Responsive and Enforced Interrupt Handling for Real-Time System Virtualization

Hyoseung Kim*, Shige Wang†, Ragunathan (Raj) Rajkumar*
*Carnegie Mellon University
†General Motors R&D
hyoseung@cmu.edu, shige.wang@gm.com, raj@ece.cmu.edu

***Abstract*—The increasing performance of modern processors makes virtualization a viable solution for consolidating real-time systems into a single hardware platform. Although real-time task scheduling in a virtual machine can benefit from hierarchical scheduling, unbounded interrupt handling time and vulnerability to interrupt storms make practitioners hesitant to virtualize interrupt-driven real-time applications. In this paper, we propose vINT, an interrupt handling scheme designed for real-time system virtualization. vINT provides a pseudo-VCPU abstraction dedicated for interrupt handling, which overcomes the limits imposed by the timing parameters of virtual CPUs in an analyzable way. vINT also accounts for and enforces interrupt handling and resulting execution flows within a guest virtual machine. vINT does not require any change to the guest OS code, so it can be used for virtualizing proprietary, closed-source OSs. We analyze interrupt handling time as well as VCPU and task schedulability, with and without vINT. Our experimental results indicate that vINT achieves timely interrupt handling while providing as good task schedulability as when it is not used. Our case study based on a prototype implementation on the KVM hypervisor shows that vINT yields significant benefits in reducing interrupt handling time and in protecting real-time tasks against interrupt storms permeating into the virtual machine.**

I. Introduction

Virtualization has gained significant interest as an appealing solution for the consolidation of individually-developed, complex real-time embedded systems into a single hardware platform. Each system to be consolidated is provided a virtual machine (VM) that can maintain its own software infrastructure, such as an OS, middleware and libraries, logically isolated from other VMs. Many legacy real-time applications can be reused without porting them to a new OS by using virtualization. The flexibility of hosting multiple OSs under virtualization allows real-time OSs to co-exist with general-purpose OSs on the same hardware platform.

In general, virtualization has a two-level hierarchical scheduling structure. Each guest VM has one or more virtual CPUs (VCPUs). The tasks of a VM are scheduled on the VCPUs of that VM by the guest OS scheduler. Each VCPU is a schedulable entity in the hypervisor, analogous to a thread in a non-virtualized OS. Hence, the hypervisor schedules VCPUs on physical CPUs. The scheduling of real-time tasks in a hierarchical structure has been extensively studied in the context of real-time systems [12, 30, 31, 32]. These real-time hierarchical scheduling theories have also been applied to many open-source virtualization platforms, e.g., Xen [37, 38], KVM [10, 17], and L4/Fiasco [39].

Interrupt handling and resulting execution flows are indispensable for many real-time systems that interact with the physical environment in a lower latency compared to polling. In a virtualized system, a *physical* interrupt generated by a sensor or network interface is first handled by the interrupt service routine (ISR) of the hypervisor, and then delivered to the corresponding VCPU in the form of a *virtual* interrupt. Once that VCPU is scheduled, the virtual interrupt is handled by the ISR of the guest OS while consuming the VCPU's budget. Finally, the interrupt triggers the execution of any task responsible for reacting to that interrupt.

We have identified three strong requirements for the interrupt handling scheme of real-time system virtualization:

R1. Providing responsive and bounded interrupt handling time while ensuring task schedulability: Under virtualization, interrupt handling time may become excessively long due to the preemption and budget depletion of a VCPU. Arbitrarily increasing VCPU priority or budget is not a good option, since it may adversely affect the schedulability of tasks in other VCPUs.

R2. Enforcing virtual interrupts to protect real-time tasks from interrupt storms[1]: Unlike a native system, a virtualized system may suffer from virtual interrupt storms. The negative impact of virtual interrupt storms can be much significant than that of physical interrupt storms because a VCPU typically has a fraction of physical CPU time as its budget when hosted together with other VMs.

R3. Supporting unmodified guest OSs: One of the primary purposes of using virtualization is to co-host multiple VMs using different OSs. If an interrupt handling scheme requires modifications to a guest OS, its applicability is limited to only VMs using OSs that are open-source, which is not the case for many commercial real-time OSs.

However, previous work on interrupt handling in virtualization [6, 18, 20, 24] does not satisfy all these requirements. Table I gives a comparative summary of previous work.

In this paper, we propose vINT, an analyzable interrupt handling scheme to address the aforementioned requirements in real-time system virtualization. vINT provides responsive, bounded, and enforced interrupt handling *without* making any change to the guest OS code. Hence, it can be easily applicable to *full virtualization* scenarios hosting unmodified, proprietary guest OSs.[2] We analyze interrupt handling time in a virtualized environment with and without vINT. We also provide analyses on the schedulability of VCPUs and tasks in the presence

[1] An interrupt storm is a condition where a system receives interrupts at an unexpectedly high rate and the processing of those interrupts takes the majority of the CPU time. It is also known as the receive livelock problem [26].

[2] Full virtualization is a technique to host an unmodified guest OS in a privilege level lower than the hypervisor. Under full virtualization, any privileged instruction in the guest OS, such as disabling interrupts, is trapped and emulated by the hypervisor to protect the hypervisor itself and other VMs.

TABLE I: Comparison with previous work

Schemes	Priority based sched.	VCPU temporal isolation	Bounded interrupt handling	Enforced interrupt handling	Task sched. analysis	Unmod. guest OS
[6]			X	X		X
[18]	X	X				
[20]	X	X				
[24]	X	X				
Our work	X	X	X	X	X	X

of physical and virtual interrupts. Our experimental results indicate that vINT achieves timely interrupt handling while providing as good task schedulability as when it is not used. We have implemented a prototype of vINT on the KVM hypervisor (chosen for convenience). Our case study using this implementation shows the benefits of vINT in providing responsive interrupt handling times and protecting real-time tasks against virtual interrupt storms.

The rest of this paper is organized as follows. Section II reviews related work. Section III describes the system model used. Section IV explains the problems with interrupt handling in virtualization. Section V presents our proposed vINT scheme. Section VI shows our analyses on interrupt handling time, and VCPU and task schedulability. Section VII provides detailed evaluation, and Section VIII concludes the paper.

II. Related Work

The issues of responsive and enforced interrupt handling have been intensively studied in the context of non-virtualized systems. Previous work commonly uses a split interrupt handling model to execute deferrable work within a task context [21, 25, 34]. Specifically, Zhang and West [40] proposed the Process-Aware Interrupt (PAI) mechanism that schedules and accounts Linux bottom halves with the highest priority of the tasks waiting on the corresponding interrupt. Palmer and West [28] proposed to use deferrable servers to handle interrupts in order to minimize the receive livelock problem [26]. Danish et al. [11] proposed a Priority Inheritance Bandwidth-Preserving (PIBP) policy to handle interrupts and I/O requests with the budget and priority of the associated task. All these schemes, however, are designed for a non-virtualized system and cannot address the problems of virtual interrupt handling in a virtualized system, which will be discussed in Section IV.

There also exist many research efforts attempting to address other aspects of interrupts in a non-virtualized system. Leyva-del-Foyo et al. [22] proposed an integrated task and interrupt management model. By using a very short ISR that only activates a task corresponding to the interrupt, the proposed model could reduce the interference from interrupts associated with lower-priority tasks. Elliott and Anderson [13] focused on the priority inversion problem caused by the interrupts of GPU asynchronous I/O in a multi-core system using global scheduling. Brandenburg et al. [8] investigated various interrupt accounting mechanisms for multi-core systems using global EDF scheduling.

The virtualization of real-time systems has gained much attention recently. RT-Xen [37, 38] provides a hierarchical scheduling framework in the Xen hypervisor. The latest version of RT-Xen supports both partitioned and global scheduling at the hypervisor and provides a deferrabler server and a periodic server for the VCPU budget replenishment policy. Yang et al. [39] implemented a hierarchical scheduling framework on the L4/Fiasco microkernel-based hypervisor [1]. Bruns et al. [9] evaluated the real-time properties of L4/Fiasco.

