RTVirt: Enabling Time-sensitive Computing on Virtualized Systems through Cross-layer CPU Scheduling

Ming Zhao
Arizona State University
mingzhao@asu.edu

Jorge Cabrera
Arizona State University
jecabre2@asu.edu

ABSTRACT

Virtualization enables flexible application delivery and efficient resource consolidation, and is pervasively used to build various virtualized systems including public and private cloud computing systems. Many applications can benefit from computing on virtualized systems, including those that are time sensitive, but it is still challenging for existing virtualized systems to deliver application-desired timeliness. In particular, the lack of awareness between VM host- and guest-level schedulers presents a serious hurdle to achieving strong timeliness guarantees on virtualized systems. This paper presents RTVirt, a new solution to time-sensitive computing on virtualized systems through cross-layer scheduling. It allows the two levels of schedulers on a virtualized system to communicate key scheduling information and coordinate on the scheduling decisions. It enables optimal multiprocessor schedulers to support virtualized time-sensitive applications with strong timeliness guarantees and efficient resource utilization. RTVirt is prototyped on a widely used virtualization framework (Xen) and evaluated with diverse workloads. The results show that it can meet application deadlines (99%) or tail latency requirements (99.9th percentile) nearly perfectly; it can handle large numbers of applications and dynamic changes in their timeliness requirements; and it substantially outperforms the existing solutions in both timeliness and resource utilization.

CCS CONCEPTS

• Computer systems organization → Cloud computing; **Real-time system architecture**; **• Software and its engineering → Virtual machines**;

KEYWORDS

virtualization, time-sensitive computing, cloud computing

ACM Reference Format:
Ming Zhao and Jorge Cabrera. 2018. RTVirt: Enabling Time-sensitive Computing on Virtualized Systems through Cross-layer CPU Scheduling. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3190508.3190527

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '18, April 23–26, 2018, Porto, Portugal
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190527

1 INTRODUCTION

System virtualization [4, 11, 26] allows applications to be conveniently deployed with customized execution environments using virtual machines (VMs) and enables them to flexibly share various types of resources from the physical host. It is a core technology of public and private cloud computing systems [1, 9, 28] which can elastically and cost-effectively provision resources on demand. Many applications can benefit from computing on virtualized systems, including those that have different degrees of timeliness requirements, e.g., a server that needs to stream videos at guaranteed rates or an in-memory datastore that needs to deliver low-latency responses to queries.

However, it is challenging for virtualized systems to deliver the desired timeliness to hosted applications, because of several important reasons. First, the high resource consolidation enabled by virtualization creates complex and dynamic resource contention and performance interference among the applications. Time-sensitive applications are particularly vulnerable to variations of resource availability, and are difficult to achieve the desired timeliness Quality of Service (QoS) on virtualized systems. Second, the resource management on a virtualized system is typically optimized for fair sharing and maximizing overall throughput, but not for timeliness QoS. Consequently, when the resources are shared by multiple VMs hosting applications with different time constraints, the system cannot allocate the necessary resources in time to meet their timeliness requirements. Finally, even if both the VM host and VM guests employ real-time schedulers, the lack of awareness between these two levels of schedulers in the traditional virtualization architecture makes it challenging to provide timeliness guarantees. The host-level VM scheduler is agnostic of the characteristics of guest-level applications and cannot schedule a VM according to its applications' timeliness requirements; and the guest-level application scheduler is unaware of the host-level decisions and cannot schedule its applications properly when it is given time to run.

To address the above challenges, this paper presents *RTVirt*, a new solution to time-sensitive computing on virtualized systems through cross-layer scheduling. First, it provides a new virtualization architecture that enables cross-layer communication and coordination between the host-level VM scheduler and guest-level application schedulers and supports the diverse timeliness requirements of virtualized applications. This cross-layer interface is built upon paravirtualization (specifically a hypercall and shared memory) and supports low-latency and low-overhead interactions between the two levels of schedulers for co-scheduling time-sensitive applications. Second, based on this cross-layer scheduling architecture, RTVirt enables optimal multiprocessor schedulers (e.g., DP-WRAP [16]) which can schedule any task set whose utilization does not exceed processor capacity. As a result, RTVirt can

achieve strong timeliness guarantees with efficient CPU bandwidth utilization.

RTVirt supports time-sensitive applications that have stringent *deadline* requirements (e.g., meeting 99% of the deadlines) but can tolerate some deadline misses. For such applications, deadline misses would not cause catastrophic failures but may lead to undesirable consequences (e.g., reduced service quality to customers and loss of revenues for service providers). RTVirt also supports applications that do not really have deadlines but rather service-level objectives (*SLOs*) specified in latencies (99.9th-percentile latency target). In this paper, we refer to all of such applications as RTAs for conciseness. RTVirt supports both *uniprocessor* and *multiprocessor* VMs hosting time-sensitive applications with *periodic* or *sporadic* requests and with *dynamic* arrivals and dynamically changing parameters. The design of RTVirt is generally applicable to different virtualization frameworks, and it requires *no change to applications*. RTVirt is prototyped on a widely used VM system (Xen [4]).

The paper presents a thorough evaluation of RTVirt. The results show that it meets stringent timeliness requirements (meeting at least *99%* of all the deadlines or a *99.9th percentile* latency target) for virtualized applications in complex and dynamic settings. At the same time, it makes efficient use of the resources and saves up to 50.2% of CPU bandwidth compared to the state-of-the-art works [30]. RTVirt also supports VMs with dynamic bandwidth requirements hosting dynamic RTAs, which cannot be handled by existing solutions [8, 12, 30]. RTVirt allows real-world applications such as video streaming servers to *deliver guaranteed streaming rates* and memcached services to substantially *cut down tail latencies* when they are run on VMs and under intensive resource contention. Finally, the results show that RTVirt has good scalability and low overhead (< 1%) when running 100 virtualized RTAs concurrently on the same host.

Overall, compared to existing solutions, the advantages of RTVirt come from 1) the cross-layer scheduling approach which enables VM guest/host schedulers to collaboratively achieve strong timeliness with good scalability and resource efficiency, and 2) its various designs (including the paravirtualization-based interface, optimal multiprocessor scheduling, and dynamic RTA/VM admission/scheduling) for realizing the full potential of this approach in supporting diverse time-sensitive applications.

The current focus of RTVirt is on cross-layer CPU scheduling, which is a challenging problem to solve on its own. Therefore, in this paper, we assume that the time-sensitive applications are CPU-bound, and each computing task runs from start to completion once it arrives. The applications can still perform other activities such as I/Os; but because RTVirt cannot provide any timeliness guarantee for such activities, the time that an application spends on these activities is assumed to be insignificant compared to its time on CPU. The applications need to declare their timeliness requirements (e.g., required CPU bandwidth and deadline) when they register with RTVirt, but these requirements are allowed to change dynamically.

In the rest of the paper, Section 2 introduces the background and motivations, Section 3 describes the design and implementation, Section 4 presents the evaluation, Section 5 examines the related work, Section 6 discusses several open issues, and Section 7 concludes the paper.

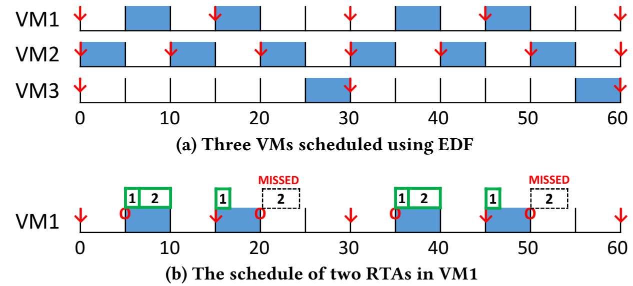


Figure 1: (a) The schedule parameters of three VMs (VM1: (slice=5, period=15), VM2: (slice=5, period=10), and VM3: (slice=5, period=30)). (b) The scheduling of two RTAs (RTA1: (slice=1, period=15) and RTA2: (slice=4, period=15)) inside VM1. The shaded portion shows when VM1 is scheduled by the VMM, and the numbered boxes represent RTA1 and RTA2. RTA1 has the same deadline as VM1 (arrows), and RTA2's deadlines are marked with circles. Every other deadline is missed for RTA2.

2 BACKGROUND AND MOTIVATIONS

System virtualization is implemented by the layer of software called virtual machine monitor (VMM, a.k.a. hypervisor), which is responsible for multiplexing physical resources among the VMs. There are two major approaches to virtualization: *full-virtualization* [11, 26] presents the same hardware interface to guest OSes as the physical machines and supports unmodified OSes; *paravirtualization* [4] presents a slightly modified hardware interface to a guest OS, with features designed to reduce virtualization overheads, but at the expense of requiring the guest OS to be modified to conform to the paravirtualized interface.

