to other threads. As a result, the Proactive Interruptability-Aware Scheduling Module also takes the fairness into consideration and furnishes a fine-grained scheduling policy.

On the one hand, the maximum continuous running time of each favored vCPU also is restricted. This can be achieved by introducing a specific data field and setting a predefined value according to the network workload, thus ensuring that the other threads in the system can run normally. On the other hand, the hypervisor scheduler continually monitors the running time and automatically adjusts the running priority of each thread, which means that each favored IH will be allocated less CPU cycles in the next scheduling periods because it achieves more CPU cycles than normal. Therefore, the Proactive Interruptability-Aware Scheduling Module will not compromise the fairness of hypervisor scheduler in a long period.

**IV. Implementation**

We have implemented a prototype of CoINT in the KVM hypervisor with Linux kernel 4.2.1. The idea of CoINT is generic enough and can be applied to many other hypervisors, such as the Xen and VMware.

For paravirtual I/O network devices, the KVM provides two types of implementation: (1) emulated by qemu [28] in user space (qemu-virtio), and (2) implemented as a module in the kernel (vhost-net). We base the implementation of CoINT on vhost-net, which has a higher performance than the qemu-virtio.

**A. Interruptibility Holder Detecting Module Implementation**

The main modification is in struct kvm. Each struct kvm is related to a VM and is accessed throughout the whole runtime of the KVM hypervisor. So it is natural that we put the IH information into kvm and provide it to KVM CFS to check at the runtime. We add a data field (inta_holder) to kvm to trace the vCPUID of the current IH. For the interruptability information detecting, we modify the kvm_set_msi() function to save the targeted vCPU in inta_holder after a virtual interrupt is injected into the guest.

Normally, the virtio device in the guest OS turns its interruptability ON or OFF by negotiating a control flag with its vhost-net device residing in the hypervisor. So, the second modification is to add the address of the virtio device’s interruptability flag in struct kvm, which is a data field named (inta_addr) with an unsigned integer type. This involves a small change to the function vhost_dev_init(), where the guest’s vhost-net device is initialized and we need to record the address value of the interruptability flag in inta_addr.

**B. Proactive Interruptability-Aware Scheduling Implementation**

Linux kernel adopts a Completely Fair Scheduler (CFS) as the default scheduler and provides a fair time-sharing pCPU resource mechanism for threads in the function schedule(). Each thread is scheduled by CFS in the form of a struct task_struct, each of which is related to a struct pid. So, in order to achieve the interruptability-aware scheduling, it is the basic to identify the vCPU threads in the function schedule(). The KVM assigns a data structure named struct kvm_vcpu to each vCPU, in which the corresponding pid is stored. The linux kernel also provides a function named get_pid_task() which can be used to acquire the task_struct corresponding to
Algorithm 1: Proactive Interruptability-Aware Scheduling Algorithm

1: \texttt{inta\_holder}, the interruptability holder reserved at the moment;
2: \texttt{inta\_addr}, the address of the interruptability flag;
3: \texttt{acc\_array}, the number of extra running times each vCPU has been given continuously at the moment;
4: \texttt{threshold}, the maximum number of extra continuously running times for each vCPU;
5: \texttt{cur\_task}, the current running task;
6: \texttt{next\_task}, the next running task;
7: \texttt{cur\_vcpu} = \texttt{cur\_task}\_vCPU\_id
8: \textbf{if} \texttt{cur\_vcpu} == \texttt{inta\_holder} \&\& \* \texttt{inta\_addr} == false \textbf{then}
9: \hspace{1em} \texttt{next\_task} \leftarrow \texttt{cur\_task};
10: \hspace{1em} \texttt{acc\_i} \leftarrow \texttt{acc\_i} + 1;
11: \textbf{else}
12: \hspace{1em} \texttt{next\_task} \leftarrow \texttt{pick\_next\_task};
13: \hspace{1em} \texttt{acc\_i} \leftarrow 0;
14: \textbf{end if}
15: \textbf{else}
16: \hspace{1em} \texttt{next\_task} \leftarrow \texttt{pick\_next\_task};
17: \textbf{end if}
18: Schedule \texttt{next\_task} to run;

A certain \textit{pid}. With the one-to-one relationship among these data structures, we add a function named \texttt{get\_task\_vCPU()} to get the corresponding vCPU ID of a \texttt{task\_struct}. For the \texttt{task\_struct} which is not associated to a vCPU thread, the function \texttt{get\_task\_vCPU()} returns -1;

Once the vCPU thread is identified, CoINT implements the Proactive Interruptability-Aware Scheduling as Algorithm 1.

To achieve the proactivity of CoINT, the work of identifying whether a vCPU is an IH or not is conducted before the descheduling, based on the prepared information in \texttt{inta\_holder}. As mentioned in Section III-C, an array \texttt{acc\_array} is added to \texttt{struct kvm} to keep track of the extra running times that each vCPU has been continuously awarded in the past period. When one vCPU gets the extra running times, the corresponding bit of \texttt{acc\_array} is updated. Considering different network workloads, we add a parameter to \texttt{vhost-net} called \texttt{threshold} to dynamically set the maximum continuously extra running times when initializing the \texttt{vhost-net}. When the continuously extra running times of an IH reaches to the \texttt{threshold}, the default scheduling algorithm in the function \texttt{pick\_next\_task()} is invoked to schedule another thread to run as usual.