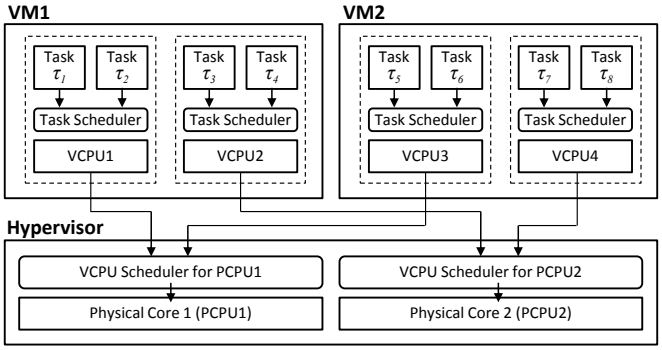


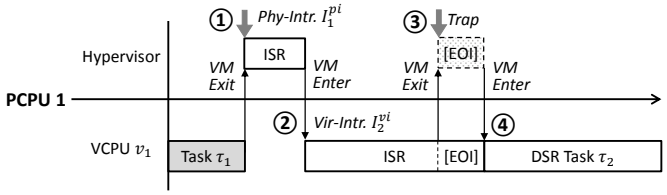
Fig. 1: Partitioned hierarchical scheduling structure

To overcome the limitations of hierarchical scheduling in real-time virtualization, approaches based on *paravirtualized scheduling* [18, 20, 24] have been studied. All of these approaches require modifications to the scheduler of a guest OS to let the hypervisor know the currently-executing task within the VM. Using this information, the hypervisor increases the priority of the corresponding VCPU so that the VCPU is not preempted by other VMs executing lower-priority tasks. However, none of these approaches bounds the worst-case interrupt handling time. They also do not enforce virtual interrupt handling. Specifically, the work in [20] proposes to assign a separate budget and priority to a subset of tasks and interrupts of a VCPU, but does not consider virtual interrupt storms and does not show how the separate budget and priority values can be determined.

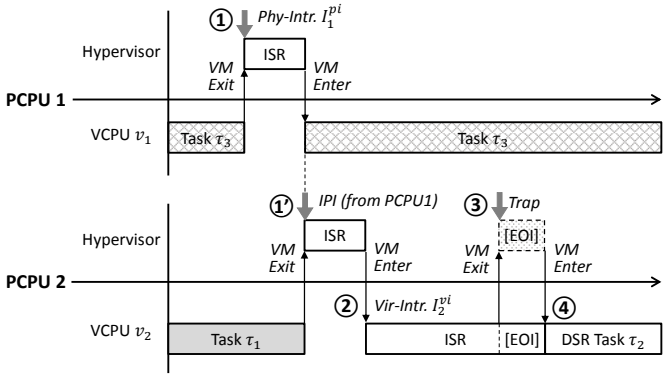
Beckert et al. [6] proposed an interrupt handling scheme that can be implemented without guest OS modifications. However, their approach has several limitations: (i) the hypervisor is assumed to use TDMA to schedule VCPUs, which does not conform to the latest research efforts on real-time system virtualization, (ii) virtual interrupts may be handled while consuming the budgets of unrelated other VCPUs, meaning that each VCPU is not guaranteed to use its assigned budget for its own purpose, and (iii) task schedulability in the presence of virtual interrupts is not considered. In this paper, we have addressed these limitations.

III. System Model

We consider a system that has one or more physical CPU cores (PCPUs), each running at the same fixed clock frequency. The system runs a hypervisor capable of hosting guest VMs in full-virtualization mode. Each VM has one or more VCPUs, each of which appears as a processing core in the VM. The system has a two-level hierarchical scheduling structure, as shown in Figure 1. The tasks of each VM are scheduled on these VCPUs by the guest OS of the VM. The VCPUs are scheduled on PCPUs by the hypervisor. In this work, we focus on *partitioned fixed-priority preemptive scheduling* for both the hypervisor and the VMs due to its wide support in real-time hypervisors and OSs, such as OKL4 [4] and PikeOS [5]. Therefore, each VCPU is statically assigned to a single PCPU and each task is statically assigned to a single VCPU. Any fixed-priority assignment, such as Rate-Monotonic [23], can be used for both VCPUs and tasks.



(a) Interrupt handling on the same PCPU



(b) Interrupt handling across two PCPUs

Fig. 2: Interrupt handling in virtualization

VCPUs and Tasks: The resource requirement of a VCPU v_i is represented by two parameters as follows:

$$v_i := (C_i^v, T_i^v)$$

where,

- C_i^v: the maximum execution budget of VCPU v_i
- T_i^v: the budget replenishment period of VCPU v_i

Any application task or OS code can execute only if the corresponding VCPU has a non-zero remaining budget. For the VCPU budget replenishment policies, we consider the *deferrable server* [35] and *sporadic server* [33] variants, because they have been extensively studied in the literature on real-time hierarchical scheduling [12, 30, 32] and virtualization [17, 37, 38]. Specifically, under the deferrable server policy, the budget of a VCPU is fully replenished at the start of every replenishment period, resulting in a release jitter equal to $T_i^v - C_i^v$ [7]. Under the sporadic server policy, only the amount of budget used is replenished T_i^v units after the start of the use of that amount, yielding zero release jitter.

We assume sporadic tasks with implicit deadlines. Each task has a unique priority within its VCPU, which can be easily achieved by an arbitrarily tie-breaking rule. Task τ_i is represented as follows:

$$\tau_i := (C_i, T_i)$$

where,

- C_i: the worst-case execution time (WCET) of task τ_i
- T_i: the minimum inter-arrival time of task τ_i

Interrupts: We consider two types of interrupts: *physical* and *virtual*. A physical interrupt I_i^{pi} is a signal issued from a hardware device to a PCPU. Each physical interrupt is assumed to be statically pinned to one PCPU, which can be easily done in software with the support of a programmable interrupt controller (PIC). When a PCPU receives a physical interrupt, the currently executing VCPU on that PCPU is halted and the corresponding ISR of the hypervisor is executed (Step ① in Figure 2). The CPU time usage of the ISR of a physical interrupt is accounted for as the hypervisor's usage, not as the halted VCPU's usage. Each physical interrupt has a unique priority π_i^{pi} determined by the PIC. The ISR of a lower-priority physical interrupt can be preempted by that of a higher-priority physical interrupt. The ISRs of physical interrupts are not preemptible by VCPUs. Therefore, a VCPU preempted by an ISR can only resume its execution when all ISRs have been completed. A physical interrupt I_i^{pi} is represented as follows:

$$I_i^{pi} := (C_i^{pi}, T_i^{pi})$$

where,

- C_i^{pi}: the WCET of the ISR of I_i^{pi}
- T_i^{pi}: the minimum inter-arrival time[3] of I_i^{pi}

The response time of a physical interrupt (or physical interrupt handling time) is the time from the arrival of the physical interrupt signal to the completion of the corresponding ISR.

A virtual interrupt I_i^{vi} is a software signal from the hypervisor to a guest VM, issued upon the completion of the ISR of a physical interrupt.[4] Each virtual interrupt is assumed to be statically pinned to one VCPU of a VM. If a target VCPU is located on a PCPU different from that of a physical ISR, e.g., a physical interrupt shared among multiple VCPUs, the delivery of a virtual interrupt from a physical ISR to the VCPU causes an inter-processor interrupt (IPI) that is an additional physical interrupt to notify a state change to the VCPU running on a different PCPU (Steps ①' and ② in Figure 2(b)). Otherwise, a virtual interrupt is immediately delivered to the corresponding VCPU (Step ② in Figure 2(a)). When a VCPU receives a virtual interrupt, the currently executing task in that VCPU is halted and the corresponding ISR of the guest OS is executed. Each virtual interrupt has a unique priority π_i^{vi} given by the emulated PIC. Within each VCPU, the ISRs of lower-priority virtual interrupts can be preempted by those of higher-priority virtual interrupts, and virtual ISRs are not preemptible by tasks. As in most CPU architectures, each ISR executes an End-Of-Interrupt (EOI) instruction at the end to notify the completion of the ISR to the PIC. As the EOI is a privileged instruction called by the guest OS, it is trapped and emulated by the hypervisor while consuming the budget of the corresponding VCPU (Step ③ in Figure 2). A virtual interrupt is *pending* if it has been injected to the corresponding VCPU but its ISR has not yet been completed. A virtual interrupt I_i^{vi} is represented as follows:

$$I_i^{vi} := (C_i^{vi}, T_i^{vi})$$

where,

- C_i^{vi}: the WCET of the ISR of I_i^{vi}
- T_i^{vi}: the minimum inter-arrival time of I_i^{vi}

We assume that a *split interrupt handling* model is used by the guest OS due to its the wide acceptance in both real-time and non-real-time OSs. Under split interrupt handling, the ISR

[3]Similarly to prior work [8, 22], the minimal inter-arrival time of an interrupt refers to a value expected or identified at design time. An interrupt unexpectedly arriving faster than that value may cause an interrupt storm at runtime.