While virtualization technologies have supported a wide variety of workloads, their use in time-sensitive environments has not yet flourished. A fundamental challenge in dealing with time-sensitive applications comes from the fact that VMs are designed to be functionally equivalent to physical machines, but are subject to differences in timing and resource constraints. While many applications achieve acceptable service under a best-effort timing regime, applications that are sensitive to delays that cause deadlines to be missed, or jitter and deviations that lead to degraded QoS, may perform poorly in virtualized environments—including real-time simulation, multimedia streaming, and latency-critical services, among others. Although existing real-time scheduling algorithms can be employed by a VM host for VM scheduling and by a VM guest for process scheduling, the lack of awareness between these two levels makes it difficult to provide strong timeliness guarantees. The host-level VM scheduler is agnostic of the characteristics of guest-level applications and cannot schedule a VM according to its applications' timeliness requirements; and the guest-level application scheduler is unaware of the host-level decisions and cannot schedule its RTAs properly when it is given time to run.

We use a simple example to demonstrate this problem. Figure 1a shows three VMs running periodic tasks and sharing a single CPU. It shows the deadlines for each VM and when each VM is scheduled and for how long by an EDF scheduler in the VMM. Based on their

parameters (listed in the caption), the VMs use a total of 100% of CPU bandwidth, so they are supposed to be schedulable (assuming no scheduling overhead); but, in fact, the RTAs running inside the VMs cannot always meet their deadlines. Figure 1b shows an example of two RTAs running in VM1 and scheduled by an EDF scheduler in the guest. VM1 is allocated enough time to schedule both RTAs, i.e., 5/15 = 1/15 + 4/15. When VM1 is scheduled for the first time, both RTAs have arrived and are able to use VM1's CPU time to meet their deadlines. RTA1 is scheduled before RTA2 because the former has an earlier deadline. However, when VM1 is scheduled the second time, only RTA1 is ready to run; and when RTA2 arrives, VM1's time on the CPU has already passed so it misses its deadline. This pattern repeats and RTA2 misses every other deadline.

This example shows how having real-time schedulers at both levels is not enough to guarantee that RTAs running inside VMs will meet their deadlines. First, even though the VMM is aware of a VM's bandwidth requirement, it is still unaware that the RTAs with distinct deadlines are being scheduled inside the VM, and it does not schedule the VM at the times required by the RTAs. Second, the guest-level scheduler is unaware of what points in time the VM will be scheduled, because it does not know the scheduling requirements of the other VMs or the scheduling policy used by the VMM-level scheduler. Consequently, even if the VM is given the necessary CPU bandwidth, and the guest-level EDF scheduler allocates time to each RTA correctly, it cannot avoid missing deadlines.

One possible solution to this problem is to move all the guest-level process scheduling decisions to the host-level scheduler so that the latter manages the scheduling globally across the entire system. This approach is appropriate for embedded systems which host only a small number of VMs, but will be difficult to scale on multiprocessor/multicore systems supporting many VMs. Another approach is to design a static scheduling hierarchy by analyzing the requirements of all the RTAs in the system. It however does not work in dynamic environments (such as cloud systems) where applications/VMs arrive/leave dynamically and their timeliness requirements may also change over time. In comparison, RTVirt leverages cross-layer scheduling to provide strong timeliness guarantees for diverse, dynamic applications with good scalability and efficient resource utilization, as explained in the rest of the paper.

3 CROSS-LAYER REAL-TIME SCHEDULING

3.1 Architecture

The goal of this paper is to design a VM system that is capable of supporting applications with stringent timeliness requirements. The key challenge to realizing this goal on the traditional virtualization architecture is the lack of awareness between a VM host and its VM guests regarding their scheduling information and decisions. In such a virtualized system, the host-level scheduler is agnostic of the guest-level process scheduling, whereas a guest-level scheduler also has no knowledge of the underlying host-level VM scheduling. Although such transparency is key to virtualization, it presents a serious obstacle to delivering strong timeliness guarantees to virtualized time-sensitive applications. Despite the unawareness of each other's scheduling decisions, the VMM and VMs interact in complex ways. Application-desired timeliness requirements cannot be met

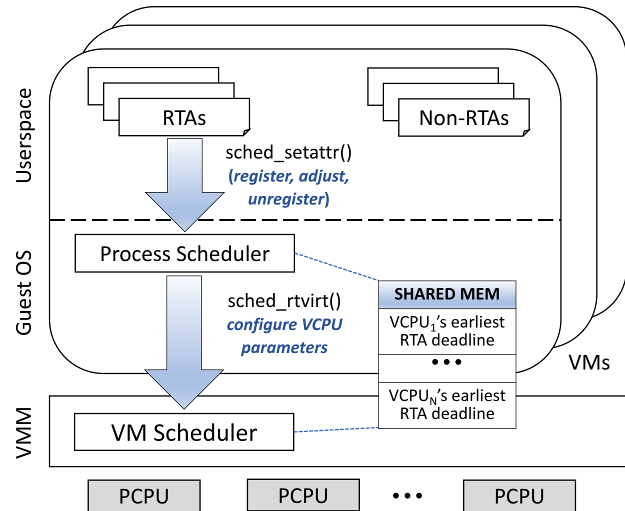


Figure 2: Overview of the architecture of RTVirt. RTAs register, adjust parameters, and unregister with their OS in the VM using an existing system call. The guest OS communicates the VM's scheduling parameters with the VMM on the host using a new hypercall and shared memory. The two levels of schedulers work cooperatively to enforce timeliness guarantees for the virtualized RTAs.

even if both levels employ real-time capable resource schedulers, as demonstrated by the motivating example in the previous section. Therefore, a tradeoff needs to be made on such transparency in order to meet application timeliness requirements on a virtualized system.

RTVirt provides a new virtualization architecture, as illustrated in Figure 2, which keeps the schedulers at the VM-host and -guest levels while enabling cross-layer communication and coordination between the two levels for providing strong timeliness guarantees. On one hand, this architecture preserves the benefits of hierarchical scheduling in virtualized systems, where the guest-level scheduler addresses the timeliness of RTAs (and schedules other background applications (BGAs)) in the VM and the host-level scheduler addresses the timeliness of the VMs in its physical host. Therefore, it can scale to host a large number of VMs and applications. On the other hand, the cross-layer scheduling architecture enables the information sharing and scheduling cooperation between these two levels of schedulers: a guest can inform the host on the timeliness requirement (e.g., bandwidth needs and deadlines) for satisfying its hosted applications, while a host can notify the guest its scheduling decision (e.g., allocated CPU time), both of which are essential for supporting different degrees of timeliness required by the virtualized applications.

RTVirt is designed to support both periodic and sporadic workloads. Without loss of generality, we follow the typical RTA model where once the task is activated, it requires a slice of CPU time s over a period of time p in order to meet a deadline which is at the end of the period. The difference between a periodic task and a sporadic task is that the former is executed at known regular

intervals, and the latter is executed in response to external events with unknown release times. A *periodic task* is said to be arriving at exactly every interval of time p, and thus it is activated p units apart, whereas the release time of a *sporadic task* is not known *a priori* but it arrives at a minimum of p units apart. For both types of workloads, the RTA's time slice and period can be used to summarize its CPU bandwidth requirement, i.e., in order to meet its deadlines, a portion of time equivalent to s/p of a CPU needs to be allocated for that task.

Figure 2 shows the interfaces used at both the guest and host levels. At the user space of a guest, applications use a *system call* to register themselves as RTAs; modify their timeliness requirements; and unregister when they terminate or change to non-time-sensitive. These events are first handled by the scheduler in the guest OS which may later use a *hypercall* and *shared memory* to communicate the change of the VM's timeliness needs with the VMM. The VMM is in charge of performing the schedulability tests and then allocating CPU bandwidths to the VMs based on their requests.

The specific algorithms used by RTVirt in the above operations to perform admission control and scheduling are explained in the rest of this section. This paravirtualization-based cooperative scheduling approach requires changes of existing VM interfaces which currently do not allow a guest or host to influence the other layer's scheduling decisions. Similar to the motivation for existing paravirtualization interfaces [4], it exposes a certain aspect—*scheduling*—of virtualization to guests, trading complete transparency for an important improvement—*improved timeliness* of virtualized applications. Note that while the discussions here use Xen and Linux as examples of hypervisors and guest OSes, the general design of RTVirt is applicable to other systems which can be extended to implement the hypercall and shared memory interface discussed above.

