# Towards Virtualization-Agnostic Latency for Time-Sensitive Applications

Haoran Li
Washington University in St. Louis
Saint Louis, Missouri, USA
lihaoran@wustl.edu

Meng Xu
University of Pennsylvania
Philadelphia, Pennsylvania, USA
mengxu@cis.upenn.edu

Chong Li
Washington University in St. Louis
Saint Louis, Missouri, USA
chong.li@wustl.edu

Chenyang Lu
Washington University in St. Louis
Saint Louis, Missouri, USA
lu@wustl.edu

Christopher Gill
Washington University in St. Louis
Saint Louis, Missouri, USA
cdgill@wustl.edu

Linh Phan
University of Pennsylvania
Philadelphia, Pennsylvania, USA
linhphan@cis.upenn.edu

Insup Lee
University of Pennsylvania
Philadelphia, Pennsylvania, USA
lee@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania
Philadelphia, Pennsylvania, USA
sokolsky@cis.upenn.edu

## ABSTRACT

As time-sensitive applications are deployed spanning multiple edge clouds, delivering consistent and scalable latency performance across different virtualized hosts becomes increasingly challenging. In contrast to traditional real-time systems requiring deadline guarantees for all jobs, the latency service-level objectives of cloud applications are usually defined in terms of *tail latency*, i.e., the latency of a certain percentage of the jobs should be below a given threshold. This means that neither dedicating entire physical CPU cores, nor combining virtualization with deadline-based techniques such as compositional real-time scheduling, can meet the needs of these applications in a resource-efficient manner.

To address this limitation, and to simplify the management of edge clouds for latency-sensitive applications, we introduce *virtualization-agnostic latency (VAL)* as an essential property to maintain consistent tail latency assurances across different virtualized hosts. VAL requires that an application experience similar latency distributions on a shared host as on a dedicated one. Towards achieving VAL in edge clouds, this paper presents a *virtualization-agnostic scheduling (VAS)* framework for time-sensitive applications sharing CPUs with other applications. We show both theoretically and experimentally that VAS can effectively deliver VAL on shared hosts. For periodic and sporadic tasks, we establish theoretical guarantees that VAS can achieve the same task schedule on a shared CPU as on a full CPU dedicated to time-sensitive services. Moreover, this can be achieved by allocating the minimal CPU bandwidth to time-sensitive services, thereby avoiding wasting CPU resources. VAS has been implemented on Xen 4.10.0. In case studies running time-sensitive workloads on Redis and Spark streaming services, we show that in practice the task schedule on a shared CPU can closely approximate the one on a full CPU.


*RTNS'2021, April 7–9, 2021, NANTES, France*

ACM ISBN 978-1-4503-9001-9/21/04…$15.00
https://doi.org/10.1145/3453417.3453420

## CCS CONCEPTS

• **Computer systems organization → Real-time system architecture**; **Real-time operating systems**.

## KEYWORDS

Real-Time Scheduling, Virtualization, Deferrable Server

**ACM Reference Format:**
Haoran Li, Meng Xu, Chong Li, Chenyang Lu, Christopher Gill, Linh Phan, Insup Lee, and Oleg Sokolsky. 2021. Towards Virtualization-Agnostic Latency for Time-Sensitive Applications. In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3453417.3453420

## 1 INTRODUCTION

With the emergence of edge computing as a new distributed computing paradigm, time-sensitive applications are deployed in *edge clouds* [33] located close to the sources of data [19]. Workloads within edge clouds typically comprise a mix of time-sensitive services and non-real-time applications [15]. Unlike traditional real-time applications with deadline constraints, the service level objectives (SLOs) [43] of time-sensitive cloud applications are usually defined in terms of tail latency, i.e., the latency of a certain percentage of the jobs should be below a given threshold [11, 12, 17, 30]. For example, a common SLO of Amazon DynamoDB was that 99.9% requests should be completed within 300ms [12]; a Google BigTable benchmark was targeted at $99.9^{th}$ percentile latency [11].

In contrast to centralized cloud infrastructure, edge cloud operators face the additional challenge of managing numerous edge

clouds distributed in different locations. This challenge is compounded by the need to meet the SLOs of time-sensitive applications deployed on different edge clouds. Traditionally, significant effort is required to test, tune, and configure an edge service for each edge cloud. This is a labor-intensive process that cannot scale effectively for a large number of edge sites. It is thus necessary to provide a simple and effective system management solution for time-sensitive workloads on edge clouds[1].

Current cloud infrastructure typically meets applications' SLOs by dedicating physical CPUs (PCPUs) to time-sensitive services, i.e., allowing time-sensitive services to monopolize *full CPUs*. For example, Heracles [29] isolates cores to achieve a specific latency SLO. VMware also supports time-sensitive applications by dedicating PCPUs [39], as does RedHat in the real-time KVM project [16]. For time-sensitive cloud services like Redis, it has been suggested [28] to set the CPU affinity and use a dedicated core to maximize performance. While effective in meeting applications' SLOs, dedicating PCPUs to VCPUs can incur resource overprovisioning, which is undesirable for resource-constrained edge platforms.

In contrast to the full CPU approach, the real-time systems community has developed compositional scheduling frameworks and real-time virtualization technologies that allow VMs to share a PCPU, thereby achieving real-time performance on *partial CPUs* [1, 2, 20, 22, 26, 40, 42]. However, existing compositional scheduling approaches are geared toward meeting deadlines for hard real-time systems. Treating tail latency thresholds as hard deadlines would lead to resource overprovisioning. Furthermore, allocating CPU resources based on existing compositional scheduling analyses [3, 7, 9, 14, 27, 34–36] may lead to low resource utilization due to pessimism in providing real-time deadline guarantees in a hierarchical manner.

To simplify management and resource sharing in edge clouds, we introduce *virtualization-agnostic latency (VAL)* as a desirable property to maintain consistent latency across different edge clouds. VAL requires that an application experiences similar latency distributions on a shared host as on a dedicated one. With VAL the same time-sensitive application can be deployed on multiple edge clouds with similar latency distributions. Thus, VAL greatly reduces the effort needed for performance tuning to deliver desired tail latency at any percentile of the latency distribution[2].

To achieve VAL in a resource-efficient manner, we propose *virtualization-agnostic scheduling (VAS)* to deliver VAL for a time-sensitive application on a shared CPU, while allowing general-purpose applications to reclaim unused CPU cycles in a virtualized host.[3]

Specifically, the contributions of this work are four-fold:

- We propose *virtualization-agnostic latency (VAL)* as a desirable property to maintain similar latency distributions for a time-sensitive application on different edge clouds.
- We introduce *virtualization-agnostic scheduling (VAS)*, a practical CPU scheduling framework for time-sensitive applications on shared virtualized hosts. Tailored for resource-constrained edge devices, VAS employs a simple but effective scheduling approach to achieve VAL on shared CPUs.
- For periodic and sporadic tasks, we establish *theoretical guarantees* that time-sensitive tasks within a partial CPU can achieve the same task schedules as those on a full CPU, and thereby achieve the same latency distribution. Moreover, we also prove that the minimal CPU bandwidth can be achieved while maintaining VAL.
- We have implemented VAS on Xen 4.10.0. Experimental results show that even on a real platform with scheduling and context switching overheads, the task schedule in a partial CPU can closely approximate that in a full CPU in case studies running time-sensitive applications on Redis and Spark Streaming services.

## 2 BACKGROUND

We now give an overview of the Xen hypervisor [6] as a representative real-time scheduling approach for virtualized hosts. While VAS is implemented and evaluated in the Xen hypervisor, the scheduling and resource provisioning approach can also be generalized to other virtualization platforms.

A Xen-based virtualized system includes an administration VM (Domain 0) and several guest domains. Domain 0 is used by system operators to manage the hypervisor as well as other VMs. Each guest VM has its operating system, and each VM has tasks scheduled by the guest OS on the Virtual CPUs (VCPUs) of the VM. The Xen scheduler schedules all VCPUs of all domains on physical cores.