[4]There might be some cases where virtual interrupts are generated as a result of polling at the hypervisor. Considering such a mixed use of interrupts and polling in real-time virtualization remains as future work.

of a virtual interrupt performs the minimum amount of work and activates zero or more tasks to execute a deferred service routine (DSR) in the task context (Step ④ in Figure 2). Hence, the priorities of DSRs can be easily configured in contrast to ISRs, and the majority of interrupt handling can be done with desired priorities. We use $\mathbb{D}(I_i^{vi})$ to denote the set of DSR tasks triggered by the ISR of a virtual interrupt I_i^{vi}. The minimum inter-arrival time of any task in $\mathbb{D}(I_i^{vi})$ is therefore equal to or greater than T_i^{vi}. The response time of a virtual interrupt (or virtual interrupt handling time) is the time from the arrival of the virtual interrupt to the completion of the corresponding ISR and DSR. Lastly, we denote the sum of the WCETs of the ISR and DSR of a virtual interrupt I_i^{vi} as:

$$\mathbb{C}_i^{vi} = C_i^{vi} + \sum_{\tau_j \in \mathbb{D}(I_i^{vi})} C_j$$

Definition 1. *An interrupt-triggered execution flow in a virtualized environment is the sequence of executions from the arrival of a physical interrupt to the completion of the ISR and DSR of the corresponding virtual interrupt.*

Definition 2. *The total interrupt handling time is the amount of time to complete the corresponding interrupt-triggered execution flow.*

Definition 3. *An interrupt-triggered execution flow is serviceable, if its total interrupt handling time does not exceed the minimum inter-arrival times of the corresponding physical and virtual interrupts.*

IV. Problems with Virtual Interrupts

The main difference between interrupt handling in virtualized and non-virtualized environments is the presence of virtual interrupts. In this section, we detail two major problems associated with virtual interrupts.

A. Timing Penalties to Virtual Interrupt Handling

Once a virtual interrupt is injected into a VCPU, it is handled by using the priority and budget of the VCPU. Virtual interrupt handling time is thus affected by the followings:

- *VCPU budget depletion*: When a virtual interrupt I_i^{vi} is delivered to a VCPU v_j, the budget of v_j might have been completely consumed by other tasks within v_j. Hence, the handling of I_i^{vi} may be delayed until the start of the next replenishment period of v_j.
- *VCPU preemption*: Although a VCPU v_j has an enough budget to handle a virtual interrupt I_i^{vi}, the handling of I_i^{vi} may be delayed by the execution of any task on higher-priority VCPUs that can preempt v_j.

B. Virtual Interrupt Storms

Previous work proposed to address interrupt storms in the context of non-virtualized systems [11, 21, 28] uses a dedicated aperiodic server, e.g., a deferrable server or a sporadic server, for interrupt handling. When the server budget is depleted, the associated interrupt is not handled until the start of the next replenish period of the server. By doing so, the impact of an interrupt storm on CPU time is limited to the amount of the budget assigned to the associated server.

While previous work can be applied to the hypervisor to address physical interrupt storms, it may not be used for virtual interrupt storms in a full-virtualization scenario, where an unmodified guest OS is used and it is unaware of being virtualized. In general, OSs measure the passage of time by reading and comparing two clock values, e.g., $t_1 - t_0 =$ elapsed time from t_0 to t_1. Under full virtualization, an unmodified guest OS can check the passage of *physical time* in this manner. However, the guest OS cannot use the same manner to check the passage of *virtual time*, which is the actual CPU time used by the guest VCPU. This is because the guest OS is unaware of when and how much VCPU-level preemptions are caused. In other words, when previous work is used for virtual interrupts under full virtualization, it may result in significant errors in the accounting of virtual interrupt handling.

Goals: In this work, our goals are twofold: (i) minimize and bound interrupt handling time in a virtualized environment, and (ii) account for virtual interrupt handling and protect real-time tasks from virtual interrupt storms without any modifications to the guest OS.

V. vINT Scheme

The problems with virtual interrupt handling described in Section IV are caused by the fact that a virtual interrupt is handled by the same VCPU as the one used by other regular tasks. Motivated by this, we propose vINT that can conceptually split virtual interrupt handling from the VCPU of regular tasks in an analyzable way, without modifying the guest OS code. vINT can be selectively used for a subset of virtual interrupts that cannot be serviced within their minimal inter-arrival times by default, or have a possibility of causing virtual interrupt storms. For convenience of explanation, we assume that all virtual interrupts are managed by vINT in Section V-A and V-B. In Section V-C we relax this assumption.

A. Pseudo-VCPU Resource Abstraction

vINT uses a *pseudo-VCPU* resource abstraction to represent the resource requirement of the ISR and DSR of a virtual interrupt as a separate VCPU to the hypervisor. The pseudo-VCPU differs from its original VCPU in that it does not have an execution context. In other words, the use of the pseudo-VCPU introduces no additional processing core visible to the guest VM, which is typically a high demand to host legacy guest OSs that may support only uniprocessors.

Each virtual interrupt can be exclusively associated with one pseudo-VCPU that is located on the same PCPU as its original VCPU. A pseudo-VCPU v_p is described by the same types of parameters as a regular VCPU: C_p^v and T_p^v. The replenishment period of a pseudo-VCPU v_p is equal to or greater than the minimum inter-arrival time of the associated virtual interrupt I_i^{vi}, i.e., $T_p^v \geq T_i^{vi}$. The budget C_p^v of a pseudo-VCPU v_p associated with a virtual interrupt I_i^{vi} is assigned as follows:

$$C_p^v = \left\lceil \frac{T_p^v}{T_i^{vi}} \right\rceil \mathbb{C}_i^{vi} \quad (1)$$

It is worth noting that, once a virtual interrupt is assigned its pseudo-VCPU, the budget of its original VCPU can be reduced because the virtual interrupt will be handled by using the budget of the pseudo-VCPU.

Prioritization of pseudo-VCPUs: One of our goals is to provide responsive interrupt handling time, which is challenging due to the VCPU-level preemption while handling a virtual interrupt. To achieve this goal, vINT prioritizes pseudo-VCPUs over regular VCPUs. The priority of a pseudo-VCPU v_p associated with a virtual interrupt I_i^{vi} is assigned a priority of $\pi_B^v + (\pi_o^v - 1) \cdot L_o + \pi_{\mathbb{D}}(I_i^{vi})$, where π_B^v is a base VCPU-priority level greater than that of any regular VCPU on the same PCPU, π_o^v is the priority of the original VCPU of I_i^{vi}, L_o is the number of priority levels for all DSR tasks in the original VCPU, and $\pi_{\mathbb{D}}(I_i^{vi})$ is the priority difference between the highest-priority DSR task of I_i^{vi} and the highest-priority DSR task among all DSR tasks in the original VCPU. With this approach, the pseudo-VCPU v_p is not preempted by any regular VCPU, and the relative priority ordering of DSRs within the same original VCPU are preserved.

B. Pseudo-VCPU Realization

As a pseudo-VCPU does not have an execution context, in its realization, the actual execution of the ISR and DSR of a virtual interrupt still happens within the execution context of their original VCPU. We now explain how vINT handles a virtual interrupt as if it was handled in its pseudo-VCPU.