3.2 Guest-level Process Scheduling

At the guest level, we extend existing real-time schedulers [24] in OSes to provide deadline-aware scheduling of the applications and support the cross-layer cooperation with the host-level VM scheduler. For example, on Linux-based guests, RTVirt leverages the SCHED_DEADLINE scheduler class to perform EDF scheduling of processes inside the guest. An application makes explicit scheduling requests to the guest OS through an existing system call (e.g., *sched_setattr()* on Linux). By using the existing system call, any application can run on RTVirt without any change. The implementation of the system call and the guest OS scheduler are modified to support RTVirt's cross-layer scheduling. The guest-level scheduler performs admission control. If there is bandwidth available in the VM, it uses a new hypercall, *sched_rtvirt()*, to request the necessary bandwidth from the VMM, and shares its specific scheduling parameters with the VMM via shared memory.

Specifically, 1) when a new RTA *registers*, the guest scheduler chooses a virtual CPU (VCPU) with enough bandwidth, but before assigning the RTA to the candidate VCPU, it makes the *sched_rtvirt()* hypercall with the INC_BW flag along with the VCPU ID and its required bandwidth. Given these guest-level parameters, the host-level scheduler performs admission control. If the request is granted, the guest-level EDF scheduler then proceeds to assign the RTA to the chosen VCPU, and schedule all the RTAs assigned on this VCPU according to their deadlines; 2) when an existing RTA *requests more bandwidth*, it is handled similarly to the previous case. But if the RTA has to be rescheduled to a different VCPU (due to the lack of bandwidth on the previous one), the guest makes the hypercall with the INC_DEC_BW flag along with the IDs and updated bandwidth requirements of the involved two VCPUs; 3) when an RTA *reduces the bandwidth requirement*, the guest scheduler makes the hypercall with the DEC_BW flag along with the ID and updated bandwidth requirement of the VPCU assigned to this RTA; and 4) when an RTA *unregisters*, RTVirt handles it similarly to 3).

Although the SCHED_DEADLINE scheduler class is based on the global EDF (gEDF) scheduler algorithm, we modify it to perform partitioned EDF (pEDF) scheduling [5]. The difference between pEDF and gEDF is that in pEDF, tasks are pinned to their VCPUs, and in gEDF tasks can migrate between VCPUs. Therefore, when a new RTA is registered, or when an RTA requests a higher CPU bandwidth than what is allocated to it, the guest OS uses pEDF to find a VCPU with available bandwidth to satisfy the request, and the task is pinned to this VCPU if it is found. Note that the guest can reshuffle the placement of RTAs if there is enough bandwidth on the VM to satisfy a request but the available bandwidth is fragmented across the VCPUs. This scenario, however, only happens when RTAs register or increase their current bandwidth requirements.

We use pEDF as the guest-level scheduler because by assigning tasks statically to a specific VCPU, RTVirt can quickly derive the scheduling parameters of the VCPU from the pinned tasks. It does not sacrifice resource usage efficiency, as our VMM-level scheduler (explained in the next section) allows the VCPUs to migrate among physical CPUs (PCPUs) and make full use of the system's available bandwidth. Note that when the number of VCPUs of a VM is not enough for its RTAs, RTVirt uses CPU hotplug to add additional VCPUs to the VM online and still support the RTAs transparently. In comparison, using gEDF would introduce unnecessary complexity in configuring the VCPUs' bandwidths and overhead from migrating process across VCPUs.

3.3 Host-level VM Scheduling

At the host level, we create a new VM scheduler to make optimal use of multiprocessor/multicore resources and support the cross-layer cooperation with the guest-level application schedulers. Specifically, our host-level scheduler is based on the DP-WRAP scheduling algorithm which is an optimal multiprocessor/multicore scheduler that schedules tasks using deadline partitioning (DP) [16]. By doing DP, all tasks are scheduled using the same deadline, instead of using each task's deadline derived from its own period. These new *global deadlines* are the set of deadlines contained in the union of all the tasks' deadlines. The CPU time between consecutive global deadlines (namely, *global slice*) is then partitioned among all tasks based on their required bandwidth (slice over period). DP-WRAP is an optimal multiprocessor scheduler which can use all of the processors' bandwidths for real-time scheduling and guarantee that the tasks meet their deadlines as long as the sum of the tasks' bandwidths is equal or less than the total bandwidth of the processors [16]. It uses task migrations to enable full utilization of the system's bandwidth; but it performs these operations conservatively with an

upper-bound of m-1 (m is the number of PCPUs) migrations during a global slice.

RTVirt follows the DP-WRAP algorithm to schedule VMs at the host level according to the timeliness requirements of the guest-level RTAs. Instead of requiring global knowledge of all RTAs and scheduling them directly, the scheduler makes scheduling decisions at the VCPU granularity and stores only the VCPU parameters in the VMM. The goal of this design is to reduce the complexity of host-level scheduling (including both processing and storage overhead) and improve the scalability of RTVirt.

Specifically, the guest-level scheduler determines the scheduling parameters of a VCPU based on the bandwidth needs of the RTAs running on the VCPU. Each VCPU is configured with a budget and period according to the slice and period parameters of its RTAs: the budget is derived using the sum of the bandwidths of all the RTAs, and the period is decided by the smallest period among the RTAs' periods. In practice, the budget of the VCPU should be set slightly higher (e.g., 500μs more in our evaluation) than what the RTAs need in order to compensate for scheduling overhead of both the guest and VMM levels. The guest-level scheduler shares the scheduling parameters of the VCPU with the host-level DP-WRAP scheduler via the *sched_rtvirt()* hypercall when the assigned RTAs register or change their timeliness needs.

Moreover, in order to meet the deadlines of RTAs running on a VCPU, the host-level DP-WRAP scheduler also requires the next earliest deadline among these RTAs. The guest-level scheduler shares the next earliest deadline on every VCPU of the VM with the VMM using shared memory. The host-level scheduler then decides the next global deadline for the entire system based on the earliest one considering the next earliest deadline of every VCPU from all the VMs in the system. After getting the next global deadline, the host-level scheduler splits the time on each PCPU between the previous and next global deadlines, i.e., the global slice, among all VCPUs currently assigned to this PCPU. Each VCPU is allocated a partition of the global slice on its assigned PCPU proportionally to its bandwidth need. In practice, RTVirt limits the smallest global slice (e.g., 250μs in our evaluation) in order to bound the scheduling overhead, similarly to how an OS scheduler limits the smallest time slice for scheduling processes.

In this way, by considering the RTAs' deadlines, we address the problem illustrated in the motivational scenario (Section 2), which is allocating CPU bandwidth to the VM when the tasks actually need it, and not later after deadline misses occur. At the same time, the information shared between the guests and the host is kept at minimum—only the total bandwidth need and the next earliest deadline for each VCPU, instead of all the scheduling parameters of all the RTAs. This approach allows the system to be scalable when handling large numbers of RTAs with diverse parameters. Moreover, RTVirt leverages the cache coherence of commodity processors to share scheduling information among the PCPUs without explicit synchronization. The guest-level schedulers running on the PCPUs save their VCPUs' next earliest deadlines in memory; the host-level scheduler uses one of the PCPUs to calculate the global deadline from these VCPU deadlines (in $O(logn)$, n = the total number of VCPUs)) and disseminates it to the other PCPUs also via shared memory.

For periodic RTAs, the next earliest deadline of a VCPU can be easily derived among all RTAs assigned on the VCPU, since the time at which each RTA arrives is determined by its period. For sporadic RTAs, however, the next earliest deadline cannot be known beforehand because the RTAs may arrive at anytime. But it is known that once a sporadic task arrives, a minimum period of p needs to elapse before the next time the RTA is activated. Using this knowledge, we configure the next earliest deadline value to handle the worst-case scenario; that is the case in which the sporadic task with the minimum period p is activated immediately p units of time apart. This is the only way to guarantee that the sporadic RTA can meet its deadline when it arrives.

3.4 Summary

Figure 2 shows a summary of all the levels involved in RTVirt-based cross-layer scheduling. RTVirt delivers strong timeliness guarantees transparently to virtualized applications based on the new cross-guest-host scheduling architecture, without modifying the existing user-space interfaces that applications use to request real-time scheduling. It enables the guest-level pEDF scheduler and the host-level DP-WRAP scheduler to cooperate on scheduling decisions using existing paravirtualization mechanisms such as hypercalls and shared memory, and collaboratively enforce the timeliness guarantees for applications. As we demonstrate quantitatively in the next section, by using cross-layer scheduling, RTVirt is able to efficiently utilize practically all of the system's available CPU bandwidth for time-sensitive applications while providing strong timeliness guarantees. After satisfying the requirements for scheduling time-sensitive applications, the remaining bandwidth of the system is allocated among the VMs proportionally, which is used by the guests to satisfy the resource needs of their non-time-sensitive processes; a certain amount of bandwidth can be also reserved for such processes to avoid starvation.