For the \textit{instant transitory scheduling}, we modify the function \texttt{vmx\_handle\_exit()} to invoke the \texttt{schedule()} function if the \texttt{inta\_holder} has been awarded extra running times and the interruptability flag stored in the \texttt{inta\_addr} is true (re-enabled), thus instantly descheduling it and keeping the whole system fair. The corresponding bit of the \texttt{acc\_array} in \texttt{struct kvm} is also cleared.

V. Evaluation

Our experiments are conducted on two Dell PowerEdge R730 servers, connected by two Intel Ethernet Controller X710 10GbE NICs. Each server is equipped with an 8-core Intel Xeon E5-4610 v2 CPU, 32GB physical memory and two 250GB SATA hard disk drives. We use a 64-bit KVM hypervisor with Linux kernel 4.2.1 and guest OS as Ubuntu 14.04.1. The VSMP guest under test is configured with 4 vCPUs and 2GB memory. For the \textit{paravirtual I/O} setting, the VSMP guest’s virtual NIC is bridged to the host’s physical NIC with macvtap.

We measured the performance in three scenarios:

- **KVM-native**: the original KVM hypervisor as baseline.
- **Interrupt Remapping (IntR)**: with simple interrupt-remapping deployed in the KVM hypervisor.
- **CoINT**: CoINT over simple IntR with IHP avoidance.

To verify the CoINT performance benefits obtained from IHP-avoidance, we conduct comprehensive experiments with micro-benchmarks (Ping, Netperf) and macro-benchmarks (ApacheBench and Httperf). Also, we evaluate the system overhead of CoINT with Syssbench to prove the negligible overhead of CoINT.

A. Micro-level Benchmarks

This section evaluates the network throughput and latency using the micro-benchmarks. Moreover, the tested result also illustrates that CoINT only introduces an negligible and reasonable overhead.

To create an extreme pCPU-sharing case, we pinned the four vCPUs of the tested guest on one physical core and see how much improvement CoINT can attain. We also run the \texttt{lookbusy} tool in the tested guest to maintain the CPU load at an expected level.

1) Network Latency: We use Linux’s \texttt{ping} to validate the networking response time of the VSMP guest, by continuously pinging the guest from another server every 0.4 second to evaluate the round-trip time during one minute. Figure 4 shows the tested RTT (Round-Trip Time) results. From Figure 4(a) we can see that in KVM-native, the ping RTT varies significantly between 1.71ms and 18.6ms. While, in Figure 4(b), IntR reduces the ping RTT to less than 4.05ms, but it still suffers from the large variations. In Figure 4(c), CoINT can effectively cut down the RTT to less than 1.5ms with a very small variance and achieves a 92% and a 67% decrease compared with KVM-native and IntR, respectively. The tested results can be concluded that vCPU scheduling latency involves great network I/O processing delay in guest, and IntR provides a considerable performance improvement by redirecting the virtual interrupts to an available vCPU. Furthermore, CoINT shows that it can lead to great performance advantages by solving the IHP issue with the help of Proactive Interruptability-Aware Scheduling, in a more competitive way.

2) Network Throughput: We use Netperf benchmark to verify the performance advantage of CoINT in terms of network throughput. A \texttt{netserver} process is running on the
VSMP guest under test, and a Netperf process running on another server sends periodic network requests to it.

![Figure 4. Ping RTT Test Results](image)

Figure 5 shows the throughput performance tested with TCP_STREAM, UDP_STREAM, TCP_RR and UDP_RR of Netperf. The performance results are obtained with the average value of ten repetitions, and each test lasts for 30 seconds. Figure 5(a) shows TCP_STREAM test. When TCP_STREAM with 512B packet size, CoINT’s throughput achieves up to 1200Mbps, while the throughput of KVM-native is only less than 600Mbps and the throughput of IntR is less than 1000Mbps. This is because CoINT not only can avoid the vCPU scheduling latency but also can significantly eliminate the IHP issue. When sending 64B packets in TCP_STREAM, CoINT achieves a network throughput of 890Mbps+, which is almost 3x throughput of KVM-native and 1.3x throughput of IntR. From the results, we can conclude that CoINT’s performance benefit is more significant with small packet size case, where more interrupts are generated to the guest and the guest’s network device has a higher possibility to enter the polling mode. CoINT also shows similar performance benefits over the KVM-native and IntR with UDP_STREAM test in Figure 5(b).