DSR task priority adjustment: Since pseudo-VCPUs are assigned higher priorities than regular VCPUs, the executions of DSRs should not be preempted by regular tasks in the realization. vINT therefore statically adjusts the priority of each DSR task τ_j to $\pi_{B,v_o} + \pi_j$, where π_{B,v_o} is a base task-priority level greater than any regular task in the task τ_j's original VCPU v_o, and π_j is the original priority of τ_j. Note that this priority adjustment is not needed if the priorities of DSR tasks are already higher than those of regular tasks in the original VCPU. In addition, since even closed-source, proprietary OSs provide an interface to configure task priorities, the priority adjustment does not violate the requirement of full virtualization.

Virtual interrupt injection: vINT maintains a counter for each pseudo-VCPU to indicate the number of virtual interrupts that can be handled by the pseudo-VCPU at that moment. The maximum possible value of the counter for a pseudo-VCPU v_p associated with a virtual interrupt I_i^{vi} is given by $\lceil T_p^v / T_i^{vi} \rceil$. When a virtual interrupt is generated, vINT checks the counter value of the corresponding pseudo-VCPU. If the counter is greater than zero, the counter is decremented by one and the virtual interrupt is injected into its original VCPU. Otherwise, the injection of the virtual interrupt is delayed until the counter becomes greater than zero. The replenishment rule of the counter is similar to that of the VCPU budget. Under the deferrable server policy, the counter is fully replenished at the start of every replenishment period of the pseudo-VCPU. Under the sporadic server policy, the counter is replenished by one at the time when the budget is replenished.

Virtual interrupt handling: We first consider a non-nested interrupt handling scenario. When a virtual interrupt I_i^{vi} is injected, the original VCPU v_o should handle the ISR and DSR of I_i^{vi} by using the priority and budget of the corresponding pseudo-VCPU v_p. Hence, vINT immediately raises the priority of v_o to that of v_p, and let v_o use the budget of v_p for the amount of $\mathbb{C}_i^{vi}$. As the DSR tasks of I_i^{vi} have higher priorities than regular tasks, they are guaranteed to be executed as soon as the corresponding ISR finishes. When the VCPU v_o has consumed $\mathbb{C}_i^{vi}$ units of the budget of v_p, vINT restores the priority of v_o and lets v_o use its own budget afterwards. The ISR and DSR of I_i^{vi} may be finished earlier than $\mathbb{C}_i^{vi}$ and regular tasks may be executed while their VCPU is still using the budget and priority of the pseudo-VCPU of I_i^{vi}. However, this does not change the worst-case interference that can be imposed on other VCPUs.

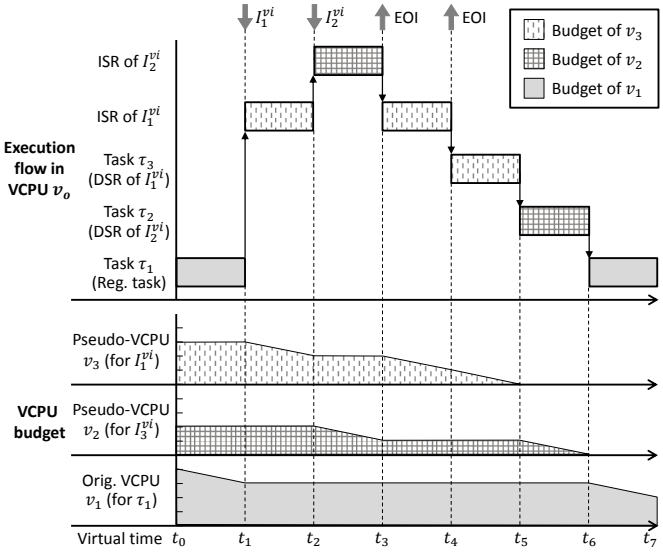


Fig. 3: vINT nested interrupt handling

We next consider a nested interrupt handling scenario. vINT exploits the following two factors to support nested interrupt handling with pseudo-VCPUs: (i) the hypervisor is aware of the set of all pending virtual interrupts in each VCPU, and (ii) the hypervisor traps an EOI instruction called at the end of each virtual ISR. When a new virtual interrupt is injected into a VCPU v_o, vINT lets v_o use the budget and priority of the pseudo-VCPU that is associated with the highest-priority virtual interrupt among all pending interrupts. This is because the VCPU executes the ISR of the highest-priority pending interrupt first. When the hypervisor catches an EOI from v_o, vINT checks if there is another pending interrupt. If so, vINT lets v_o use the budget and priority of the pseudo-VCPU of the higher-priority pending interrupt, and repeats this until there is no pending interrupt. If there is no pending interrupt, vINT now lets v_o use the budget and priority of the highest-priority pseudo-VCPU, the budget of which has not yet been used for the amount of $\mathbb{C}_i^{vi}$ by v_o to handle the injected interrupt I_i^{vi}. As the relative priorities of pseudo-VCPU follow those of DSR tasks, this approach makes the sequence of the pseudo-VCPU usage correspond to that of the DSR task executions. Figure 3 shows an example of nested interrupt handling with vINT. In this figure, the x-axis represents the passage of virtual time so the activities of the hypervisor and other VCPUs are omitted.

C. Selective Use of vINT

We now relax our assumption that all virtual interrupts are managed by vINT. If a virtual interrupt is not managed by

vINT, it is not associated with a pseudo-VCPU. The priorities of its DSR tasks remain unchanged. However, the presence of such an unmanaged virtual interrupt affects the pseudo-VCPU budgets of virtual interrupts managed by vINT. Consider a virtual interrupt I_i^{vi} associated with a pseudo-VCPU v_p. If there is any virtual interrupt not managed by vINT in the original VCPU of I_i^{vi}, the budget C_p^v of v_p is assigned by:

$$C_p^v = \left\lceil \frac{T_p^v}{T_i^{vi}} \right\rceil \left(\mathbb{C}_i^{vi} + \sum_{\substack{I_j^{vi} \in \mathbb{V}(I_i^{vi}) \wedge \\ pseudo(I_j^{vi}) = \emptyset}} \left\lceil \frac{T_i^{vi}}{T_j^{vi}} \right\rceil C_j^{vi} \right) \quad (2)$$

where, $\mathbb{V}(I_i^{vi})$ is the original VCPU of I_i^{vi}, and $pseudo(I_j^{vi})$ is a function returning the pseudo-VCPU of I_j^{vi} if exists, and $\emptyset$ otherwise. The second term in the parenthesis of Eq. (2) is an extra budget for the executions of the ISRs of virtual interrupts not managed by vINT. Since those ISRs may block the handling of I_i^{vi} in the realization, the extra budget allows the ISRs to be executed with the budget and priority of I_i^{vi}'s pseudo-VCPU. Therefore, when an instance of I_i^{vi} is injected into its original VCPU v_o, vINT lets v_o use the budget and priority of the pseudo-VCPU v_p for the sum of the terms in the parenthesis of Eq. (2), instead of only $\mathbb{C}_i^{vi}$.

VI. Analysis

In this section, we first analyze VCPU and task schedulability in the presence of physical and virtual interrupts. Then, we analyze interrupt handling time with and without vINT. For convenience, we use the following notation in this section:

- $\mathbb{P}(v_i)$ and $\mathbb{P}(I_j^{pi})$: PCPUs for a VCPU v_i and for a physical interrupt I_j^{pi}, respectively
- $\mathbb{V}(\tau_i)$ and $\mathbb{V}(I_j^{vi})$: Original VCPUs for task τ_i and for a virtual interrupt I_j^{vi}, respectively
- $pseudo(\tau_i)$ and $pseudo(I_j^{vi})$: Pseudo VCPUs for task τ_i and for a virtual interrupt I_j^{vi}, respectively, if exist; $\emptyset$ otherwise.