4 EVALUATION

This section presents a comprehensive evaluation of RTVirt on its ability to provide strong timeliness guarantees for diverse applications, make efficient use of CPU bandwidth, and support a large number of dynamic RTAs and VMs.

4.1 Setup

The prototype of RTVirt was implemented on Xen [4], one of the most widely used VM systems. In the experiments, both Dom0 and DomUs ran paravirtualized Linux 4.6.0 kernel. This version does not implement the *steal_clock* function, which is now available in the latest 4.8 release, so we added it ourselves. This function allows a guest to be aware of the "stolen time" which is the time used by the VMM to run other guests when this one was preempted. The hardware testbed consisted of a cluster of servers each with dual eight-core Xeon 2.4GHz processors (with hyperthreading disabled), 64GB RAM, two 1TB HDDs, and a 400GB SSD. One node was used to execute the RTA VMs, and the others were used to run the clients for the sporadic RTA experiments. Dom0 was allocated one full CPU, so the remaining 15 processors were used for scheduling DomUs. The experiments considered both uniprocessor and multiprocessor DomUs.

Category	Group	List of RTAs (Slice, Period)			
Harmonic	H-Equiv	(13,20)	(25,40)	(49,80)	(19,100)
	H-Dec	(7,10)	(13,20)	(18,40)	(13,100)
	H-Inc	(5,10)	(13,20)	(31,40)	(10,100)
Non-harmonic	NH-Equiv	(13,20)	(26,40)	(39,60)	(13,100)
	NH-Dec	(23,30)	(13,20)	(5,10)	(10,100)
	NH-Inc	(11,21)	(26,43)	(40,60)	(13,100)

Table 1: Parameters (in ms) of periodic RTA groups

	RTA		RT-Xen VM		RTVirt VM	
	Slice	Period	Slice	Period	Slice	Period
	23ms	30ms	4ms	5ms	23.5ms	30ms
	13ms	20ms	3ms	4ms	13.5ms	20ms
	5ms	10ms	2ms	3ms	5.5ms	10ms
	10ms	100ms	1ms	9ms	10.5ms	100ms
BW:	2.02 CPUs		2.33 CPUs		2.11 CPUs	

Table 2: Bandwidth requirements for the NH-Dec RTA group, and the corresponding VM configurations

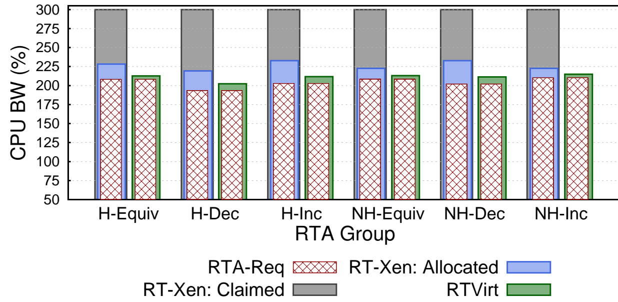


Figure 3: Total CPU bandwidth requirement for each RTA VM group when scheduled under RT-Xen and RTVirt

We used a variety of representative workloads including both synthetic benchmarks and real-world applications to evaluate RTVirt. We compared it with the state-of-the-art related work RT-Xen 2.0 [30], hereinafter referred to as RT-Xen. RTVirt is based on the cross-layer scheduling architecture with pEDF at the guest level and DP-WRAP at the host level, as discussed in the previous section. On account of the scheduling overhead, a $500\mu s$ slack was added to each VCPU's bandwidth allocation, and the smallest global slice was set to $250\mu s$, both empirically determined based on the speed of the hardware. For RT-Xen, we considered its best configuration with pEDF at the guest level and gEDF with deferrable server at the host level, which produces the most efficient VCPU bandwidth allocation and lowest deadline miss ratio compared to its other configurations [30].

4.2 Synthetic Periodic and Sporadic RTAs

The first group of experiments evaluates the deadline guarantees and bandwidth requirements of RTVirt for multiprocessor/multicore environments using synthetic applications.

Periodic RTAs. We ran periodic RTAs using a tool, *rt-app* [3]. It takes the time slice and period as in put, and simulates a periodic load which runs for a specified duration. Table 1 lists the RTAs and their parameters considered in the experiments. The RTAs in each group were run concurrently, one RTA per VM. Each group was run for 100 seconds, and this experiment was repeated for each framework.

Before discussing the results, we use the non-harmonic RTAs (*NH-Dec*) listed in Table 2 to explain how we arrive to a CPU bandwidth allocation for RTVirt and RT-Xen. For RTVirt, the CPU bandwidth requirements of the VMs can be straightforwardly determined based on the RTAs' bandwidth requirements. For VMs equipped with multiple VCPUs and running multiple RTAs, the configuration process is also straightforward as described in Section 3.3.

RT-Xen requires using the CARTS tool [21] to get the period and slice of each VM based on compositional scheduling analysis (CSA) according to the real-time requirements of the RTAs. CARTS also requires the period of the VM as an input which is difficult to determine as the VM's bandwidth requirement changes as the period input varies according to CSA. We try different period values and choose the one that gives the smallest bandwidth requirement for the VM. Then we use the Deterministic Multiprocessor Resource periodic model (DMPR) to get the minimum number of CPUs required to schedule a group of VMs [30]. We have to follow this process to configure all the RT-Xen experiments in this evaluation, which is a nontrivial and time-consuming process.

Both RTVirt and RT-Xen met all the deadlines of all the periodic RTAs, but the amounts of CPU bandwidths that they required are quite different, as shown in Figure 3. The *RT-Xen: Claimed* bar in the figure represents the amount of bandwidth that must be set aside by RT-Xen in order to meet the RTA deadlines according to CSA. It is much higher than the amount of bandwidth actually allocated to the VMs, and the difference is wasted bandwidth which cannot be used to run any new RTAs due to the pessimism of CSA. For example, when running the *H-Equiv* group of RTAs, RT-Xen requires that 2.283 CPUs be allocated and 3 CPUs claimed for the VMs, which leaves 0.717 units of CPU bandwidth unusable for RTAs. If we attempt to run another RTA, there is no guarantee from CSA any more and deadlines are indeed missed in the experiments. This is also the reason why we compare the CPU bandwidth allocation instead of utilization here. Across these experiments, RT-Xen requires on average 0.736 more CPU than what is needed by the RTAs, and the worst case is the *H-Dec* group which has 0.807 CPU wasted. This result is on par with RT-Xen's own finding of 60% wastage [30].

Unlike RT-Xen, RTVirt does not have bandwidth wastage, and any remaining bandwidth that is not allocated to current VMs can be used for new RTAs. As discussed in Section 3, the reasons for this efficiency are two-fold. First, the configuration method used to abstract the bandwidth needs of all RTAs on each VCPU allows RTVirt to be as efficient as possible when allocating bandwidth

Video FPS	CPU Bandwidth Need	RTA Parameters
24	44.5%	s=19ms p=41ms
30	54.1%	s=18ms p=33ms
48	84.5%	s=17ms p=20ms
60	93.6%	s=15ms p=16ms

Table 3: Timeliness characteristics of real-world video streaming applications

to the VMs. Second, the information shared (e.g., the next earliest deadline of each VCPU) through the cross-layer mechanisms allows RTVirt to leverage the optimality of the DP-WRAP scheduler to provide tight timeliness guarantees to every individual RTA running inside each VM.