Xen introduced a real-time scheduler, called the *Real-Time Deferrable Server (RTDS)* scheduler [40], in Xen 4.5.0. RTDS provides guaranteed CPU capacity to guest VMs on symmetric multiprocessing (SMP) machines. Each VCPU is implemented as a *deferrable server* in the RTDS scheduler. A VCPU is represented as $V_i = (B_i, P_i)$, where $P_i$ is the *period* and $B_i$ is the *budget*, indicating that the VCPU is guaranteed to run for $B_i$ time units in every interval $P_i$. The deferrable server mechanism defines how a VCPU's budget is managed: a VCPU's budget decreases when the VCPU is running on a core; a VCPU is suspended till the end of the current period when its budget has been exhausted; a VCPU's budget is replenished to $B_i$ at the start of every period $P_i$, and a VCPU's remaining budget is discarded at the end of the current period.

The deferrable server model has two key properties for real-time systems: (1) it guarantees that a VCPU always gets its configured CPU resource (i.e., $B_i$ time per $P_i$ interval) when the system is schedulable; and (2) it prevents a VCPU from running for more time than the budget in each period, thereby providing isolation between VCPUs sharing PCPUs. The deferrable server approach may cause additional delays for tasks running on it [25, 26]: if the budget is exhausted before the end of the period, no further service is provided until the next period. Thus, the pending task will wait for budget replenishment at the beginning of the next period,

[1] An edge cloud may employ different infrastructure technologies including virtualization, containers, or serverless computing. This work focuses on virtualized hosts, although the scheduling approaches may be extended to container-based platforms.

[2] We note that maintaining consistent latency distributions is a sufficient but not necessary condition to achieve specific tail latency. A contribution of this work is to demonstrate that VAL provides a practical and convenient abstraction to deliver desired tail latency on virtualized hosts.

[3] As a first step towards virtualization-agnostic latency, we focus on CPU scheduling in this paper. To develop a fully virtualization-agnostic system solution, future work is needed to manage performance interference caused by other shared resources such as cache, memory, and I/O subsystems.

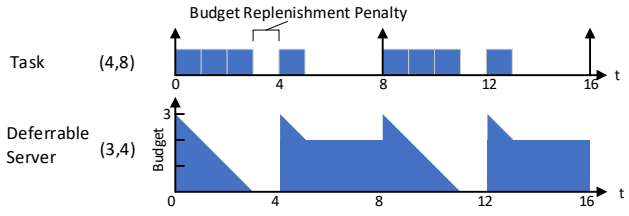



**Figure 1: Budget Replenishment Penalty**

referred to as a *budget replenishment penalty*. For example, Fig. 1 shows a periodic task, whose period is 8 and worst-case execution time is 4, running in a deferrable server with a period of 4 and a budget of 3: At time 3, because the deferrable server has exhausted its budget, the pending job is delayed until time 4; the response time is 5, which suffers from a budget replenishment penalty of 1 time unit.

While RTDS is an effective real-time scheduler for meeting deadlines in VMs sharing CPUs, it is not designed to achieve expected tail latency objectives required by time-sensitive applications. Furthermore, it cannot provide consistent latency distributions in different edge clouds.

# 3 SCHEDULING APPROACH

## 3.1 Task Model

An edge server is a multi-tenant system comprising time-sensitive tasks and general-purpose tasks. A *time-sensitive* task has an SLO in terms of expected tail latency. We assume that a time-sensitive task maintains the same execution time when deployed in different edge clouds. In practice, an edge cloud operator can maintain consistent execution times in different edge clouds by adopting machines from the same vendor according to uniform specifications. While the edge servers share the same hardware profile, they run different mixes of time-sensitive and general-purpose tasks, which may make it challenging to meet the tail latency requirements in different edge clouds. For example, while a manufacturer may deploy edge clouds using the same hardware platform in different factories, the same time-sensitive task may be co-located with a different set of general-purpose applications locally.

A *general-purpose* task has no latency requirements, but may have throughput requirements and demand a certain share of CPU in a multi-tenant system. To improve resource efficiency, we allow a VCPU hosting time-sensitive tasks to share a PCPU with VCPUs hosting general-purpose tasks.

VAS aims to provide (1) *virtualization-agnostic latency (VAL)* for time-sensitive tasks and (2) *resource isolation* for time-sensitive and general-purpose tasks. VAL requires that an application experiences similar latency distributions on a shared host as on a dedicated one. With VAL the same time-sensitive application can be deployed on different edge clouds with similar latency distribution, thereby greatly reducing the effort needed for performance tuning on individual edge cloud to deliver desired tail latency. In addition, as edge clouds are multi-tenant systems, we need to provide resource isolation between VMs running time-sensitive or general-purpose tasks, so that an aggressive or faulty VM will not cause starvation in other VMs on the same server.

## 3.2 Scheduling Framework

VAS provides a practical scheduling framework to provide VAL and resource isolation for VMs sharing virtualized hosts. A VM has one or more VCPUs. A VCPU may be *time-sensitive* if it runs time-sensitive tasks, or *general-purpose* if it runs general-purpose tasks. A virtualized system employs two-level scheduling. The hypervisor schedules VCPUs on PCPUs, and the VM scheduler schedules tasks on VCPUs.

**VM Scheduler.** Each time-sensitive VM runs a partitioned scheduler in which each VCPU is scheduled by the same **stable** and **work-conserving** scheduling policy on that VCPU. A stable scheduling policy forms the same schedule for the same input. A work-conserving policy never lets the VCPU idle when there are pending tasks on the VCPU. Note that we do not require the partitioned scheduler to be work-conserving on the multicore VM. Instead, because each VCPU is scheduled independently in partitioned scheduling, we only require the scheduling policy on the VCPU to be work-conserving locally. For example, common scheduling policies such as EDF, RM, and FIFO are work-conserving.

**Hypervisor Scheduler.** The hypervisor employs partitioned fixed-priority scheduling, i.e., a set of VCPUs is pinned to and scheduled on a PCPU based on a preemptive fixed-priority policy. A VCPU is scheduled as a deferrable server with a *resource interface* $(B, P, R)$, where the *budget* $B > 0$, the *period* $P \geq B$, and $R$ is the *priority*. Accordingly, the VCPU will be scheduled to run for $B$ time units every $P$ time units, at priority $R$. The deferrable server provides a mechanism to meet the latency requirement of time-sensitive VCPUs and provide resource isolation for both time-sensitive and general-purpose VCPUs.

Only one time-sensitive VCPU can be allocated on a PCPU and runs at the highest priority on the PCPU, which can be shared with multiple general-purpose VCPUs. Therefore, a host with $m$ cores may host up to $m$ time-sensitive VCPUs. Though restricting the number of time-sensitive VCPUs on a host, this solution is suitable for edge servers on which workloads are dominated by general-purpose services[4]. While restrictive, this scheduling approach provides a practical and efficient way to achieve VAL for time-sensitive tasks. A contribution of this work is to establish both theoretically and experimentally that VAS can achieve its design goals (as is detailed in Sections 4 and 6).

# 4 THEORETICAL PROPERTIES

Our first design goal is to achieve virtualization-agnostic latency for all the tasks within a time-sensitive VM. In this section we will prove that, theoretically, VAS leads to the same task schedule on a partial CPU as on a full CPU. While it is not the only way to achieve virtualization-agnostic latency, this theoretical property further enhances the predictability of time-sensitive services deployed on partial CPUs.

[4]A similar but simpler system model was adopted for predicting latency distributions of aperiodic services on virtualized platforms, which assumed a single time-sensitive service on each PCPU [24]. VAS allows multiple time-sensitive services to share the time-sensitive VCPU on each PCPU.

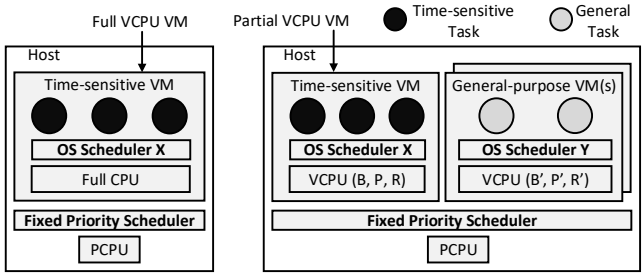



**Figure 2: System Model**

The intuition originates from a well-known property of fixed-priority scheduling: the highest-priority task's schedule is not affected by lower-priority tasks (assuming there is no other resource sharing dependency). We formalize this property in the case of VAS in Theorem 4.1. However, in a hierarchical scheduling scenario, simply assigning the highest priority to the time-sensitive VM (VCPU) is not sufficient to maintain the same task schedule within the VM, because the budget enforcement mechanism may alter the schedule of the task when a server is suspended. The key theoretical contribution, therefore, is to establish the conditions under which the budget enforcement will not be activated for a deferrable server (if the time-sensitive application conforms to its workload specification). Further, we derive the minimal VCPU bandwidth and server configuration that avoids budget enforcement (Theorem 4.2): hence, we can achieve virtualization-agnostic latency with an optimal configuration in terms of resource consumption.