A. VCPU and Task Schedulability

The schedulability of a VCPU v_i can be determined by the following recurrence equation:

$$W_i^{v,n+1} = C_i^v + \sum_{I_u^{pi} \in \mathbb{P}(v_i)} \left\lceil \frac{W_i^{v,n}}{T_u^{pi}} \right\rceil C_u^{pi} + \sum_{v_h \in \mathbb{P}(v_i) \wedge \pi_h^v > \pi_i^v} \left\lceil \frac{W_i^{v,n} + J_h^v}{T_h^v} \right\rceil C_h^v \quad (3)$$

where, $W_i^{v,n}$ is the worst-case response time (WCRT) of a VCPU v_i at the n^{th} iteration ($W_i^{v,0} = C_i^v$), π_i^v is the priority of a VCPU v_i, and J_h^v is a release jitter ($J_h^v = T_h^v - C_h^v$ for the deferrable server policy and $J_h^v = 0$ for the sporadic server policy). Eq. (3) is based on the iterative response time test in [14]. It terminates when $W_i^{v,n+1} = W_i^{v,n}$, and the VCPU v_i is schedulable if its WCRT does not exceed its period, i.e., $W_i^{v,n} <= T_i^v$. In this equation, the second term represents the interference from the ISRs of physical interrupts during the execution of v_i.

For task schedulability, we need to consider virtual interrupts. If a virtual interrupt is managed by vINT, regular tasks do not experience any direct interference from that virtual interrupt because it is handled by using the budget of its pseudo-VCPU. On the other hand, if a virtual interrupt is not managed by vINT, it may be handled by the budget of the same VCPU as regular tasks. Hence, we can extend the task response-time test under hierarchical scheduling given in [30] as follows to check the schedulability of a regular task τ_i in a VCPU v_k:

$$W_i^{n+1} = C_i + \sum_{\substack{\tau_h \in \mathbb{V}(\tau_i) \wedge \pi_h > \pi_i \\ \wedge pseudo(\tau_i) = \emptyset}} \left\lceil \frac{W_i^n + J_h}{T_h} \right\rceil C_h + \left\lceil \frac{W_i^n + C_k^v}{T_k^v} \right\rceil (T_k^v - C_k^v) + \sum_{\substack{I_u^{vi} \in \mathbb{V}(\tau_i) \wedge \\ pseudo(I_u^{vi}) = \emptyset}} \left\lceil \frac{W_i^n + J_u^{vi}}{T_u^{vi}} \right\rceil C_u^{vi} \quad (4)$$

where, W_i^n is the WCRT of task τ_i at the n^{th} iteration ($W_i^0 = C_i$), π_i is the priority of τ_i, and J_h and J_u^{vi} are the release jitters of a task τ_h and a virtual interrupt I_u^{vi}, respectively ($J_h = J_u^{vi} = T_k^v - C_k^v$). It terminates when $W_i^{n+1} = W_i^n$, and the task τ_i is schedulable if its WCRT does not exceed its implicit deadline, i.e., $W_i^n <= T_i$. Note that the schedulability result for a task from Eq. (4) is valid only if the task's VCPU passes the VCPU schedulability test given in Eq. (3). The last summing term of Eq. (4) captures the interference from the ISRs of virtual interrupts that are not managed by vINT. In addition, since Eq. (4) conservatively assumes that the budget of the task's VCPU is available at the latest time possible within each period ($T_k^v - C_k^v$), the interference from physical interrupts does not need to be considered in Eq. (4).

B. Interrupt Handling Time

The total interrupt handling time can be bounded by the sum of (i) the WCRT of the ISR of a physical interrupt, (ii) the WCRT of the ISR of a physical IPI if the target VCPU is on a different PCPU, and (iii) the WCRT of the ISR and DSR of the corresponding virtual interrupt. For factors (i) and (ii), the WCRT of the ISR of a physical interrupt I_i^{pi} is bounded by the following recurrence equation:

$$W_i^{pi,n+1} = C_i^{pi} + \sum_{I_h^{pi} \in \mathbb{P}(I_i^{pi}) \wedge \pi_h^{pi} > \pi_i^{pi}} \left\lceil \frac{W_i^{pi,n}}{T_h^{pi}} \right\rceil C_h^{pi} \quad (5)$$

where, $W_i^{pi,n}$ is the WCRT of a physical interrupt I_i^{pi} at the n^{th} iteration ($W_i^{pi,0} = C_i^{pi}$), and π_i^{pi} is the priority of I_i^{pi}.

We now consider the last factor. When vINT is used, as shown in Figure 3, the ISR and DSR of a virtual interrupt may be blocked by the ISRs of virtual interrupts that are associated with lower-priority pseudo-VCPUs and executed in the execution context of the same original VCPU. The virtual interrupt may also be blocked by other virtual interrupts that are not managed by vINT. For a virtual interrupt I_j^{vi} associated with a pseudo-VCPU v_p, the maximum blocking time from such virtual interrupts during a time interval t is given by:

$$B_{p,j}(t) = \sum_{\substack{I_u^{vi} \in \mathbb{V}(I_j^{vi}) \wedge (pseudo(I_u^{vi}) = \emptyset \\ \vee \pi^p_{pseudo(I_u^{vi})} < \pi_j^p)}} \left\lceil \frac{t}{T_u^{vi}} \right\rceil C_u^{vi} \quad (6)$$

where, $\pi^p_{pseudo(I_u^{vi})}$ is the priority of I_u^{vi}'s pseudo-VCPU. In addition, the worst case happens when all physical interrupts

TABLE II: Base parameters for our experiments

Parameters	Values
Number of PCPUs	4
Number of VCPUs per PCPU	3
Number of physical interrupts per PCPU	6
Number of virtual interrupts per VCPU	2
VCPU replenishment period	10 msec
Minimum inter-arrival time of a physical interrupt	$[5, 10]$ msec
Minimum inter-arrival time of a regular task	$[100, 500]$ msec
WCET of ISR of a physical/virtual interrupt	$[5, 10]$ μsec
WCET of DSR of a virtual interrupt	$[10, 50]$ μsec
Number of regular tasks per VCPU	3
Number of DSR tasks per VCPU	2
Task set utilization per VCPU	10 %

on the same PCPU arrive with their minimum inter-arrival times and all higher-priority VCPUs fully consume their budgets. The WCRT of a virtual interrupt I_j^{vi} associated with a pseudo-VCPU v_p is therefore bounded by:

$$W_j^{vi,n+1} = \mathbb{C}_j^{vi} + B_{p,j}(W_j^{vi,n}) + \sum_{I_u^{pi} \in \mathbb{P}(v_p)} \left\lceil \frac{W_j^{vi,n}}{T_u^{pi}} \right\rceil C_u^{pi} + \sum_{v_h \in \mathbb{P}(v_p) \wedge \pi_h^v > \pi_p^v} \left\lceil \frac{W_j^{vi,n} + J_h^v}{T_h^v} \right\rceil C_h^v \quad (7)$$

where, $W_j^{vi,n}$ is the WCRT of a virtual interrupt I_j^{vi} ($W_j^{vi,0} = \mathbb{C}_j^{vi}$). Note that Eq. (7) is similar to the VCPU schedulability test given in Eq. (3), except the blocking term. This is because the pseudo-VCPU of a virtual interrupt is guaranteed to have enough budget to handle one instance of a virtual interrupt, and there is no other task interfering with the execution of the ISR and DSR of the virtual interrupt in the pseudo-VCPU.

When vINT is not used, the response time of a virtual interrupt should be captured by considering the executions of other tasks within the same VCPU. Therefore, the WCRT of a virtual interrupt I_j^{vi} in a VCPU v_k is bounded by:

$$W_j^{vi,n+1} = \mathbb{C}_j^{vi} + \sum_{\substack{\tau_h \in \mathbb{V}(I_j^{vi}) \wedge \pi_h > \check{\pi}_{\mathbb{D}} \\ \wedge pseudo(\tau_h) = \emptyset}} \left\lceil \frac{W_j^{vi,n} + J_h}{T_h} \right\rceil C_h + \left\lceil \frac{W_j^{vi,n} + C_k^v}{T_k^v} \right\rceil (T_k^v - C_k^v) + \sum_{\substack{I_u^{vi} \in \mathbb{V}(I_j^{vi}) \wedge u \neq j \\ pseudo(I_u^{vi}) = \emptyset}} \left\lceil \frac{W_j^{vi,n} + J_u^{vi}}{T_u^{vi}} \right\rceil C_u^{vi} \quad (8)$$

where, $\check{\pi}_{\mathbb{D}}$ is the priority of the lowest-priority task in $\mathbb{D}(I_j^{vi})$. Note that this equation is similar to Eq. (4) which captures the WCRT of a task.