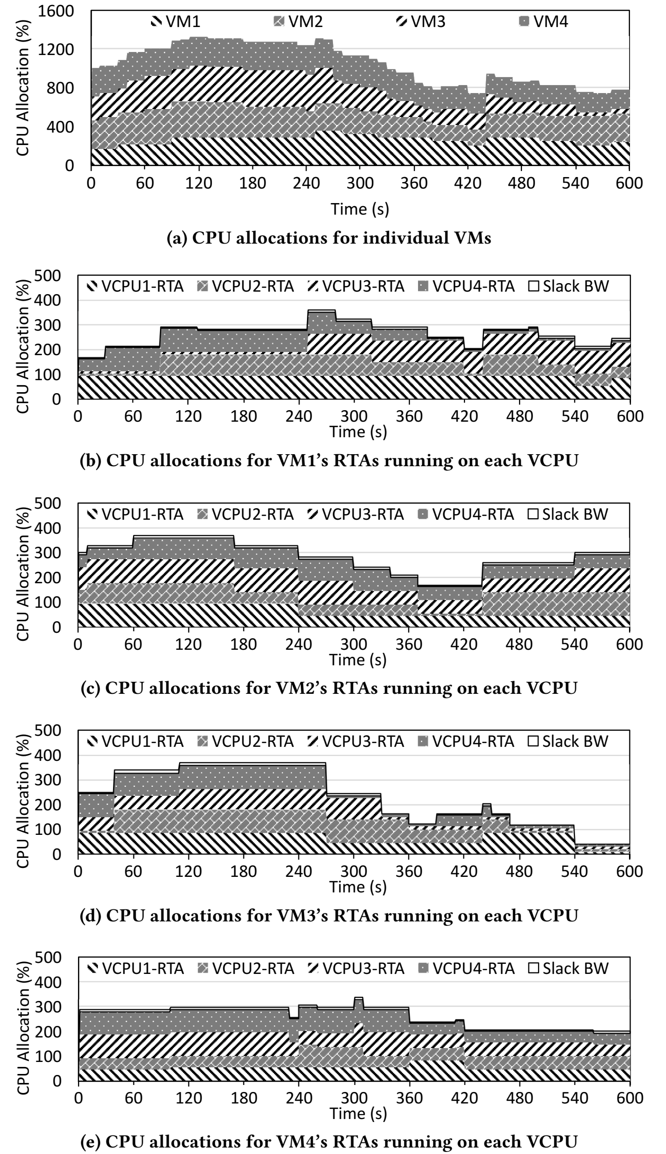
Overall, Figure 3 shows that VMs scheduled in RTVirt require little additional bandwidth than what is required by the RTAs. Compared to RT-Xen, RTVirt allocates on average 6.8% less bandwidth for its VMs, and more importantly, in terms of bandwidth claimed for scheduling RTAs, RTVirt uses on average 39.4% less bandwidth than RT-Xen.

Sporadic RTAs. We configured a set of sporadic RTAs with the same parameters as the above periodic tasks, as shown in Table 1, and the VMs were configured the same as before too. The difference is that instead of being launched at the start of every period, the sporadic workload is activated when it receives an external signal which is implemented by a TCP request from a client across the network. The TCP client runs on a separate host, and it sends requests to the RTA using randomly generated interarrival times with a uniform distribution between 100ms to 1s. Whenever the RTA receives a request, it triggers a one-time CPU-bound job that runs for the duration of the time slice and has a deadline equal to the period.

For each group in the table, we generated 100 sporadic requests to each RTA. The results show that there were no deadline misses for all the sporadic RTAs on both frameworks, but RTVirt requires an average of 39.4% less bandwidth than RT-Xen's VMs, as in the periodic RTA experiments. Note that the potential network delays to the requests are not considered here. In our experiments, we observed that the 99.9th percentile delay introduced by the network transfer is 19μs, which is indeed insignificant compared to the RTAs' deadlines. Controlling the network delay is out of the scope of this paper; please see Section 4.4 for more detailed discussion.

4.3 Video Streaming Server

RTVirt's cross-layer scheduling approach and ease of configuration allow it to support RTAs and VMs with dynamic arrivals/departures and dynamically changing parameters, which is difficult to do with existing solutions. For example, RT-Xen needs offline configuration of VM interfaces, as discussed in the previous section, which requires knowledge of the parameters of all RTAs in the system, and a lengthy process to find out the optimal VM period. It also lacks the necessary cross-layer features that RTVirt has for RTAs to inform the VMM on changes to their timeliness requirements



(a) CPU allocations for individual VMs

(b) CPU allocations for VM1's RTAs running on each VCPU

(c) CPU allocations for VM2's RTAs running on each VCPU

(d) CPU allocations for VM3's RTAs running on each VCPU

(e) CPU allocations for VM4's RTAs running on each VCPU

Figure 4: CPU allocations for video streaming VMs

and for VMM to perform admission control online and adapt the bandwidth allocation dynamically according to the changes.

In this section, we evaluate RTVirt's support for dynamic RTAs. Examples of applications that can benefit from this support include media streaming applications whose CPU bandwidth requirement changes over time depending on the required streaming quality. We model such applications by configuring *rt-app* with parameters obtained from a real-world streaming application VLC [25]. VLC launches a transcoding thread to service each new streaming request, and depending on the frame rate used to encode the video, the threads' timeliness requirements vary. We use *rt-app* with different parameters to represent VLC's transcoding threads. Table 3 shows the CPU bandwidth required for streaming a video using

VLC with four different frame rates and the RTA configurations that model the streaming application. The periods are obtained from the frame rates used by the application (we round the periods to the floor of the decimal), and the time slices are derived from the observed CPU usage of the application.

In the experiment, we ran four VMs, each with four VCPUs to run RTAs which were dynamically spawned to handle video streaming requests with various frame rate requirements. The experiment lasted for 10 minutes during which various numbers of RTAs started and stopped running in each of the VMs. Each RTA was created with random parameters chosen in the following way:

- *Timeliness requirement* of each RTA is randomly chosen from one of the four configurations listed in Table 3.
- *Start time and duration* of each RTA are randomly assigned (with uniform distributions) between the start and end of the experiment and between 10 seconds and 6 minutes, respectively.
- *VCPUs* are each randomly assigned one of the RTAs configured as discussed above or a random interval of idle time (uniformly distributed between 10 seconds and 6 minutes) during which it is reserved 10% of bandwidth.

RTVirt supports these dynamics by allowing the RTAs to dynamically register and unregister with the guest schedulers, using the system call, as they enter and leave the system, respectively. It also allows the VMs to dynamically change their CPU bandwidth requests, using the hypercall, based on the needs of their hosted RTAs. These changes can all be done quickly in the system as each event requires only the invocation and handling of these system call and hypercall (10*us* on average. See Section 4.5 for the overhead analysis).

Figure 4a shows the CPU allocations to each VM throughout the experiment. The rest of Figure 4 provides a detailed view of CPU allocations to each VCPU of the VMs as RTAs with different parameters were dynamically assigned to this VCPU over time. For example, Figure 4b shows how three RTAs ran over the period of 10 minutes on VCPU1 of VM1. The first RTA is the one with (s=15, p=16), which ran from the beginning of the test to the 540s mark, followed by an RTA with (s=18, p=33) which ran until the 580s mark, and finally an RTA with (s=18, p=33) which ran until the end of the test.

RTVirt provided strong timeliness guarantees to these dynamic RTAs—out of the 54 RTAs that were run throughout the test, only five had deadline misses, and in the worst case the deadline miss percentage was 0.136%. In the meantime, RTVirt saved substantial amount of CPU bandwidth compared to the static approach which allocates a fixed amount of bandwidth to each VM based on its peak load.

4.4 Memcached

In this section, we present a series of experiments that show RTVirt's capabilities of supporting VMs hosting sporadic RTAs. We use memcached to model real-world sporadic workloads with stringent timeliness requirements [18]. Memcached is an in-memory caching framework, which enables dynamic web applications to access remote data quickly by allowing data that would normally be stored

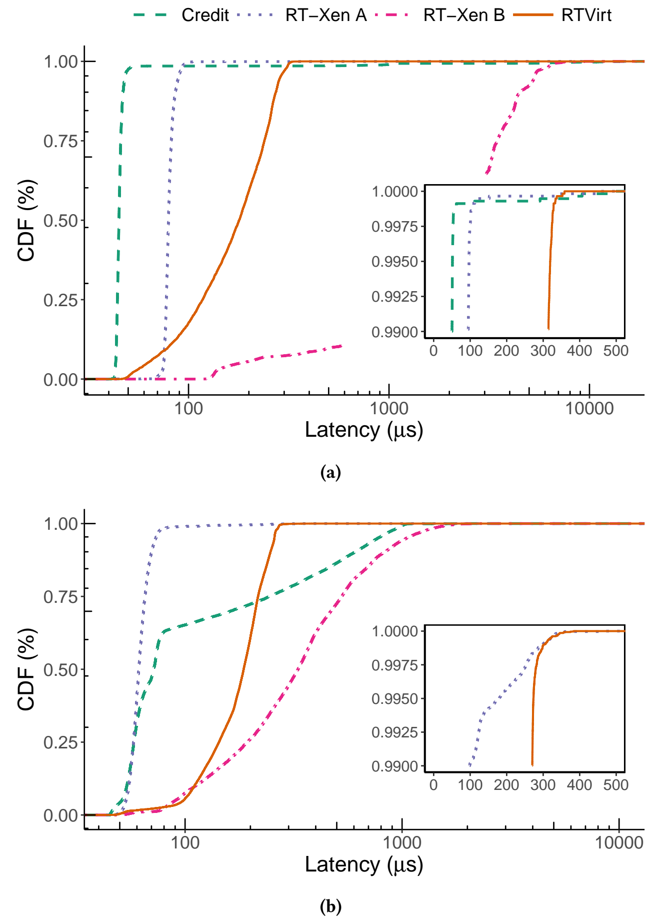


Figure 5: Latency distributions of memcached VMs contending with (a) non-RTA VMs and (b) periodic VMs emulating video streaming servers

in external storage to be cached in memory. Memcached is a good example of workloads that have sporadic activities—requests may arrive at any time—and require timeliness guarantees—response times should meet the desired service-level objectives (SLOs).