Figure 5(c) and Figure 5(d) present the results of TCP_RR and UDP_RR with varying sizes of request and response messages. See that along with the size of response packet increases, the number of transactions per second decreases. CoINT always outperforms both KVM-native and IntR for all different request and response packet sizes. Additionally, the transaction rate of CoINT reaches its peak when the combination of request/response is set to 256B/1K, for both TCP_RR and UDP_RR tests. In TCP_RR with 64B/1K case, CoINT performs almost 2.3x throughput compared with the KVM-native and provides an 1.4x throughput compared with the IntR. In UDP_RR, the maximum improvement of CoINT is obtained at the 64B/4K test case, which is 320% of KVM-native and 160% of IntR. Hence, CoINT can substantially benefit the performance of a real web application, where the network request packets from the clients are usually small while the network response packets from the server can be quite large.

3) Overhead: To evaluate the overhead of CoINT compared to KVM-native and IntR, we use Sysbench tool to test the performance of whole system in terms of CPU, Memory, Threads and Mutex. We also use netperf program to generate a certain amount of network load for the tested VM. Sysbench is a modularized, cross-platforms and multi-threads program which can be used to evaluate different system parameters. CPU scenario does simple prime number calculation. Memory scenario does sequential memory reads and writes operations. Threads test simulates the case with high concurrent threading. Mutex test simulates multiple threads running concurrently to request the mutex at the same time.

Table I shows the evaluation results of the total time spent in these varying testing scenarios. In CPU scenario, CoINT only introduces 0.7% and 0.2% extra time consumption compared with KVM-native and IntR. The total time spent in sequential memory reads and writes increases by 1.1% in CoINT than in KVM-native, while the extra time spent can be reduced by 0.6% compared with IntR. For Threads and Mutex tests, CoINT also shows its low time-consumption and high adaptability. Overall, the Sysbench evaluation results proves that even in an extreme pCPU-sharing case, CoINT is a lightweight and side-effect-free solution in the hypervisor.

B. Macro-level Benchmarks

We also use two application-level macro benchmarks, ApacheBench and Httperf, to validate the performance im-
This makes the Proactive Interruptability-Aware Scheduling Module in CoINT can sufficiently keep the interruptability awareness and effectively eliminate the IHP issue.

**Reply Time** is shown in Figure 6(b), which represents the time taken for the Apache Server to respond and for a remote client to receive the reply. As shown, the reply time of KVM-native raises linearly when the request rate exceeds 2200 request/s, and reaches 700 ms when the request rate is increased to 3000 request/s. For IntR, the reply time starts to largely increase when the request rate exceeds 2400 request/s, but is always lower than that of KVM-native. The reply time of CoINT increases gradually when the request rate is higher than 2000 request/s, but never exceeds 200 ms. We can also see that the maximum reduction in CoINT can be observed at 3000 request/s case. At this point the reduction is 75% compared to KVM-native and 60% compared to IntR.

**Reply Rate** indicates the number of reply received from the Apache Server at every second. From Figure 6(c), we can see that the reply rate for all of these three settings have a similar rising trend. The KVM-native’s reply rate starts to drop as the request rate arrives at 2200 request/s and due to a large delay and request overflow, reaches its bottom at a request rate of 2600 request/s. The reply rate of IntR also displays a downward trend at 2400 request/s and always outperforms that of KVM-native. CoINT’s reply rate has the most enduring growth and reaches its peak at 2600 request/s, with a value of 3497.1 reply/s. The result shows that CoINT has a better tolerance to the request rate than KVM-native and IntR, for the reason that it not only avoids the vCPU scheduling latency but also eliminates the IHP issue.

2) **ApacheBench performance**: ApacheBench is a program for benchmarking the performance of HTTP servers. We run an Apache server on the tested VM, and run ApacheBench tool on another server to repeatedly request for static pages to the guest.

Figure 7 shows the evaluation results of the time taken per request with a varying number of requests (1K–10K). To simulate real network conditions, we configure the number of concurrent clients to be 500. Compared with KVM-native, CoINT reduces the time for processing each request from...
larger than 1.6ms to less than 0.6ms, with 0.422ms peak reached at the 1K case. For IntR, CoINT also achieves a 46% performance improvement, which reaches at the 3K test case. The Apachebench testing results prove that for all tested configurations, CoINT has the highest capacity to handle the client load.

VI. CONCLUSION

In this work, we first analyze the problem of I/O responsiveness and performance in the the VSM environment, and demonstrate that interrupt remapping mechanism in the hypervisor is an efficient and general method for improve network I/O performance. In this context, we innovatively identify the interruptibility holder preemption (IHP) problem in the paravirtual I/O model that significantly degrades the system responsiveness and throughput. To solve the IHP problem, we propose a proactive coordinator, CoINT, to provide a substantial lightweight solution. Furthermore, we design and implement the prototype of CoINT in KVM and conduct a series of experiments using micro and macro-level network-intensive benchmarks. The experimental results verify that CoINT can significantly eliminate the IHP issue and greatly enhance the I/O performance in the VSM environment.

ACKNOWLEDGEMENT

This work was supported in part by the National Key Research & Development Program of China (2016YFB1000502), National Natural Science Funds for Distinguished Young Scholar No.61525204.

REFERENCES