VII. Evaluation

In this section, we first empirically investigate the performance characteristics and benefits of vINT, and then show its effects on a real hardware platform.

A. Experimental Setup

We consider the following schemes in our experiments: deferrable server without vINT (DSbase), sporadic server without vINT (SSbase), deferrable server with vINT (DSvINT), and sporadic server with vINT (SSvINT). We use randomly-generated task sets and interrupt sets to compare these schemes on how many task sets could be schedulable and how many interrupt sets could be serviced on a timely basis.

Since, in practice, vINT can be selectively applied to a subset of virtual interrupts that cannot be serviced within their virtual interrupt times by the baseline scheme, our experiments only focus on interrupts with short inter-arrival times. Table II lists the base parameters we use for our experiments. For each experimental setting, we first generate PCPUs, VCPUs, physical interrupts, and tasks and virtual interrupts for each VCPU based on the defined parameters. Each virtual interrupt is exclusively associated with one physical interrupt in a random manner, and the minimum inter-arrival time of each virtual interrupt is set equal to that of its associated physical interrupt. For each VCPU, the task set utilization per VCPU is split into k random-sized pieces, where k is the number of tasks per VCPU. The size of each piece becomes the utilization of the corresponding task, and the WCET of each task is calculated by dividing its utilization by its minimum inter-arrival time. For DSvINT and SSvINT, we create a pseudo-VCPU for each virtual interrupt with a period equal to the minimum inter-arrival time of the corresponding virtual interrupt and with a budget determined by Eq. (2). VCPUs and tasks are assigned unique priorities by using the Rate-Monotonic approach [23], with an arbitrary tie-breaking rule. The priorities of physical and virtual interrupts are assigned randomly. Once this is done, we finally determine the VCPU budget value for each scheme. Starting from a value equal to the VCPU period, the VCPU budget for each scheme is decreased by 1 μsec until all VCPUs pass the VCPU schedulability test given in Eq. (3).[5]

We generate 10,000 task sets and 10,000 interrupt sets for each experimental setting. The metrics used are: (i) the percentage of schedulable task sets where all tasks pass the schedulability test given in Eq. (4), and (ii) the percentage of serviceable interrupt sets where all interrupt-triggered execution flows are serviceable, checked by Eq. (5), (7) and (8).

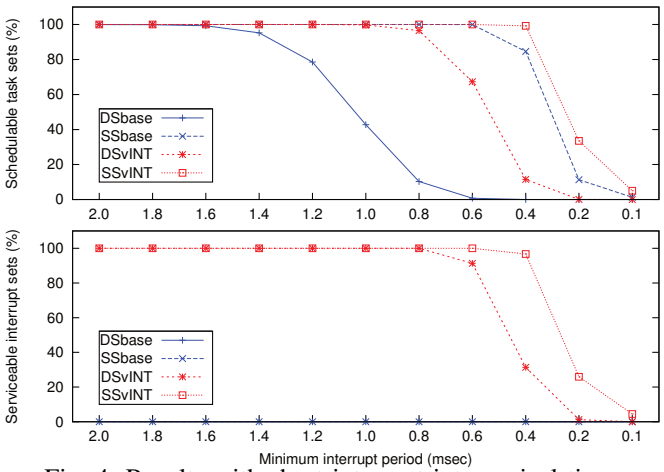


Fig. 4: Results with short interrupt inter-arrival time

B. Results

We explore three main factors that affect task schedulability and interrupt serviceability in a virtualized environment: (i) the minimum inter-arrival time of interrupts, (ii) the VCPU period, and (iii) the WCET of interrupt handlers.

Minimum inter-arrival time of interrupts: Figure 4 shows the percentages of schedulable task sets and serviceable in-

[5]Considering the time-unit granularity used in Table II, the step size of 1 μsec is fine-grained enough to find the maximum-possible VCPU budget for each scheme in our experiments.

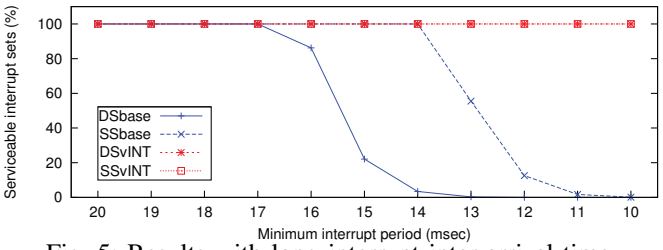


Fig. 5: Results with long interrupt inter-arrival time

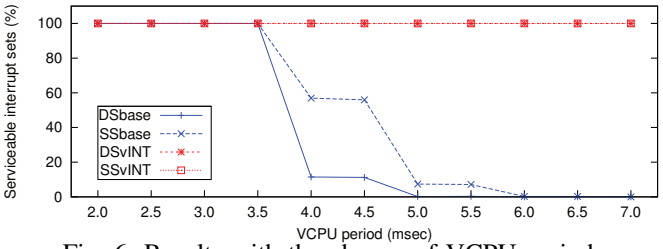


Fig. 6: Results with the change of VCPU period

terrupt sets as the minimum inter-arrival time of interrupts decreases. Each point k on the x-axis represents that the minimum inter-arrival time of each interrupt ranges $[k, k + 0.5]$ msec. In general, the sporadic server policy (SS) performs better than the deferrable server policy (DS). This is because SS has zero release jitter and allows assigning larger budget values to VCPUs than DS. vINT has benefits in both task scheduling and interrupt handling. DSvINT and SSvINT schedule more task sets than DSbase and SSbase, respectively. Especially, when the range is $[0.6, 1.1]$ msec, DSvINT schedules 67% more task sets than DSbase. The benefit is more significant in interrupt handling. While the schemes without vINT service 0% of interrupt sets in all cases, the schemes with vINT service more than 99% of interrupt sets until the range reaches $[0.8, 1.3]$ msec.

When vINT is not used, only the interrupts with slightly longer inter-arrival times can be serviced. Figure 5 depicts the results. In this figure, each point k on the x-axis represents the minimum inter-arrival time of each interrupt in the range of $[k, k + 5]$ msec. As all the schemes schedule 100% of task sets in all cases, we only display the percentage of serviceable interrupt sets in this figure. When the range reaches $[13, 18]$ msec, DSbase services less than 1% of interrupt sets. SSbase performs better than DSbase, but services less than 2% of interrupt sets when the range becomes $[11, 16]$ msec.

VCPU periods: Since the interrupt handling time is largely affected by the VCPU period when vINT is not used, we compare in Figure 6 the percentage of serviceable interrupt sets as the VCPU period increases. All the schemes could schedule 100% of task sets with all VCPU period values depicted in this figure. The schemes with vINT also show 100% of serviceable interrupt sets in all cases. However, without vINT, the percentage drops significantly when the VCPU period is longer than 3.5 msec.