We used Mutilate [20] to generate a workload that represents the query distributions at Facebook, following the methodology of the related work on tail latencies [15]. The workload includes all GET requests to 200B values uniformly at random from a set of 30B keys, and the inter-arrival times follow a normal distribution with an average rate of 100 queries per second. We ran Mutilate on a separate physical server to issue requests to the memcached VM across the network. Although the per-VM request rate is not high, the latency target for the requests is set according to the request rate. The experiment also confirms that it is not trivial to meet this target when the memcached VM is under contention from other RTA and non-RTA VMs. Each memcached VM was configured with one VCPU, and multiple concurrent memcached VMs were used to simulate sharded memcached servers.

We measured the *tail latency* of the requests, and focused on the *NIC-to-NIC latency*, the same way as another related work on tail latencies [17]. Specifically, we recorded the latency from

Scheduler	90th	95th	99th	99.9th
Credit	113.3μs	114.4μs	120.6μs	129.1μs
RT-Xen	49.6μs	50.7μs	54.6μs	65.7μs
RTVirt	51.3μs	52.2μs	54.5μs	57.5μs

Table 4: Tail latency of memcached requests for the memcached VM on a dedicated CPU using different schedulers

the time when the request arrives at Dom0 to the time when the response is sent out to the network. We do not collect the latencies reported by the client because these measurements include delay introduced by the network which is outside the control of RTVirt. The current focus of RTVirt is on addressing the challenges of meeting the timeliness requirements of virtualized CPU-bound workloads within the constraints of the VM host. By providing a solution with strong guarantees in this area, we believe that RTVirt can then be a valuable component in a more holistic solution that addresses other aspects of time-sensitive computing on a virtualized system such as network contention. We considered different levels of NIC-to-NIC tail latencies from 99th percentile to 99.9th percentile, and set the SLO to 500μs which also serves as the period for the memcached RTA. In comparison, the 99.9th percentile network latency (between the client and server) was only 19μs, which is insignificant compared to the NIC-to-NIC latency.

In order to determine the CPU bandwidth need of the memcached VM for RTVirt and RT-Xen (as well as Xen's default Credit scheduler [6] as an additional baseline), we first ran the VM on a dedicated CPU, and measured the request processing latencies for each of the frameworks (listed in Table 4). These results already show that among the three schedulers, RTVirt is able to handle most of the memcached requests using the least amount of time. That is, in order to complete 99.9% of the memcached requests in time the memcached VM needs a time slice of 58μs on RTVirt, 66μs on RT-Xen, and 129μs on Credit.

Based on these results, we can derive the configuration for the memcached VM on each framework. For Credit, we configured the VM with a weight that is equivalent to 130μs/500μs = 26% of the CPU bandwidth. The scheduler's global timeslice parameter is set to 1ms, and its ratelimit parameter to 500μs. For RTVirt, the VM was configured with (p=500μs, s=58μs). For RT-Xen, (p=500μs, s=66μs) was used as the input to the CSA tool. The most CPU-efficient configuration given by the tools is (p=14μs, s=2μs), but the period is too small and results in the VM not runnable when configured as such. Therefore, we have to use larger periods that allow the VM to run, and the two most efficient configurations are (p=283μs, s=66μs), hereinafter referred to as *RT-Xen A*, and (p=177μs, s=33μs), hereinafter referred to as *RT-Xen B*.

Non-RTA Workload Contention. In the first experiment, we ran the memcached VM using the previously mentioned configurations, alongside 19 VMs containing non-RTA CPU-bound processes. These 20 VMs shared two PCPUs. The bandwidth that was not reserved by the memcached VM was distributed equally among the 19 background VMs.

The distribution of the memcached request latencies is shown in Figure 5a. The results reveal that only *RTVirt* and *RT-Xen A* are able to meet the SLO, with a 99.9th percentile latency of 379μs and 114μs, respectively; but *RTVirt* uses 50.2% less CPU bandwidth than *RT-Xen A*. Note that RT-Xen achieves a lower tail latency here only because of its overprovisioning of resources; in comparison, RTVirt can meet the same SLO with a much lower resource usage. *Credit* and *RT-Xen B* cannot meet the SLO, and have a 99.9th percentile latency of 7.1ms and 8.4ms, respectively. Note that Credit does have a low average latency, because it prioritizes the memcached VM when it comes back from idle to service a new request; but Credit is still undesirable due to its long tail latency.

Periodic Workload Contention. In the second experiment we launched five memcached VMs alongside ten periodic VMs running emulated video streaming servers. The memcached VMs were configured as before, but each with an independent workload generated by a separate Mutilate instance. Among the video streaming VMs, three were configured to stream at a 24fps rate, three at 30fps, two at 48fps, and two at 60fps using the parameters listed in Table 3. In total, the combined bandwidth allocated to the memcached and video streaming VMs is 7.44 CPUs for *RTVirt*, 8.16 CPUs for *Credit*, 8.03 CPUs for *RT-Xen A*, and 8.27 CPUs for *RT-Xen B*. Note that, as discussed in Section 4.2, the CSA tool requires both RT-Xen groups to have a *claimed* bandwidth of 15 CPUs. It means that almost seven more CPUs need to be set aside for the RT-Xen groups in order to meet their deadlines, and this wasted bandwidth cannot be used to run any additional RTA.

Figure 5b shows the latency distribution aggregated for all five memcached VMs. The 99.9th percentile latency is 303μs for *RTVirt*, 1170μs for *Credit*, 1974μs for *RT-Xen A*, and 296μs for *RT-Xen B*. Only RTVirt and *RT-Xen B* are able to meet the SLO. Interestingly, the only RT-Xen configuration that meets the SLO in the previous experiment is *RT-Xen A*. For the video streaming VMs, RTVirt has only one VM with 0.8% of deadline misses. Credit has five VMs with many deadline misses and the worst one has 14.35% of the deadlines missed. RT-Xen has no deadline misses because of its significant overprovisioning of resources.

Overall, RTVirt is the best in meeting the SLO of sporadic workloads with dynamic request arrivals and at the same time also meeting the deadlines of periodic workloads. This strong timeliness guarantee is achieved while requiring substantially less CPU bandwidth (10% less in terms of allocated bandwidth to the VMs and 46.7% less in terms of claimed bandwidth).

4.5 Scalability and Overhead

In this section, we show that RTVirt is scalable even when hosting a large number of RTAs on VMs. In order to do so, we first present an overview of the complexity of RTVirt's architecture, followed by an experimental evaluation.

Complexity Analysis. At the guest level, RTVirt employs the simple pEDF scheduler which has a complexity of $O(logl)$, where l is the number of tasks in a VCPU. From guest to host, hypercalls are invoked when a RTA registers, adjusts its bandwidth requirements, or unregisters, and each hypercall has an overhead of 10μs on average. The space used by shared memory to communicate the deadlines between guest and host is 8 bytes for each VCPU, which is negligible. At the host level, the VMM's *schedule()* and *context switch* functions are where most of the overhead is introduced.

The *schedule()* function is where the scheduling algorithm is implemented. Specifically, there are two types of work here. First, determining the next global deadline: At the end of each global slice—the interval between two global deadlines, *schedule()* is called to determine the new global deadline as discussed in Section 3.3. This task requires sorting the next earliest deadline provided by each VCPU, which takes $O(logn)$ work, where n is the number of VCPUs, and it is done by only one PCPU (e.g., CPU 0) with the result shared with all the other PCPUs. Each PCPU is then in charge of determining the slice of each of its VCPUs, which in total requires $O(n)$ time considering all PCPUs. Therefore, the *schedule()* call does $O(n) + O(logn)$ work.

Second, choosing the next VCPU: In between calls where global deadlines are determined, the *schedule()* function performs a much lighter routine which chooses a VCPU to schedule next and then performs a context switch. A linked-list based runqueue is maintained for each PCPU, and the VCPUs in the runqueue do not need to be sorted—they can follow any specific order. This work requires $O(1)$ work, since we simply choose the VCPU at the head of the runqueue. When a VCPU is done using its allocated time on a PCPU, if it still has time remaining on another PCPU, it is migrated there; otherwise, it is moved to the waiting queue. The number of migrations performed between global deadlines for all PCPUs is $O(m - 1)$, where m is the number of PCPUs. The total number of context switches done between global deadlines is at most $O(n - 1)$, considering all the VCPUs in the system.