## 4.1 System Model

We now formalize the system model as a basis for our theoretical analysis. Recall that VAS employs a partitioned scheduler in the hypervisor on which VCPUs are partitioned among the underlying PCPUs. Because each PCPU is scheduled independently, henceforth, we focus on a single PCPU that is running a single time-sensitive VCPU and a set of general-purpose VCPUs.

A free offset periodic (or sporadic) *task system* $\Gamma$ is defined as $\Gamma = \{\tau_i = (A_i, C_i, T_i) | i = 1..N\}$, where the *worst case execution time (WCET)* $C_i > 0$, the *period* of a task (or *minimal inter-arrival time* for a sporadic task) $T_i > C_i, T_i \in \mathbb{N}^+$, , and the *offset* satisfies $0 \le A_i < T_i$. The *utilization*, $U$, of task system $\Gamma$, is $U = \sum_i \frac{C_i}{T_i}$. The *hyperperiod* of the task system is $LCM\{T_i\}$, the least-common-multiple of the period of all the tasks. A VCPU is a deferrable server whose *resource interface* is $(B, P, R)$. The *bandwidth* $W$ of a VCPU is $W = \frac{B}{P}$. On a PCPU the task system of interest resides in a *time-sensitive VCPU*, while other VCPUs on the same PCPU are denoted as *general-purpose VCPUs* (Fig. 2). The time-sensitive task system is scheduled hierarchically by a scheduler $X$ on the VCPU; and the VCPU (deferrable server) is scheduled by a preemptive *Fixed Priority (FP)* scheduler in the hypervisor, on the PCPU. A *full CPU VCPU* is a time-sensitive VCPU whose bandwidth $W = 1$. Hence, it monopolizes the single PCPU of the host (Full VCPU VM in Fig. 2). A *partial CPU VCPU* is a time-sensitive VCPU whose bandwidth $W < 1$. Thus, it can share the PCPU with VCPUs from other general-purpose VMs (Partial VCPU VM in Fig. 2).

For simplicity of presentation, in the rest of the paper, we will refer to full CPU VCPU as "full VCPU", and to partial CPU VCPU as "partial VCPU". Let the task system of interest be $\Gamma$, whose $U < 1$. $\Gamma$ is scheduled in either a full VCPU or a partial VCPU. We denote $G(\Gamma, t)$ as the scheduled job of a task system $\Gamma$ at time $t$.

For the same task system $\Gamma$ and same stable and work-conserving scheduler $X$, we want to answer two questions:

(1) Is it possible to make the schedule, $G'(\Gamma, t)$, in a partial VCPU, the same as the schedule, $G(\Gamma, t)$, in a full VCPU for all time $t$, by setting the resource interface of the partial VCPU?
(2) If so, what is the optimal resource interface setting, in terms of achieving the minimal bandwidth $W$?

## 4.2 Theorems and Proofs

Theorem 4.1. *The schedule of $\Gamma$ in a partial VCPU, $G'(\Gamma, t)$, can be identical to the schedule $G(\Gamma, t)$ in a full VCPU, as long as (1) the VCPU of the partial VCPU has the highest priority among other VCPUs and (2) the budget of the VCPU is never exhausted within any period of the VCPU.*

From Theorem 4.1, if enough budget is provided for each period of a partial VCPU with highest priority, we can ensure that $\Gamma$ achieves the same schedule as it would on a full VCPU. Then, Theorem 4.2 indicates an optimal resource interface can be achieved:

Theorem 4.2. *To make the same schedule $G(\Gamma, t)$ in a full CPU, the minimal bandwidth of a partial VCPU should not be less than the utilization of $\Gamma$, i.e., $W \ge U$. Specifically, if the period of the partial VCPU is the hyperperiod of $\Gamma$, i.e., $P = LCM\{T_i\}$, then $W = U$. Thus, the resource interface of the partial CPU is $(P \times U, P, Highest)$;*

**Proof of Theorem 4.1.** We prove Theorem 4.1 by contradiction: We assume that we can find the earliest time $t_0$ where the partial CPU's schedule differs from that of a full one, even though the partial CPU has highest priority and the budget is never exhausted within a period. That is, $\exists t_0 \ge 0$, where,

$$\begin{cases} G(\Gamma, t) = G'(\Gamma, t) & t \le t_0 \\ G(\Gamma, t_{0+}) \ne G'(\Gamma, t_{0+}). \end{cases} \quad (1)$$

For time $t_0$, there are four possible cases (Fig. 3):

(1) $G(\Gamma, t_{0+})$ has no job scheduled; $G'(\Gamma, t_{0+})$ has a job.
(2) $G(\Gamma, t_{0+})$ and $G'(\Gamma, t_{0+})$ have different jobs scheduled.
(3) $G(\Gamma, t_{0+})$ has a job scheduled; $G'(\Gamma, t_{0+})$ has no job; and $t_{0+}$ falls in a deferrable server's period.
(4) $G(\Gamma, t_{0+})$ has a job scheduled; $G'(\Gamma, t_{0+})$ has no job; and $t_{0+}$ is at the end of one period of a deferrable server.

Case (1) implies that a new job is released at $t_0$. However, the full VCPU does not schedule this pending job, which contradicts the "work-conserving" assumption. For case (2), the schedulers in the full and partial VCPU should have the same scheduling state (and pending queue), but they make different decisions, which contradicts the "stable" assumption. Because scheduler $X$ is a work-conserving scheduler, it can refuse to schedule a job for $G'(\Gamma, t_{0+})$, only if the VCPU budget is exhausted. Thus, Case (3) contradicts the "budget never exhausted within a period" assumption. For case (4), even if the budget is exhausted at the end of the previous period ($t_{0-}$), the budget will replenish at the start of a new VCPU period,

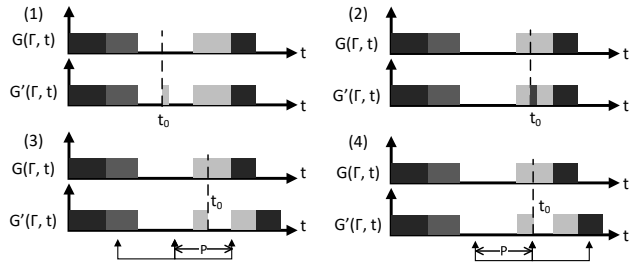



**Figure 3: Proof of Theorem 4.1**

thus also creating a contradiction. We find a contradiction for each case, thus proving Theorem 4.1. □

Before proving Theorem 4.2, we introduce some auxiliary functions and lemmas. Let the *budget demand function* , $D(t)$, of task system $\Gamma$, be

$$D(t) = \sum_{i=1}^{N} \left\lceil \frac{t - A_i}{T_i} \right\rceil C_i \qquad t \geq 0. \tag{2}$$

The budget demand function $D(t)$ denotes the total execution time that all jobs released before $t$ request from the task system $\Gamma$ (where a job of $\tau_i$ would request $C_i$ execution time units as soon as it is released). $D(t)$ is a stepwise function.

We define $\mathbb{Y} \subset \mathbb{R}$ to indicate where $D(t)$ is continuous:

$$\mathbb{Y} = \mathbb{R} \setminus \{t = kT_i + A_i | \forall k \in \mathbb{N}, i = 1, 2..N\}. \tag{3}$$

Let the continuous *budget supply function*, $S(t)$, be

$$\begin{cases} S(t) = 0 & t \leq 0 \\ \frac{d}{dt} S(t) = 0 & S(t) = D(t), \quad t \in \mathbb{Y} \\ \frac{d}{dt} S(t) = 1 & S(t) < D(t), \quad t \in \mathbb{Y} \\ \lim\limits_{x \to t} S(x) = S(t) & \forall t \in \mathbb{R}. \end{cases} \tag{4}$$

The budget supply function $S(t)$ denotes **the total execution time provided by a full VCPU before time** $t$. The third equation in (4) indicates that when there are pending (unfinished) jobs, the VCPU is active and providing service to the jobs in the queue. The second equation in (4) indicates that when there are no pending jobs and hence the queue is empty, the VCPU is idle. Hence, $S(t)$ reflects the budget supply behavior of a *work-conserving* scheduler. The fourth equation denotes the continuity of $S(t)$ [5].