We have also evaluated the impact of the pseudo-VCPU period. Figure 7 shows the percentage of schedulable task sets as the pseudo-VCPU period increases. Each number shown on the x-axis of this figure represents the ratio of the pseudo-VCPU period to the minimum inter-arrival time of interrupts. Hence, a larger value on the x-axis means a longer pseudo-

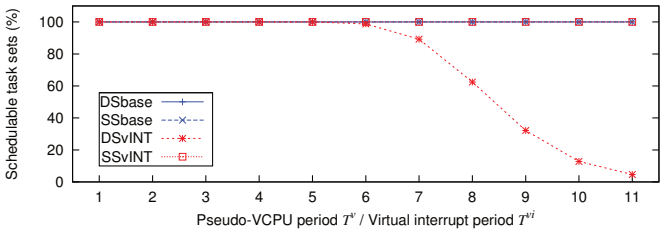


Fig. 7: Results with the change of pseudo-VCPU period

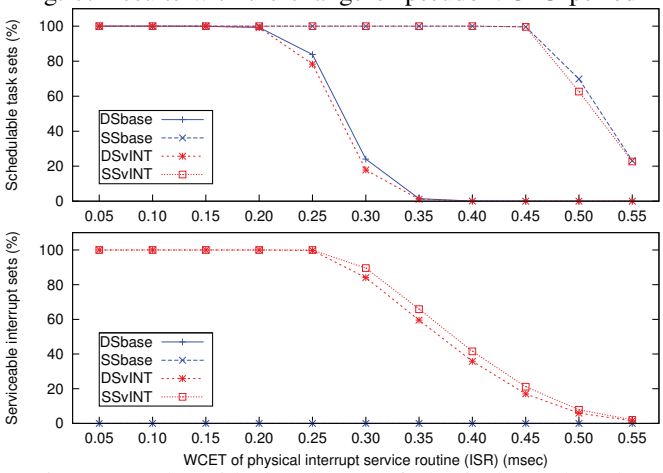


Fig. 8: Results with the change of physical ISR length

VCPU period. As the pseudo-VCPU period increases, task schedulability under DSvINT decreases. This is because DS has a release jitter equal to $T^v - C^v$. Since vINT assigns higher priorities to pseudo-VCPUs, the larger jitter values of pseudo-VCPUs under DS effectively reduce the amount of budget assigned to regular VCPUs. In contrast, SS shows no performance degradation because it has zero release jitter.

WCET of interrupt handlers: We now evaluate the impact of the length of interrupt handlers. Figure 8 and Figure 9 show the results when the WCET of a physical ISR, and the sum of the WCETs of virtual ISR and DSR change, respectively. As the WCET increases, both the percentages of schedulable task sets and serviceable interrupt sets decrease. In case of increasing the WCETs of virtual ISRs and DSRs, the schemes with vINT show lower performance in task schedulability than the schemes without vINT, but provide significantly higher performance in interrupt handling. This is mainly due to the fact that vINT creates pseudo-VCPUs and prioritizes them over regular VCPUs in order to reduce interrupt handling time.

In summary, vINT achieves timely interrupt handling while providing as good task schedulability as when it is not used in most cases. The benefit of vINT multiplies if the inter-arrival time of interrupts is short. Especially, when the minimum inter-arrival time of interrupts is much shorter than the period of VCPUs, the system with vINT outperforms the system without vINT in both task scheduling and interrupt servicing.

C. vINT on KVM Hypervisor

We present a case study demonstrating the effects of vINT by using our implementation on the KVM hypervisor. We chose KVM because it is open-source software and widely used in real-time virtualization studies [10, 18, 24]. Also, it is useful to observe the overall performance impact of vINT, which can be applied to commercial real-time hypervisors.

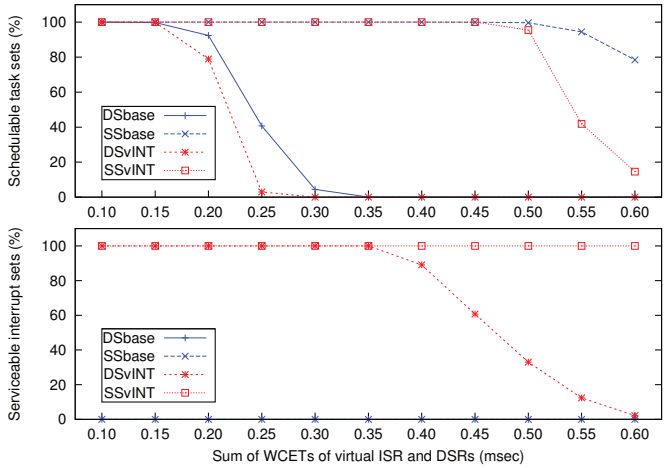


Fig. 9: Results with the change of virtual ISR and DSR length

TABLE III: Implementation cost of vINT on KVM

Primitives	Avg (μsec)	Max (μsec)
Switching btw. pseudo and reg. VCPUs	0.703	1.192
Pseudo-VCPU budget accounting	0.341	1.265
Pseudo-VCPU budget replenishment	0.621	3.045

Implementation: We have implemented a prototype version of vINT on the KVM hypervisor [19] of the latest version of Linux/RK [27, 29].[6] The KVM of Linux/RK allows the host machine to run multiple guest VMs with the deferrable server policy as the VCPU budget replenishment policy. We use an unmodified Linux kernel v3.10.39 as a guest OS.

We have applied vINT to the pass-through PCI device management of KVM. Note that PCI pass-through devices do not involve QEMU in interrupt handling. Hence, once a PCI device is assigned to a guest VM in pass-through mode, all physical interrupts generated by the device are handled by the interrupt handler of KVM, and then resulting virtual interrupts are delivered to the corresponding guest VM, without any intervention from QEMU.

Table III lists the implementation costs of vINT. The target system used is equipped with an Intel Core i7-2600 3.4 GHz quad-core processor and a TP-Link PCI Gigabit NIC using a RTL8169 controller. To reduce measurement inaccuracies, we have disabled the simultaneous multithreading and dynamic clock frequency scaling features of the processor.

Case Study: The target system hosts one guest VM, which has four VCPUs: $\{v_1, v_2, v_3, v_4\}$. Each VCPU is statically assigned to a PCPU with the same index number, i.e. v_i on Core i. The Gigabit NIC of the target system is assigned to the guest VM in pass-through mode. The physical interrupt of the NIC is statically pinned to Core 1 and the corresponding virtual interrupt is pinned to the VCPU v_1. The QEMU process is assigned the highest prioriry to prevent unexpected delays from QEMU device emulation, although it is not involved in the critical path of interrupt handling in pass-through mode. In our case study, we only focus on v_1 on Core 1 and other VCPUs on other cores are kept in idle. When vINT is not used, the VCPU v_1 is assigned 4 msec of budget and 10 msec of replenishment period (40% VCPU utilization). When vINT is used, both v_1 and a pseudo-VCPU created for NIC interrupts are each assigned 2 msec of budget and 10 msec of replenishment period (total 40% VCPU utilization).

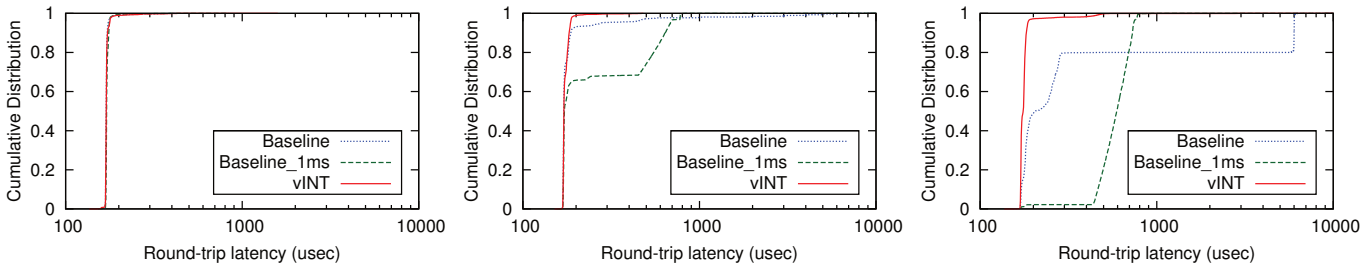
We use three applications in our case study: Netperf [3], MPlayer [2] and busyloop. *Netperf* is a network benchmark consisting of sender and receiver tasks. The Netperf sender task runs natively on a remote system, which has no other workload and is connected to the target system with a direct Ethernet connection. The Netperf receiver task runs in the VCPU v_1. When v_1 receives a virtual interrupt of the NIC, the ISR of the virtual interrupt activates the *softirq* task of the guest Linux kernel, which in turn activates the Netperf receiver task. Both the softirq and Netperf receiver tasks are assigned the highest real-time priority. *MPlayer* is an open-source movie player. MPlayer runs in v_1, with a real-time priority lower than the Netperf receiver task, and decodes a MPEG2 video stream with 1920x1080 (1080p) frame size and 29.97 fps. *Busyloop* is a background task that continuously consumes CPU time, and runs in v_1 with the lowest priority.