Overhead Measurement. In order to quantify the aforementioned overhead, we ran 10 groups of periodic RTAs with randomly generated parameters (listed in Table 5). We then measured the duration of the *schedule()* and context switch functions for two scenarios where a total of 100 RTAs were executed concurrently on the same host.

- *Multi-RTA VMs*: We ran 10 RTAs per VM and a total of 10 VMs on the system. Each VM hosted a different group of RTAs, and was configured with the minimum number of VCPUs required by its hosted RTAs. This configuration resulted in 1 VM with 4 VCPUs, 1 with 3 VCPUs, 5 with 2 VCPUs, and 3 with 1 VCPU. In total, this experiment involved 100 concurrent RTAs on 10 VMs with a total of 20 VCPUs.
- *Single-RTA VMs*: We ran 100 RTAs, 10 from each group, each on a separate single-VCPU VM. In total, this experiment involved 100 concurrent RTAs on 100 VMs with a total of 100 VCPUS.

Comparing these two scenarios, in *Multi-RTA VMs*, the guest-level scheduler has more work to do for scheduling the 10 RTAs in a VM, whereas in *Single-RTA VMs*, the VMM-level scheduler has to schedule a lot more VCPUs.

We also evaluated RT-Xen in these two scenarios. Due to RT-Xen's pessimism, it cannot support so many RTAs as RTVirt does. For the *Multi-RTA VMs* scenario, we were able to run 10 RTAs each for only the first 8 groups before CSA requires us to use more than 15 PCPUs which are all we have on the host (one PCPU was dedicated to Dom0). As a result, we had 80 RTAs across 8 VMs where 1 VM with 4 VCPUs, 1 with 3 VCPUs, 5 with 2 VCPUs, and 3 VMs with 1 VCPU. For *Single-RTA VMs*, we were able to run 93 RTAs, including 10 RTAs for Groups 1 to 3 each and 9 RTAs for Groups 4 to 10 each.

Group #	(Slice, Period)	Group #	(Slice, Period)
1	(6ms, 75ms)	6	(13ms, 124ms)
2	(7ms, 92ms)	7	(36ms, 260ms)
3	(46ms, 188ms)	8	(21ms, 159ms)
4	(12ms, 102ms)	9	(9ms, 103ms)
5	(19ms, 139ms)	10	(62ms, 208ms)

Table 5: Groups of RTAs used in scalability experiments

(a) *Multi-RTA VMs* Scenario

Framework	Time Spent on Schedule	Time Spent on Context Switches	Total Overhead (%)
RT-Xen	331,020μs	63,829μs	0.39
RTVirt	32,026μs	71,059μs	0.10

(b) *Single-RTA VMs* Scenario

Framework	Time Spent in Schedule	Time Spent in Context Switches	Total Overhead (%)
RT-Xen	850,043μs	1,307,759μs	2.16
RTVirt	238,108μs	689,119μs	0.93

Table 6: Time spent on the *schedule()* function and context switches and the total overhead in terms of percentage of the total runtime

Results from these experiments show that, first, RTVirt is able to handle a large number of RTAs in both intensive scenarios, with no deadline misses for *Multi-RTA VMs* and only 0.007% deadline misses for *Single-RTA VMs*. In terms of overhead, RTVirt spends only 0.1% and 0.93% of the total runtime on the *schedule()* function and context switches, for the *Multi-RTA VMs* and *Single-RTA VMs* scenarios, respectively (shown in Table 6). Moreover, the scheduling overhead of RTVirt is also much lower than RT-Xen. First, the time spent on the *schedule()* calls, in terms of both per *schedule()* call time and total time, is much lower than RT-Xen. Second, RT-Xen also suffers from a lot higher context switch overhead, which we believe is due to the frequent use of VM migrations performed by RT-Xen's gEDF scheduler at the host level. In comparison, RTVirt's DP-WRAP scheduler considers the cost of VM migrations and uses them judiciously.

Overall, RTVirt is the only framework capable of meeting the timeliness requirements of all RTAs while incurring a negligible overhead and fully using the system's CPU bandwidth to schedule time-sensitive workloads. Note that at the time of this paper's submission, a new experimental version of RT-Xen was released, which changed the implementation from quantum-driven to event-driven to reduce the number of *schedule()* calls. But we have verified that the per *schedule()* call overhead is still higher than RTVirt and it does not address its higher context switch overhead.

5 RELATED WORK

As shown in the previous evaluation and the motivation example in Section 2, although related real-time schedulers can be employed by either guest OS, VMM [13, 14], or both [2, 30], the lack of cooperation between the host- and guest-level schedulers makes it difficult to enforce an application's timeliness requirement without sacrificing CPU usage efficiency. RTVirt addresses these limitations with a new cross-layer scheduling architecture for virtualized time-sensitive applications.

There are several related works on improving the timeliness of virtualized systems. RT-Xen [30] has been thoroughly compared to in the previous section, and in summary, 1) it requires offline configuration of VM interfaces which is time consuming and makes it unable to support RTAs with dynamic timeliness requirements; and 2) its pessimism results in severe underutilization of CPU bandwidth. In some cases, e.g., for the periodic workloads in Section 4.4, overprovisioning helps RT-Xen to not miss any deadline; but in other cases, e.g., for the sporadic workloads in Section 4.4, even overprovisioning is not sufficient. A conventional real-time theory such as CSA is not appropriate for modern general-purpose systems (e.g., cloud servers), as its pessimism cannot guarantee that deadlines/SLOs are always met, but it does severely underutilize the resources all the time. In comparison, RTVirt delivers strong timeliness guarantees (at most 0.8% of deadline misses) with highly efficient resource utilization. These properties are particularly important to applications that can tolerate some deadline misses but desire efficient resource utilization (e.g., cloud applications).

Related works have studied flattening the hierarchical scheduling in virtualized systems by allowing guests to export scheduling information to the host [8, 12]. Although they also follow the general cross-layer scheduling approach, there are several limitations in their specific designs: 1) they support only simple host-level schedulers (e.g., EDF) which are not optimal for multiprocessor systems; 2) they cannot handle tasks and VMs with dynamic arrivals or dynamic scheduling parameters; and 3) they lack considerations for scalability and were evaluated with only a small number of (three) VMs.

In comparison to the above related works, RTVirt provides flexible (online configuration, dynamic tasks/VMs) and scalable (100 RTAs per host) support to diverse applications (periodic and sporadic tasks) with strong timeliness guarantees (in deadlines and tail latency targets).

For embedded systems, several hypervisors have been proposed to allow general-purpose OSes to run along with real-time OSes [10, 19, 22], which typically provide real-time guarantees by putting the RTAs on dedicated cores. This approach is appropriate for embedded systems as they need to support only a small set of RTAs (e.g., automative control) that these systems are specifically designed for. In contrast, RTVirt targets larger and more diverse environments such as clouds which host a large number of applications with different timeliness characteristics and requirements on highly consolidated servers; RTVirt's cross-layer scheduling approach allows these applications to time-share resources while still achieving their desired timeliness.

Cross-layer VM scheduling can also be achieved by using middleware, without changing the existing VM interfaces. For example, related work proposed such a solution [27], where 1) the host-level VM manager exploits guest-level application knowledge to better estimate the VMs' resource demands; and 2) the guest-level application manager uses the host-level feedback to adapt the applications according to changing resource availability. In comparison, RTVirt uses paravirtualization to enable fine-grained, low-overhead cross-layer scheduling which is necessary for providing strong timeliness guarantees to time-sensitive applications.

Considering the general problem of addressing latencies in a shared computing environment, there are several related works developed for non-virtualized systems. Leverich *et al.* proposed a solution [15] that supports QoS guarantees in shared cluster environments by using overprovisioning to handle interference, reducing thread migration, and replacing the default Linux CFS scheduler with a real-time scheduler. In comparison, RTVirt provides a new cross-layer scheduling framework for virtualized applications, and achieves strong timeliness guarantees without overprovisioning.

Dean *et al.* discussed fine-grained techniques that reduce latency variability at the component level, as well as coarse-grained techniques to mask unpredictable high-latency episodes across various components [7]. Li *et al.* explored possible sources of delay found in the hardware, kernel, and application layer which can affect the tail latency of applications [17]. RTVirt is complementary to these works as their techniques can be employed to reduce latencies in a distributed, virtualized system. At the same time, RTVirt can be a key component to provide timeliness guarantees at the VM/VMM level and offer strong support to the guarantees required by the cluster/datacenter-level schedulers.