Fig. 4a shows the demand and supply values vs. time, i.e., $D(t)$ and $S(t)$, of a typical periodic task system $\Gamma_1 = \{\tau_i = (A_i, C_i, T_i) | i = 0, 1, 2, 3\}$, where the parameters (in *ms*) of four tasks are (150, 40, 250), (100, 200, 500), (50, 100, 1000), and (0, 200, 2000), respectively.

Lemma 4.3. *For the same task system with any work-conserving scheduler X, $S(t)$ is unique.*

**Proof.** $D(t)$ is determined by task system $\Gamma$. $D(t)$ is a non-decreasing stepwise function, so $S(t)$ can be determined by $D(t)$ uniquely. According to the definition of $S(t)$ in Eq. 4, $S(t)$ reflects a generic work-conserving scheduler's behavior, so any work-conserving scheduler should have the same $S(t)$. □

Lemma 4.4. *If $P = LCM\{T_i | i = 1, 2..N\}$, then $S(t)$ satisfies:*

$$S(t + P) - S(t) = PU, \qquad t \geq \max_i \{A_i\}, \tag{5}$$

$$S(t + P) - S(t) \leq PU, \qquad t < \max_i \{A_i\}. \tag{6}$$

**Proof.** Let scheduler X be an EDF scheduler, considering task system $\Gamma$ with *implicit deadlines*. Then, using Theorem 1 in [23], $\exists t_0 = \max\{A_i\} \geq 0$, where the schedule repeats itself after $t_0$, with a hyperperiod $P = LCM\{T_i\}$.

In each hyperperiod, the schedule is the same. So $S(t_0 + (n+1)P) = S(t_0 + nP) + M$, where $M$ is a constant value. For time $[0, t_0)$, the accumulated budget supply is $S(t_0)$. By definition, the utilization, $U$, is

$$U = \lim_{n \to \infty} U(n) = \lim_{n \to \infty} \frac{S(t_0) + nM}{t_0 + nP} = \frac{M}{P} \Rightarrow M = PU.$$

Using Lemma 4.3, if $S(t + P) - S(t) = PU$ is true for the EDF scheduler, the theorem holds for any work-conserving scheduler. Inequality (6) holds because not all tasks start releasing jobs before $\max_i \{A_i\}$. Since the load of the system is less, the budget supply in a hyperperiod is less. □

**Proof of Theorem 4.2.** The first part of Theorem 4.2 is trivial and can be proved by contradiction: if $W < U$ holds, the partial CPU is overloaded due to a lack of resources and hence cannot achieve the same schedule as a full one.

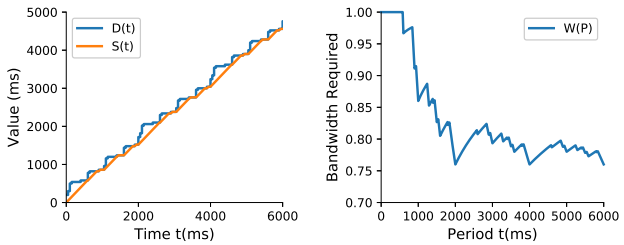
The second part of Theorem 4.2 is $P = LCM\{T_i\} \Rightarrow W = U$, which we prove as follows: From Lemma 4.4, we have $P = LCM\{T_i\} \Rightarrow S(t + P) - S(t) \leq PU$. Using Theorem 4.1, a sufficient condition to achieve the same schedule is to keep the budget supply of any period of a partial CPU not less than the budget supply of the corresponding interval in a full CPU, i.e., $B \geq S(t + P) - S(t)$. So, if $B = PU$, we can guarantee $G(\Gamma, t) = G'(\Gamma, t)$. Thus, we can achieve the minimal bandwidth which equals the task utilization, $W = \frac{B}{P} = U$. The proof for sporadic tasks shares a similar procedure, with equations (2) and (5) being changed to inequalities. □

Theorem 4.2 explicitly indicates how we can configure a partial CPU to both maintain the scheduling of a full CPU and achieve minimal VCPU bandwidth:

Given a taskset $\Gamma = \{\tau_i = (A_i, C_i, T_i) | i = 1, ..N\}$,
1. Compute the hyperperiod $LCM\{T_i\}$;
2. Compute the utilization $U = \sum_i \frac{C_i}{T_i}$;
3. For the partial VCPU with a resource interface $(B, P, R)$,
   let $P \leftarrow LCM\{T_i\}, B \leftarrow PU, R \leftarrow Highest$. □

Theorems 4.1 and 4.2 can be explained from the another point of view. Theorem 4.1 indicates that, given a certain periodic task system within a time-sensitive partial VCPU, we can always avoid budget exhaustion in any period and hence achieve VAL. All we need to do is allocate enough bandwidth for the VCPU. The minimal bandwidth is a function of the period of the VCPU, i.e., $W(P) = \max_t \{S(t + P) - S(t)\}$. Theorem 4.2 states that $W(P)$ achieves its minimum when $P = LCM\{T_i\}$, i.e., $W = \min_P \{W(P)\} = U$. For example, computing $W(P)$ for task system $\Gamma_1$, we illustrate $W(P)$ in Fig. 4b. Any point on this figure refers to a certain resource interface configuration. Using Theorem 4.1, if the point sits on or above the

[5] To clarify, $D(t)$ and $S(t)$ are different from the well-known *demand bound function*, $dbf(t)$ [8] and *supply bound function*, $sbf(t)$ [35], respectively. While demand bound function and supply bound function are typically used for schedulability analysis, we introduce $D(t)$ and $S(t)$ to find the resource interface that avoids budget exhaustion for deferrable server.




(a) D(t) and S(t) (b) W(P)

**Figure 4: Important Curves Related to $\Gamma_1$**

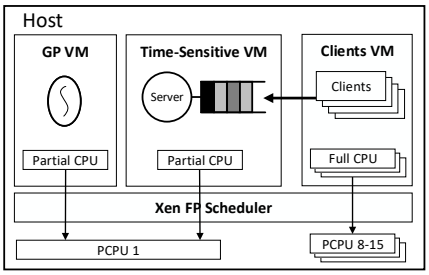



**Figure 5: VAS System Architecture for a Synthetic Application**

solid line, we can keep the virtualization-agnostic property. Using Theorem 4.2, if $P = 2000ms$, which equals the hyperperiod of $\Gamma_1$, we can achieve the minimum of $W(P)$, $W = U = 76\%$.

A potential limitation of the resource interface configuration for the time-sensitive VCPU is that the general-purpose VCPUs sharing the PCPU may be deprived of CPU time for an extended period of time when the hyperperiod or utilization of time-sensitive tasks is large. Hence, VAS assumes the general-purpose tasks to be tolerant of delays. In addition, time-sensitive tasks with harmonic periods help avoid a large hyperperiod.

## 5 SYSTEM IMPLEMENTATION

We have implemented VAS in the Xen hypervisor (4.10.0) on a multi-core host. In the hypervisor, VAS employs a partitioned fixed-priority scheduling policy, and a VCPU needs to be scheduled as a deferrable server. The existing RTDS scheduler in Xen already schedules a VCPU as a deferrable server, but employs a global EDF scheduling policy. Hence we implemented a new partitioned fixed-priority (FP) scheduler in the RTDS scheduler, which was similar to the original partitioned fixed-priority scheduler implemented in RT-Xen [40]. Specifically, we extended `struct rt_vcpu` by adding the field `uint32_t prio`, and modified the function `compare_vcpu_priority()` to compare the `prio` values rather than their absolute deadlines. To support partitioned scheduling, we set a VCPU's PCPU-affinity via the VM configuration file to bind the VCPU to a PCPU at run time. Finally, we modified *libxl* and *libxc* to support an "-r" option for the command "*xl sched-rtds*" so that any VCPU's priority is configurable via Domain 0.

To configure the resource interface of a time-sensitive VCPU in a real system, we need to take into account several system issues that are not modeled in the theoretical results. First, VAS and the VM schedulers incur scheduling and context switching overhead at both the hypervisor and the VM levels. Moreover, interrupt handlers and other kernel services in the VMs may consume additional CPU time. Although those tasks may be modeled as sporadic tasks and thus can be handled by our theoretical analysis, this solution may force us to choose a very large period for the VCPU, because the minimal inter-arrival time of those tasks could be large. We therefore take a pragmatic approach to handle the additional system activities and overhead through moderate overprovisioning. In our experiments with Linux-based VMs, we found that adding 5% more CPU bandwidth to the theoretical bound was sufficient to deliver virtualization-agnostic latency[6].