We first compare interrupt handling time with and without vINT. For this purpose, we use the UDP round-trip latency test of Netperf, which is highly affected by the system's interrupt handling time. Figure 10 shows the cumulative distribution of the Netperf UDP round-trip latency. "Baseline" and "Baseline_1ms" show the results without vINT, and "vINT" shows the results with vINT. Baseline and vINT use the aforementioned VCPU parameters for v_1. Baseline_1ms uses 0.4 msec of budget and 1 msec of replenishment period for v_1, which results in the same VCPU utilization as Baseline. As shown in the figure, Baseline and Baseline_1ms are significantly affected by the executions of lower-priority tasks within the same VCPU, but vINT is nearly unaffected. Especially, when both MPlayer and busyloop are running, vINT handles 95% of round-trips in less than 200 μsec, while Baseline and Baseline_1ms handle only 50% and 2% of round-trips in 200 μsec, respectively. Interestingly, Baseline_1ms would be expected to outperform Baseline due to its shorter replenishment period, but the results are the opposite due to the higher overhead occurred.

Next, we identify the impact of vINT overhead on the throughput of NIC. Figure 11 shows the results of the TCP throughput test of Netperf with and without vINT as the VCPU utilization increases. The VCPU period is 10 msec in all cases. Only the budget varies from 2 msec to 6 msec. In case of vINT, each point on the x-axis represents the utilization of the pseudo-VCPU. As can be seen, there is no noticeable difference between Baseline and vINT in TCP throughput. This implies that the impact of the overhead induced by vINT is either negligible or acceptably small.

Lastly, we demonstrate the effect of vINT in protecting a real-time task against a virtual interrupt storm. Figure 12 compares the frame rate of MPlayer with and without vINT, in the presence of a virtual interrupt storm which is generated by the TCP throughput test of Netperf. In case of vINT, each point on the x-axis represents the total utilization of original and pseudo-VCPUs. Hence, the budget of the original VCPU varies from 4 msec to 8 msec for Baseline, and from 2 msec to 6 msec for vINT (the pseudo-VCPU budget is unchanged). When vINT is not used, the frame rate of Mplayer is severely

[6]Linux/RK is available at https://rtml.ece.cmu.edu/redmine/projects/rk.



(a) Idle (b) MPlayer only (c) Both MPlayer and busyloop

Fig. 10: Cumulative distribution of Netperf UDP round-trip latency

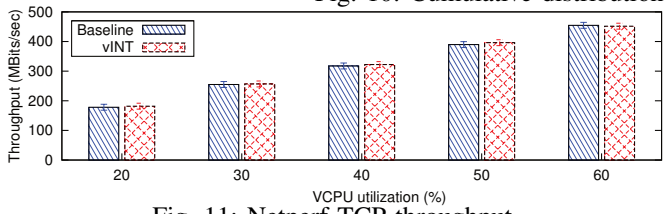


Fig. 11: Netperf TCP throughput

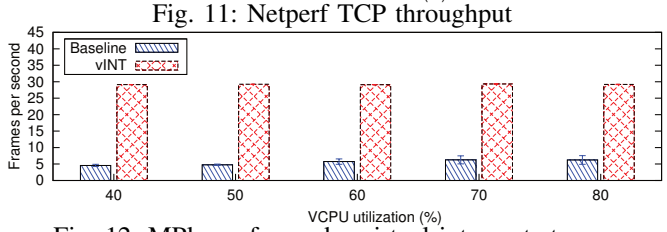


Fig. 12: MPlayer fps under virtual interrupt storms

degraded by a virtual interrupt storm, even when 80% of VCPU utilization is assigned. In contrast, when vINT is used, the frame rate is very close to when there is no interrupt storm. This result shows that vINT can effectively protect the execution of a real-time task against the occurrence of a virtual interrupt storm in a virtualized environment.

VIII. Conclusions

In this paper, we have proposed vINT, an interrupt handling scheme to provide responsive and enforced interrupt handling in a virtualized environment. We have presented our analyses on interrupt handling time, and the schedulability of VCPUs and tasks with and without vINT. Experimental results show that vINT yields significant improvements in interrupt handling performance. For example, a system with vINT services 99% of interrupt sets while a system without vINT cannot service any interrupt set. Our case study on the KVM hypervisor, chosen for convenience, also shows the effects of vINT in reducing interrupt handling time and protecting against interrupt storms. For example, a system with vINT handles 95% of Ethernet round-trips in 200 μsec, and a system without vINT handles only 50% during that time. Under interrupt storms, the frame rate of MPlayer with vINT is nearly unaffected while the frame rate without vINT is dropped to one-fifth of the original one. Future work involves developing algorithms to find efficient VCPU parameters and VCPU-to-PCPU allocation, and allowing migration of tasks, VCPUs and interrupts. Addressing contention on memory resources, such as cache [15] and DRAM [16, 36], in the context of real-time virtualization is also an important topic for future work.

References

[1] L4/Fiasco Microkernel. https://os.inf.tu-dresden.de/fiasco.
[2] MPlayer: an open-source movie player. http://www.mplayerhq.hu.
[3] Netperf network benchmark. http://www.netperf.org.
[4] OKL4 Microvisor. http://www.ok-labs.com.
[5] SYSGO PikeOS Embedded Virtualization. http://sysgo.com.
[6] M. Beckert et al. Sufficient temporal independence and improved interrupt latencies in a real-time hypervisor. In *DAC*, 2014.
[7] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *RTSS*, 1999.
[8] B. B. Brandenburg et al. An overview of interrupt accounting techniques for multiprocessor real-time systems. *J. of Systems Architecture*, 57(6):638–654, 2011.
[9] F. Bruns et al. An evaluation of microkernel-based virtualization for embedded real-time systems. In *ECRTS*, 2010.
[10] T. Cucinotta et al. Respecting temporal constraints in virtualised services. In *COMPSAC*, 2009.
[11] M. Danish, Y. Li, and R. West. Virtual-CPU scheduling in the Quest operating system. In *RTAS*, 2011.
[12] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *RTSS*, 2005.
[13] G. A. Elliott and J. H. Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *ECRTS*, 2012.
[14] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.
[15] H. Kim et al. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS*, 2013.
[16] H. Kim et al. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.
[17] H. Kim, S. Wang, and R. Rajkumar. vMPCP: A synchronization framework for multi-core virtual machines. In *RTSS*, 2014.
[18] J. Kiszka. Towards Linux as a real-time hypervisor. In *RTLWS*, 2009.
[19] A. Kivity et al. KVM: the Linux virtual machine monitor. In *Linux Symposium*, volume 1, pages 225–230, 2007.
[20] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig. Flattening hierarchical scheduling. In *EMSOFT*, 2012.
[21] M. Lewandowski et al. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *RTAS*, 2007.
[22] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz. Integrated task and interrupt management for real-time systems. *ACM TECS*, 11(2):32, 2012.
[23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
[24] R. Ma et al. Performance tuning towards a KVM-based embedded real-time virtualization system. *J. Inf. Sci. Eng.*, 29(5):1021–1035, 2013.
[25] N. Manica et al. Schedulable device drivers: Implementation and experimental results. In *OSPERT*, 2010.
[26] J. C. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.
[27] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *RTSS Work-In-Progress*, 1998.
[28] G. Parmer and R. West. Predictable interrupt management and scheduling in the Composite component-based system. In *RTSS*, 2008.
[29] R. Rajkumar et al. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *MMCN*, 1998.
[30] S. Saewong, M. H. Klein, R. Rajkumar, and J. P. Lehoczky. Analysis of hierarhical fixed-priority scheduling. In *ECRTS*, 2002.
[31] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, 2008.
[32] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM TECS*, 7(3):30, 2008.
[33] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.
[34] U. Steinberg, J. Wolter, and H. Hartig. Fast component interaction for real-time systems. In *ECRTS*, 2005.
[35] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.
[36] N. Suzuki et al. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, 2013.
[37] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: towards real-time hypervisor scheduling in Xen. In *EMSOFT*, 2011.
[38] S. Xi, M. Xu, C. Lu, L. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in Xen. In *EMSOFT*, 2014.
[39] J. Yang et al. Implementation of compositional scheduling framework on virtualization. *ACM SIGBED Review*, 8(1):30–37, 2011.
[40] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *RTSS*, 2006.