6 DISCUSSIONS

Security is important in scenarios where the applications may not be trustworthy, and request CPU resources more than what they actually need. While security is out of the scope of this paper, there are several factors that mitigate such risks. First, the schedulers can monitor the applications'/VMs' actual CPU usages, and tax the applications/VMs if they claim more than what the need. The tax rate of an application/VM can be determined based on the observed idle CPU ratio and will be used by the schedulers to proportionally deduct the application's/VM's CPU allocation when the system's CPU bandwidth is oversubscribed. This technique is similar to the idle memory tax used by VMM for reclaiming idle memory from VMs [26]. In practice, RTVirt also limits the smallest interval between global deadlines (250μs in the prototype implementation), which ensures that inappropriate deadlines given by VMs do not affect the overall efficiency of the system. Second, in typical public cloud environments [1, 9, 28], users are charged based on the amount of resources that they claim, not how much they actually use, and are9 therefore not encouraged to claim more than what they actually need.

Overhead in RTVirt for allowing VMs to migrate across PCPUs and making efficient use of the system's CPU bandwidth is low, as the host-level DP-WRAP scheduler limits the number of migrations to at most m-1 (m = the total number of PCPUs) during a global slice, and it is also shown to be insignificant in our evaluation results. In larger systems where this overhead becomes more considerable, RTVirt can further reduce this limit as needed. Similarly, RTVirt can

also support CPU affinity for VMs that are sensitive to processor cache locality by simply excluding such VMs from the m-1 VMs that the host-level scheduler considers to migrate.

Support for other OSes and VMMs is conceivable following RTVirt's cross-layer scheduling approach, as it requires only a hypercall and a shared memory area for communicating the guest-level scheduling parameters to the host-level scheduler. The changes required to the guest/host interface is small. While Xen is the most well-know paravirtualized system, other VMMs have also adopted paravirtualization for various purposes.

Other scheduling considerations can be addressed by extending the solid cross-layer scheduling framework provided by RTVirt. First, the cross-layer approach can be extended to the scheduling of other important resources such as storage and network I/Os for supporting I/O-intensive time-sensitive applications. Second, considering the availability of multiple hosts, RTVirt's VM admission and scheduling process can be extended to optimize the placement of VMs across different hosts, in addition to the placement of VCPUs across different PCPUs on a single host. Live VM migration can be considered to dynamically adjust VM placement at runtime, but its overhead must be properly accounted for [29]. Third, while RTVirt is designed to deliver strong timeliness guarantees, deadline misses are sometimes unavoidable (less than 0.8% in our evaluation results) due to the nondeterminism of the system. In general, deadline misses can be further reduced by increasing the scheduling slack (500μs per VCPU in the prototype). Considering the different priorities of RTAs, the slack can also be assigned in proportion to the priorities so that more important RTAs will have less likelihood to miss their deadlines than the less important ones.

7 CONCLUSIONS AND FUTURE WORK

In this paper we have studied cross-layer techniques to enable time-sensitive CPU scheduling for virtualized applications. The result is RTVirt, a framework that is capable of meeting stringent timeliness requirements of virtualized applications while efficiently utilizing the system's CPU resources and incurring low overhead. Our cross-layer scheduling approach allows us to overcome some of the major hurdles to enabling time-sensitive applications on virtualized systems. Specifically, our approach entails synergizing the scheduling capabilities held by the VM host and guest levels and creating a hierarchical scheduling framework that has enough information to deliver CPU resources to applications at the right time and for the right duration.

Our experimental evaluation confirms that RTVirt can support applications with diverse timeliness requirements, including applications that have periodic or sporadic tasks and applications that have deadlines or just need low latencies. For these applications, RTVirt is able to meet at least 99% of all the deadlines or deliver the required latency at the 99.9th percentile target. Compared to the existing solutions, RTVirt is substantially more efficient in CPU usage, and it supports dynamic RTAs and dynamic VMs with changing timeliness requirements. Finally, our experiments show that RTVirt is also scalable, supporting a large number of concurrent RTAs and VMs, while keeping both the deadline miss percentage and overhead to under 1%.

RTVirt relies on paravirtualization to build the cross-layer scheduling, which trades the transparency between VM and host for the ability to provide strong timeliness guarantees with high system utilization. We believe that this tradeoff is necessary for many time-sensitive applications. Moreover, for users, the use of paravirtualization and cross-layer scheduling in RTVirt is completely transparent to their applications.

RTVirt is an open-source project [23] and a solid framework for a variety of further studies on virtualized time-sensitive computing. One of our future objectives is to expand the support of cross-layer scheduling to include I/O resources, in order to support applications that are dependent on timely delivery of I/O resources, in addition to CPU bandwidth. Another topic of interest is studying the combination of host-level resource scheduling with cross-host network scheduling, in order to deliver end-to-end timeliness guarantees to distributed applications.

ACKNOWLEDGEMENT

The authors thank Steven Hand for shepherding the paper and the anonymous reviewers for their helpful comments. This research is sponsored by U.S. Department of Defense award W911NF-13-1-0157, National Science Foundation CAREER award CNS-1619653 and award CNS-1562837, and a gift from Huawei Technologies.

REFERENCES

[1] Amazon EC2. http://aws.amazon.com/ec2.
[2] Åsberg, M., Nolte, T., and Kato, S. Towards partitioned hierarchical real-time scheduling on multi-core processors. *ACM SIGBED Review 11*, 2 (2014), 13–18.
[3] Bagnoli, G. rt-app. https://github.com/gbagnoli/rt-app.
[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, Oct. 19–22 2003), vol. 37, 5 of *Operating Systems Review*, ACM Press, pp. 164–177.
[5] Bastoni, A., Brandenburg, B. B., and Anderson, J. H. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proceedings of the IEEE 31st Real-Time Systems Symposium (RTSS)* (2010), IEEE, pp. 14–24.
[6] Credit scheduler. http://wiki.xen.org/wiki/Credit_Scheduler.
[7] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.
[8] Drescher, M., Legout, V., Barbalace, A., and Ravindran, B. A flattened hierarchical scheduler for real-time virtualization. In *Proceedings of the 13th International Conference on Embedded Software* (2016), ACM, p. 12.
[9] Google compute engine. https://cloud.google.com/compute/.
[10] Globallogic's Nautilus platform. https://www.globallogic.com/wp-content/uploads/2016/12/GlobalLogic-Nautilus-Platform.pdf.
[11] Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.
[12] Lackorzyński, A., Warg, A., Völp, M., and Härtig, H. Flattening hierarchical scheduling. In *Proceedings of the tenth ACM international conference on Embedded software* (2012), ACM, pp. 93–102.
[13] Lee, M., Krishnakumar, A. S., Krishnan, P., Singh, N., and Yajnik, S. Supporting soft real-time tasks in the Xen hypervisor. *SIGPLAN Not. 45*, 7 (Mar. 2010), 97–108.
[14] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., and Hyden, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications 14*, 7 (1996), 1280–1297.
[15] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 4:1–4:14.
[16] Levin, G., Funk, S., Sadowski, C., Pye, I., and Brandt, S. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)* (2010), IEEE, pp. 3–13.
[17] Li, J., Sharma, N. K., Ports, D. R., and Gribble, S. D. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.
[18] Memcached. https://memcached.org/.
[19] Mentor embedded hypervisor. https://www.mentor.com/embedded-software/hypervisor/.

[20] Mutilate. https://github.com/leverich/mutilate.
[21] Phan, L. T., Lee, J., Easwaran, A., Ramaswamy, V., Chen, S., Lee, I., and Sokolsky, O. CARTS: A tool for compositional analysis of real-time systems. *ACM SIGBED Review 8*, 1 (2011), 62–63.
[22] RTS real-time hypervisor. https://www.real-time-systems.com/products/index.php.
[23] RTVirt. https://github.com/jorge-cabrera/rtvirt.
[24] Sched_deadline. http://www.evidence.eu.com/sched_deadline.html.
[25] VideoLAN Streaming Solution. http://www.videolan.org/vlc/streaming.html.
[26] Waldspurger, C. A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.
[27] Wang, L., Xu, J., and Zhao, M. Application-aware cross-layer virtual machine resource management. In *Proceedings of the 9th ACM International Conference on Autonomic Computing* (San Jose, CA, USA, 2012), ICAC'12, ACM.
[28] Windows Azure Platform. http://www.microsoft.com.
[29] Wu, Y., and Zhao, M. Performance modeling of virtual machine live migration. In *Proceedings of IEEE international conference on Cloud computing (CLOUD)* (2011), IEEE, pp. 492–499.
[30] Xi, S., Xu, M., Lu, C., Phan, L., Gill, C., Sokolsky, O., and Lee, I. Real-time multi-core virtual machine scheduling in Xen. In *Proceedings of the International Conference on Embedded Software (EMSOFT)* (Oct 2014), pp. 1–10.