## 6 EVALUATION

In this section, we present three different sets of experiments for testing the VAS system: (1) synthetic taskset experiments, (2) a case study on Redis, and (3) a case study on Spark Streaming. The synthetic taskset experiments examined how VAS performs in periodic/sporadic task systems with different work conserving schedulers. The two case studies were designed to assess the effectiveness of our approach for real-world applications.

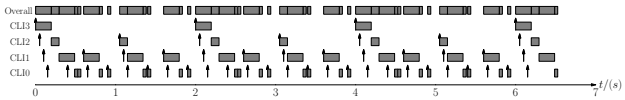
We conducted experiments on a machine with one Intel E5-2683v4 16-core CPU and 64 GB memory. We disabled hyper threading and power saving features and fixed the CPU frequency at 2.1 GHz to reduce unpredictability, as in [21, 40, 41]. We used Xen 4.10.0 with our VAS implementation as the hypervisor scheduler. We used Linux 4.4.19 for all VMs. We configured Domain 0 with one full CPU pinned to one dedicated core, i.e., PCPU 0. We used Redis version 4.0.9, Kafka version 2.0.0, and Spark version 2.3.1.

We mainly concerned with whether a task set within a partial VCPU VM in the VAS system can achieve VAL, i.e., the same latency distributions as those in a full VCPU. We plotted cumulative distribution functions (CDF) to illustrate the difference between the two distributions, and used the *Wasserstein Distance* [37] to quantify differences between two distributions.

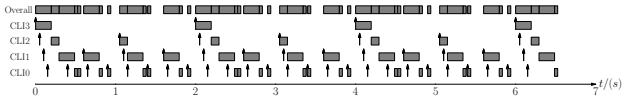
### 6.1 Synthetic Server Evaluation

**System Architecture.** The synthetic task system we used for evaluation is comprised of one server and several clients. Clients dispatch jobs periodically by sending requests to the server within the time-sensitive VM. The server queues incoming jobs, schedules them according to a work-conserving scheduling policy, and provides services to jobs. This architecture pertains to many cloud applications, such as Redis, Spark and FTP services. We used three VMs for this experiment, as Fig. 5 shows: (1) a 1-VCPU general-purpose VM (GP VM), with the VCPU pinned to PCPU 1. (2) a 1-VCPU time-sensitive VM for running the synthetic server, which shares PCPU 1 with the GP VM in a partial CPU configuration. and (3) an 8-VCPU VM for running clients, with each VCPU pinned to a

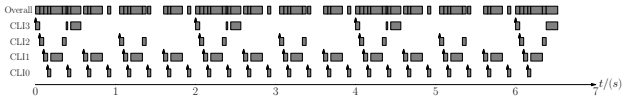
[6]While the amount of overprovisioning is heuristically derived, the same margin is used in Linux [10]. As future work it will be interesting to leverage earlier research on characterizing virtualization overhead [2] to configure the overprovisioning systematically.



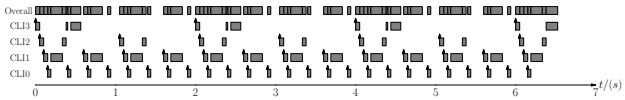

(a) Full CPU (FIFO)




(b) Partial CPU (FIFO)

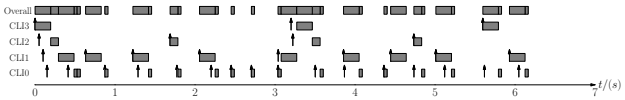



(c) Full CPU (FP/RM)

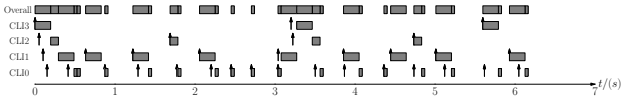



(d) Partial CPU (FP/RM)

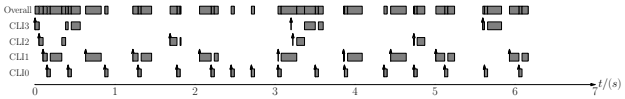
**Figure 6: Periodic Tasks: Schedule Comparison**



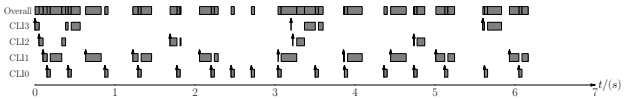

(a) Full CPU (FIFO)




(b) Partial CPU (FIFO)

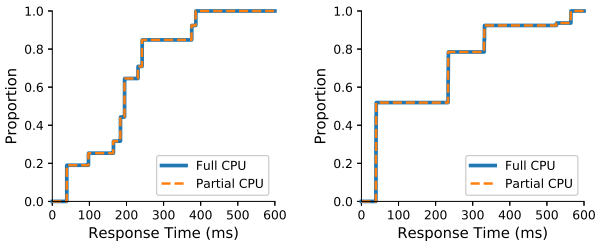



(c) Full CPU (FP/RM)



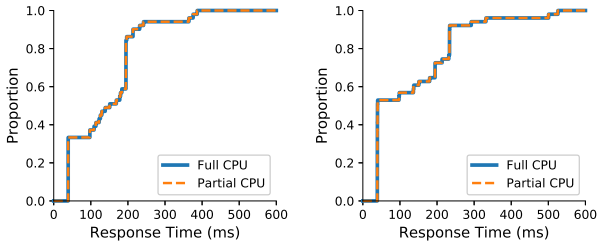

(d) Partial CPU (FP/RM)

**Figure 8: Sporadic Tasks: Schedule Comparison**




(a) FIFO (b) FP/RM

**Figure 7: Periodic Tasks: Empirical CDF Comparison**




(a) FIFO (b) FP/RM

**Figure 9: Sporadic Tasks: Empirical CDF Comparison**

distinct PCPU (from PCPU 8 to PCPU 15). In the VM for the clients, to make the clients as independent as possible, we assigned clients to VCPUs in a round-robin fashion. The VM for the clients does not share PCPU 8-15 with the other VMs.

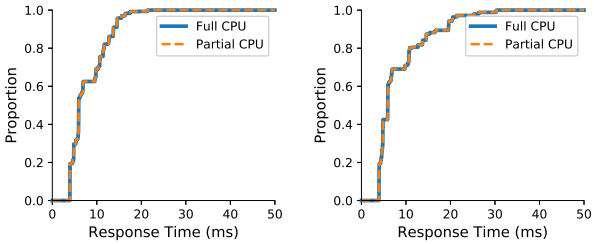
For the partial CPU VM configuration, we configured the resource interface of the VCPU in the time-sensitive VM based on Theorem 4.2. The GP VM, which was created to share the PCPU with the time-sensitive VM, ran a purely CPU intensive workload to consume CPU cycles when possible. For the full CPU VM configuration, we did not run the GP VM: setting the VCPU in the time-sensitive VM to have full bandwidth, we let the time-sensitive VM occupy PCPU 1 exclusively.

We measured the response time of each job for different combinations of settings: periodic / sporadic tasks, harmonic / non-harmonic tasks, and FIFO / FP schedulers. We then also tested our system extensively by using randomly generated test cases.

**Periodic Tasks.** We used a harmonic periodic task setting with a free offset, $\Gamma_1$ (defined in section 4), whose hyperperiod is 2000 ms. According to Theorem 4.2, the minimal VCPU bandwidth equals the utilization (76%) of the taskset: the interface $(B, P)$ of the partial CPU is (1520 ms, 2000 ms).

We ran the experiment for 100 seconds for both the full and the partial CPU setting on both the FIFO and FP scheduler. For the FP scheduler, we used a rate monotonic priority assignment. We show the first 6 seconds in Fig. 6, and make three observations: (1) The schedules of the partial CPU approximate the schedules of the full CPU well, in either a FIFO job scheduler (Fig. 6a and 6b) or a FP scheduler (Fig. 6c and 6d). (2) Each schedule repeats itself every hyperperiod. (3) The server's "active states" are identical between the two work-conserving schedulers ("Overall" traces of Fig. 6b and 6d), which supports the claim in Lemma 4.3: any work-conserving scheduler can yield the same $S(t)$.

Fig. 7 shows the latency distribution of two different VCPU settings in either the FIFO or FP scheduler. We observe that (1) The "stepwise" feature of the CDFs indicates the hyperperiodicity of the system. (2) In either Fig. 7a or 7b, the partial CPU VM can appropriately achieve VAL: the CDF curves can hardly be differentiated




(a) FIFO (b) FP/RM

**Figure 10: Non-Harmonic Periodic Tasks, Empirical CDF**

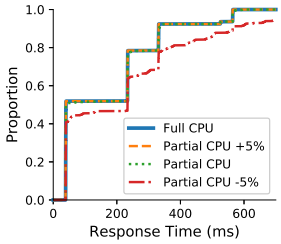



**Figure 11: CDF, Different RI**

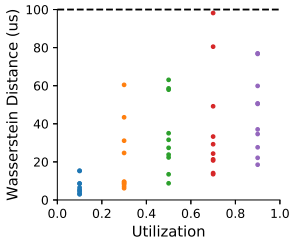



**Figure 12: Multi-Testcase**

visually, so we used Wasserstein distance to distinguish them, as report later under *Practical Overprovisioning*.

**Sporadic Tasks.** We modified task system $\Gamma_1$ for the sporadic test, with the minimal job arrival interval of each sporadic task being the same as the period of the corresponding periodic task. Each job arrival interval is generated from a random variable with a uniform distribution over $[P, 2P]$. The random inter-arrival times are generated offline, and thus we can use the same "input" for different runs and make comparisons among them. From the first 6 seconds of the schedule shown in Fig. 8, we observe that: (1) The schedules of the partial CPU approximate the schedules of the full CPU well. (2) Because of the randomized inter-arrival times, each schedule no longer repeats itself. (3) The servers follow the same activity patterns for two different work-conserving schedulers (FIFO and FP).

In Fig. 9, we again observe overlapping CDF curves: thus, we can appropriately achieve VAL in a **sporadic** setting as well as in a periodic one.

**Non-Harmonic Periodic Tasks.** We also tested non-harmonic period settings: $\Gamma_2 = \{\tau_i = (A_i, C_i, T_i) | i = 0, 1, 2, 3\}$, where the parameters (in *ms*) of four tasks are (15, 4, 20), (10, 6, 30), (5, 10, 50), and (0, 7, 70), respectively. The hyperperiod is 2100*ms*, which is much larger than each task's period. From the overlapping curves shown in Fig. 10, we can conclude that our system is also effective for a non-harmonic setting. A non-harmonic task system can potentially incur a large hyperperiod (with a proportionally large budget), especially for co-prime intervals. As a result, it may prevent a low-priority VCPU from running for a relatively long time. Hence, VAS is more suitable when time-sensitive VMs with harmonic tasks or when general-purpose tasks are tolerant of delays.

**Practical Overprovisioning.** Using a FP scheduler with RM priority assignment for synthetic server, we tested the periodic task system $\Gamma_1$ with different VCPU bandwidth configurations. The task set utilization was 76%. We repeated the periodic task experiment for 100 seconds each run, under three bandwidth settings: 71% (overload), 76% (theoretical minimum), and 81% (overprovisioned). We plotted the CDF curves in Fig. 11. The Wasserstein distances of each setting were 67.169 ms, 1.3491 ms, and 65.038 $\mu$s.

These results support three main observations: (1) The overload setting (-5% BW) deviates significantly from the full CPU's CDF because the system is overloaded. If the system were run indefinitely, the pending jobs would accumulate and the queue would never be emptied. Even running the system for a finite time (100 seconds), the latency distribution shows a very long tail, with a max latency up to 1457.2 ms. (2) The overprovisioned setting (+5% BW) outperforms the others by giving the best approximation, with a Wasserstein distance of 65.038 $\mu$s. (3) The theoretical minimal bandwidth setting approximates the full CPU well. The Wasserstein distance to the full CPU distribution was 1.3491 ms. The system overhead can potentially induce temporary overload of a VCPU and thus deteriorate the latency performance of the system. Hence, we suggest a mild VCPU bandwidth overprovision (by 5%) when adopting our configuration in real-world systems.

**Multi-Testcase Evaluation.** We generated randomized testcases for our system. Given a desired utilization: (1) We uniformly picked a period $P_i$, in milliseconds, from the harmonic set {10, 20, 40, 80, 160, 320, 640, 1280}. (2) We uniformly generated an offset, $A_i$, over $[0, P_i]$. (3) We also generated a utilization, $U_i$, for this task, following a medium bimodal distribution, which distributed uniformly over $[0.0001, 0.5)$ with probability of 2/3, or $[0.5, 0.9]$ with probability of 1/3 as was done in prior work [40]. We then calculated the WCET, $C_i = P_i U_i$. (4) We repeated steps (1) through (3) to generate more tasks as long as the total utilization was less than desired, then trimmed down the last task to fit the desired total utilization (if necessary), and terminated the procedure.

We generated ten testcases for each of the following utilization settings: 0.1, 0.3, 0.5, 0.7, and 0.9. We ran each testcase for 100s, for both partial and full CPU settings. Fig. 12 shows that the Wasserstein distances of all testcases are lower than 100 $\mu$s, indicating that the partial CPU configuration approximates the full CPU configuration well for these randomized task sets. We also observed that high utilization settings tended to yield greater distances: the higher the desired utilization, the higher the average task numbers, and the greater the likelihood of long period tasks. Those long period tasks generated fewer samples with the same duration in a single run. As a result, we have a higher chance to find a large Wasserstein distance, since fewer samples can deviate significantly (higher p-value). Longer period tasks had lower priorities (under RM priority assignment), and thus were more likely to be preempted by higher priority tasks. The execution time jitter of short period tasks also may accumulate, and affect the response time of a long period task.

## 6.2 Case Study: Redis

**Redis** is a widely used single-threaded [32] in-memory data storage server. Redis is typically used within a virtualized host such as AWS ElastiCache [4]. In the following experiments, we employed Redis as a real-world time-sensitive server to test our VAS system design.

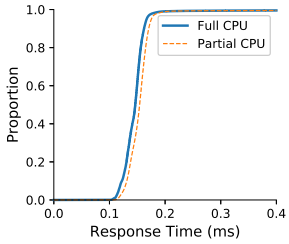



Figure 13: Single Redis CDF

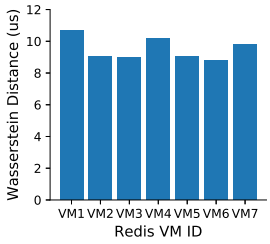



Figure 14: Multi-Tenant

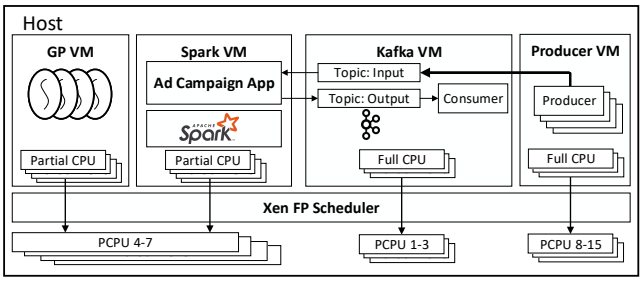



Figure 15: Applying VAS to a Spark Streaming Application

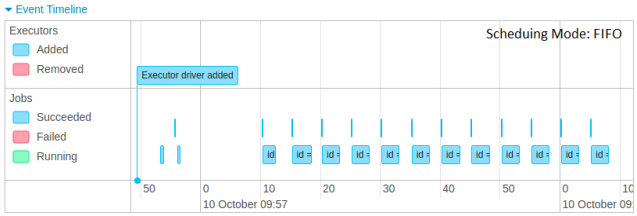



Figure 16: Periodic Processing Pattern of Spark

**Single Redis VM.** The testbed for this experiment was similar to the synthetic evaluation architecture shown in Fig. 5, except the synthetic server was replaced with a Redis server. Clients sent "HSET" queries (which are used in the Structured Yahoo Streaming Benchmark [5]) to the Redis server. We leveraged `libhiredis` to implement the querying clients. The WCET of the "HSET" query was 0.12 ms in our system. We used six periodic clients that each sent one "HSET" query with a random key-value every 1ms. Thus, the utilization was 72%. We overprovisioned the bandwidth by 5%, which meant the GP VM could consume at least 23% of the CPU cycles per CPU. We ran the system for 100s. From the CDF curves in Fig. 13 and the Wasserstein distance of 9.499$\mu$s, we can conclude that the partial CPU configuration in VAS also effectively approximated the full CPU configuration in the Redis test.

**Multi-Redis-Tenant on PCPUs.** Our system can be applied to a multi-core host by using a partitioned approach. On the remaining 15 cores in our host, we created seven single VCPU Redis VMs, with each VCPU pinned to a PCPU from among PCPUs 1-8. Another 7-VCPU GP VM, running seven CPU-intensive processes, shared PCPUs 1-8 with the Redis VMs. We used the other eight PCPUs for a client VM and ran seven groups of client processes, with six clients in each group. Each client sent a HSET query (WCET = 0.12ms) to one Redis server instance every 1 ms. The utilization of each Redis server was 72%. We ran the system for 100s, collecting the response times corresponding to each Redis VM. We then computed the Wasserstein distance between the distributions for the full and partial CPU settings.

As Fig. 14 illustrates, all seven Redis VMs show similar performance, with a consistent Wasserstein distance around 10$\mu$s. We can conclude that our design remained effective in a multi-tenant scenario. This experiment yielded a relatively better result (10$\mu$s distance) than that shown in Fig. 12. This phenomenon is due to our use of the same execution time and period settings as in the single Redis evaluation, while varying the period and execution time in the synthetic experiments.

## 6.3 Case Study: Spark Streaming

**Spark Streaming** is a popular streaming and data analytics engine, often run on virtualized hosts. For example, the *Structured Yahoo Streaming Benchmark* [5], an open-source real-time advertisement campaign application, is deployed on the DataBricks' platform running on AWS. In this experiment, we evaluated whether our system model can be easily extended and adopted for a Spark application.

**System Architecture.** We created four VMs for this experiment, as Fig. 15 shows: (1) a 4-VCPU VM for Spark, with each VCPU pinned to a PCPU from among PCPUs 4-7, (2) a 4-VCPU GP VM, which shares PCPUs 4-7 with the Spark VM, (3) a 3-VCPU VM for a Kafka message broker, consumer, and Zookeeper, with each VCPU pinned to a PCPU from PCPU 1-3, and (4) a 8-VCPU VM for producers, with each VCPU pinned to a PCPU from PCPUs 8-15.

The Spark VM ran an advertisement campaign application. We modified DataBricks' *Structured Yahoo Streaming Benchmark* [5] under realistic conditions. We set Spark to operate in a local mode with four workers. Each worker ran on one VCPU in a Spark VM. We set the shuffle partitions to 4. The Spark scheduler worked in FIFO mode. We used an 8-VCPU VM for producers, which produced and published advertisement-events to a Kafka "input" topic, at a rate of 28,000 events/s. Those events were then consumed by Spark.

**Periodic Processing.** Spark features micro-batch processing: The arriving events will not be processed until a periodic micro-batch ends. As a result, Spark processes the incoming events periodically, regardless of the events' arrival pattern. We set the micro-batch window size to 5 seconds via the `writeStream.trigger()` method. Fig. 16 is a run-time screenshot of the Spark Web UI when we ran Spark with a full CPU setting. We observed periodic behavior: Spark processed the incoming events every 5 seconds. Effectively, each VCPU within the Spark VM ran a periodic task with a period of 5 seconds. Moreover, this observation was true for every worker in Spark, since the Spark Driver triggered the processing of each worker every 5 seconds. Effectively, each VCPU within Spark VM ran a periodic task with a period of 5 seconds.

**Execution Time.** Configuring the Spark VM with full-VCPUs, we measured the Spark jobs' elapsed times, which provide a safe estimation of the execution time for each Spark worker. With a total input rate of 28,000 event/s, a 5-second batch-window, four workers (i.e.,

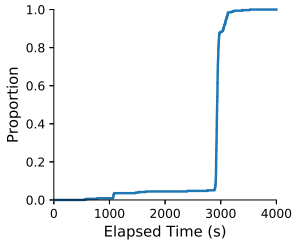



**Figure 17: Elapsed Time**

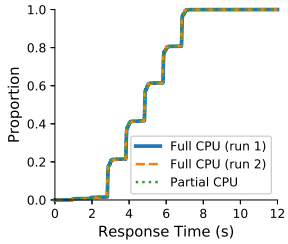



**Figure 18: Spark CDF**

four periodic tasks) on four PCPUs, Fig. 17 shows the distribution of elapsed time. The maximum elapsed time was 3522 ms.

**Partial CPU Setting.** We used 3522 ms and 5000 ms as the WCET and period for each Spark task, respectively. The utilization for each worker on each VCPU was 70.44%. We set 75% bandwidth for each VCPU in the Spark VM (overprovisioning by 4.56%). Each VCPU in the GP VM took the remaining 24%. The final resource interface for each VCPU in Spark VM was (3750 ms, 5000 ms, Highest).

Note that the Spark experiment was more realistic and yet introduced more sources of variation: (1) The advertisement events were randomly generated for different runs independently. (2) We cannot control the Spark internal or guarantee it handles tasks exactly the same in different runs. (3) Compared to the single-threaded Redis and synthetic server platforms, the Spark platform needs to manage additional threads (e.g., for Spark Web Service, and Spark Driver).

Hence, the absolute value of the Wasserstein distance was much larger than in either the synthetic experiments or the Redis case study, making it hard to tell whether results violate our claims. Thus, we need two full CPU runs and compare the Wasserstein distance between them to indicate the impact of those variations, as a baseline. Then, we made another run in partial CPU setting, to observer whether the Wasserstein distance between partial and full CPU runs significantly exceeds the baseline of two full CPU runs.

We made three runs, each for 26 minutes, two with the full CPU setting, and the other with the partial CPU setting. We compared two runs with **the same full CPU setting**, where the Wasserstein distance was 9.190 ms. Ironically, the partial CPU setting can yield an even better Wasserstein distance of 8.378 ms. Thus, we can still maintain that the partial CPUs in VAS system achieve comparable VAL, as the results in Fig. 18 indicate that the latency distribution of the partial CPU again closely approximates that of the full CPU.

## 7 RELATED WORK

Recent years have witnessed significant research on VCPU scheduling and resource allocation for real-time virtualization systems. A multitude of scheduling approaches have been explored for real-time VCPU scheduling. The Quest-V separation kernel [26] schedules each process in a sandbox as a deferrable server, and takes budget replenishment delay into consideration for predictable communications among sandboxes. vMPCP [22] provides a partitioned hierarchical scheduling framework based on deferrable servers and a synchronization protocol based on the Resource Kernel [31]. RT-Xen [40] developed real-time schedulers in the Xen hypervisor (including the RTDS scheduler adopted in the hypervisor). A predictable VM scheduling framework has also been developed based on standard qemu/KVM, and Linux SCHED_DEADLINE scheduler [1]. RTVirt [42] introduced cross-layer scheduling by sharing scheduling metadata between the hypervisor and the OS scheduler within a VM. These systems leveraged real-time scheduling analysis to provide guarantees in terms of meeting task deadlines. They are not designed to meet probabilistic tail latency target or predictable latency distributions. In contrast, VAS is, to our knowledge, the first scheduling framework to provide VAL for time-sensitive applications on partial CPUs. Based on the predictable latency distribution, it is hence straightforward to achieve tail latency guarantees with VAS on multiple edge clouds. A limitation of VAS is that it can support only one time-sensitive VCPU per PCPU (shared with multiple general-purpose VCPUs), and hence up to $m$ VCPUs on an $m$-core processor. VAS is therefore suitable for managing edge clouds each co-hosting a mix of numerous general-purpose workloads and a small set of time-sensitive services with stringent tail latency SLO.

Tableau [38] provides a scalable scheduling framework based on dispatching tables that can be generated on-demand. With low and predictable scheduling overhead, Tableau helps reduce tail latency, but it is not designed to achieve tail latency guarantees or VAL. Tableau is specifically designed for efficient scheduling of high-density workloads. In contrast, VAS focuses on providing VAL for a small number of time-sensitive VCPU sharing CPUs with (potentially larger numbers of) general-purpose VCPUs. Tableau and VAS therefore complement each other, and it will be interesting explore approaches to combine their advantages.

While VAS achieves predictable latency distributions for periodic and sporadic workloads on partial CPUs, it cannot provide the same guarantee for aperiodic services. The problem of predicting latency distributions of aperiodic services on virtualized platforms has been studied theoretically [24]. This work adopted a similar system model as VAS but assumes a single time-sensitive service on each PCPU, while VAS can accommodate multiple time-sensitive services on a single time-sensitive VCPU for each PCPU. For a time-sensitive service with Poisson arrivals and scheduled as a deferrable server, a queueing model is proposed to predict the latency distribution of the time-sensitive service. The scheduling framework proposed in [18] extended the SAF model [13] by allowing aperiodic tasks to run within polling servers. An important research direction is to extend the VAS framework to support aperiodic time-sensitive services based on the theoretical models.

## 8 CONCLUSIONS

We introduce virtualization-agnostic latency (VAL) as a desirable property for deploying and managing time-sensitive applications on different edge clouds. In a system providing VAL, a time-sensitive application can maintain similar latency distributions on a partial CPU as on a dedicated CPU, thereby alleviate the significant effort of testing, tuning, and configuring a time-sensitive service for targeted tail latency on numerous edge clouds. We present virtualization-agnostic scheduling (VAS), a simple and effective approach for achieving VAL on a host shared by time-sensitive and

general workloads. Two case studies involving commonly used time-sensitive cloud services, Redis and Spark Streaming, demonstrated the efficacy of VAS in achieving VAL in virtualized environments.

While this work has taken the first step in supporting VAL, there are several promising directions for future work. First, in addition to CPU scheduling, VAL will also require scheduling support for non-CPU resources. Second, it will be interesting extend and leverage the analysis in [24] to support aperiodic time-sensitive tasks. Finally, to generalize the VAS approach to heterogeneous platforms, VAS can be extended to tailor the resource interfaces based on the execution times on different platforms and avoid throttling and preemption of time-sensitive VMs, thereby achieving predictable latency distribution on heterogeneous platforms.

## ACKNOWLEDGMENTS

This research was sponsored, in part, by NSF through grant 1646579 (CPS), by ONR through grant N00014-20-1-2744, and by the Fullgraf Foundation.

## REFERENCES

[1] Luca Abeni, Alessandro Biondi, and Enrico Bini. 2019. Hierarchical scheduling of real-time tasks over Linux-based virtual machines. *Journal of Systems and Software* 149 (2019), 234–249.

[2] Luca Abeni and Dario Faggioli. 2019. An Experimental Analysis of the Xen and KVM Latencies. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, Valencia, Spain, 18–26.

[3] Luis Almeida and Paulo Pedreiras. 2004. Scheduling within temporal partitions: response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software*. ACM, Pisa, Italy, 95–103.

[4] Amazon.com Inc. 2018. Amazon ElastiCache for Redis. https://aws.amazon.com/elasticache/redis/.

[5] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, Houston, TX, USA, 601–613.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[7] Sanjoy Baruah and Nathan Fisher. 2009. Component-based design in multiprocessor real-time systems. In *2009 International Conference on Embedded Software and Systems*. IEEE, Washington, DC, USA, 209–214.

[8] Sanjoy K Baruah, Louis E Rosier, and Rodney R Howell. 1990. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-time systems* 2, 4 (1990), 301–324.

[9] Matthias Beckert and Rolf Ernst. 2017. Response time analysis for sporadic server based budget scheduling in real time virtualization environments. *ACM TECS* 16, 5s (2017), 161.

[10] Jonathan Corbet. 2008. SCHED_FIFO and realtime throttling. https://lwn.net/Articles/296419/.

[11] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[13] José Luis Díaz, Daniel F García, Kanghee Kim, Chang-Gun Lee, L Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. 2002. Stochastic analysis of periodic real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* IEEE, Austin, Texas, USA, 289–300.

[14] Arvind Easwaran, Insik Shin, and Insup Lee. 2009. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems* 43, 1 (2009), 25–59.

[15] Song Han, Tao Gong, Mark Nixon, Eric Rotvold, Kam-Yiu Lam, and Krithi Ramamritham. 2018. Rt-dap: A real-time data analytics platform for large-scale industrial process monitoring and control. In *2018 IEEE International Conference on Industrial Internet (ICII)*. IEEE, Bellevue, WA, USA, 59–68.

[16] RedHat Inc. 2018. Enabling RT-KVM For NFV. https://access.redhat.com/.

[17] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, London, UK, 435–448.

[18] Giordano A Kaczynski, Lucia Lo Bello, and Thomas Nolte. 2007. Deriving exact stochastic response times of periodic tasks in hybrid priority-driven soft real-time systems. In *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*. IEEE, Patras, Greece, 101–110.

[19] Robert Kaiser. 2011. Applicability of virtualization to embedded systems. In *Solutions on Embedded Systems*. Springer, New York, USA, 215–226.

[20] Robert Kaiser and Dieter Zöbel. 2009. Quantitative analysis and systematic parametrization of a two-level real-time scheduler. In *2009 IEEE Conference on Emerging Technologies & Factory Automation*. IEEE, Palma de Mallorca, Spain, 1–8.

[21] Hyoseung Kim and Ragunathan Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*. IEEE, Pittsburgh, PA, USA, 1–10.

[22] Hyoseung Kim, Shige Wang, and Ragunathan Rajkumar. 2014. vMPCP: A synchronization framework for multi-core virtual machines. In *2014 IEEE Real-Time Systems Symposium*. IEEE, Rome, Italy, 86–95.

[23] Joseph Y-T Leung and ML Merrill. 1980. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters* 11, 3 (1980), 115–118.

[24] Haoran Li, Chenyang Lu, and Chistopher Gill. 2019. Predicting Latency Distributions of Aperiodic Time-Critical Services. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, Hong Kong, China, 30–42.

[25] Haoran Li, Meng Xu, Chong Li, Chenyang Lu, Christopher Gill, Linh Phan, Insup Lee, and Oleg Sokolsky. 2018. Multi-mode virtualization for soft real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Porto, Portugal, 117–128.

[26] Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. 2014. Predictable communication and migration in the Quest-V separation kernel. In *2014 IEEE Real-Time Systems Symposium*. IEEE, Rome, Italy, 272–283.

[27] Giuseppe Lipari and Enrico Bini. 2003. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.* IEEE, Porto, Portugal, 151–158.

[28] Shokunin Consulting LLC. 2014. Running Redis in Production. http://shokunin.co/blog/2014/11/11/operational_redis.html.

[29] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, Portland, Oregon, USA, 450–462.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. USENIX, Lombard, IL, 385–398.

[31] Shuichi Oikawa and Ragunathan Rajkumar. 1999. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the fifth IEEE real-time technology and applications symposium*. IEEE, DC, USA, 111–120.

[32] RedisLab. 2020. Introduction to Redis. https://redis.io/.

[33] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE IoT Journal* 3, 5 (2016), 637–646.

[34] Insik Shin, Arvind Easwaran, and Insup Lee. 2008. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*. IEEE, Prague, Czech Republic, 181–190.

[35] Insik Shin and Insup Lee. 2003. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003.* IEEE, Cancun, Mexico, 2–13.

[36] Insik Shin and Insup Lee. 2008. Compositional real-time scheduling framework with periodic model. *ACM TECS'08* 7, 3 (2008), 30.

[37] SS Vallender. 1974. Calculation of the Wasserstein distance between probability distributions. *Theory of Probability & Its Applications* 18, 4 (1974), 784–786.

[38] Manohar Vanga, Arpan Gujarati, and Björn B Brandenburg. 2018. Tableau: a high-throughput and predictable vm scheduler for high-density workloads. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, NY, USA, 1–16.

[39] VMWare LLC. 2015. Deploying Extremely Latency-Sensitive Applications in VMware. https://www.vmware.com/techpapers.html.

[40] Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. 2014. Real-time multi-core virtual machine scheduling in xen. In *2014 International Conference on Embedded Software (EMSOFT)*. IEEE, New Delhi, India, 1–10.

[41] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. 2017. vcat: Dynamic cache management using cat virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Pittsburgh, PA, USA, 211–222.

[42] Ming Zhao and Jorge Cabrera. 2018. RTVirt: enabling time-sensitive computing on virtualized systems through cross-layer CPU scheduling. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, Porto, Portugal, 1–13.

[43] Timothy Zhu, Daniel S Berger, and Mor Harchol-Balter. 2016. SNC-Meister: Admitting more tenants with tail latency SLOs. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, New York, NY, USA, 374–387.