Unikernel-Based Real-Time Virtualization Under Deferrable Servers: Analysis and Realization

Kuan-Hsun Chen
University of Twente, The Netherlands

Mario Günzel
TU Dortmund University, Germany

Boguslaw Jablkowski
EMVICORE GmbH, Dortmund, Germany

Markus Buschhoff
EMVICORE GmbH, Dortmund, Germany

Jian-Jia Chen
TU Dortmund University, Germany

Abstract

For cyber-physical systems, real-time virtualization optimizes the hardware utilization by consolidating multiple systems into the same platform, while satisfying the timing constraints of their real-time tasks. This paper considers virtualization based on unikernels, i.e., single address space kernels usually constructed by using library operating systems. Each unikernel is a guest operating system in the virtualization and hosts a single real-time task.

We consider deferrable servers in the virtualization platform to schedule the unikernel-based guest operating systems and analyze the worst-case response time of a sporadic real-time task under such a virtualization architecture. Throughout synthesized tasksets, we empirically show that our analysis outperforms the restated analysis derived from the state-of-the-art, which is based on Real-Time Calculus. Furthermore, we provide insights on implementation-specific issues and offer evidence that the proposed scheduling architecture can be effectively implemented on top of the Xen hypervisor while incurring acceptable overhead.

2012 ACM Subject Classification Computer systems organization → Embedded and cyber-physical systems; Software and its engineering → Real-time systems software

Keywords and phrases Unikernel, Virtualization, Reservation Servers, Deferrable Servers, Cyber-Physical Systems, Real-Time Systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.6

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.1.2`

Funding This work has been supported by Deutsche Forschungsgemeinschaft (DFG), as part of Sus-Aware (Project No. 398602212). This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170).

1 Introduction

Virtualization technology has been widely and successfully used in data centers and cloud environments to consolidate multiple systems as virtual machines (VMs) into the same platform (so-called host). Due to the increasing use of multi-core processors in embedded and cyber-physical systems (CPS's), platform virtualization now is gaining traction also in these domains [15], since it allows for cost reduction, increases efficiency and enhances flexibility.

However, virtualization technology was initially not designed to cope with strict timing constraints and those are inherent to CPS's. In such systems, meeting timing requirements (so-called *timeliness*) is as important as the functional correctness. Depending on the applications, a deadline miss may result in lower service quality or even a catastrophic system failure in the worst case. Thus, virtualization must be compatible to real-time software stack and satisfy the time constraints by employing hypervisor-level real-time scheduling policies.

In order to satisfy timing requirements in virtualized environments, e.g., the popular Xen hypervisor [36, 37], periodic server-based approaches have been widely used – especially *deferrable servers* [33]. For instance, the real-time deferrable server (RTDS) scheduler [35, 36] has been officially[1] supported in Xen since 2015, by which each virtual CPU (vCPU) is treated as one deferrable server assigned with an execution budget and a (replenishment) period to serve its corresponding VMs. The budget of the vCPU is consumed only when a task is running on the vCPU. To ensure each real-time task is served sufficiently, corresponding timing analyses [3, 31, 30] should be employed to compute the required capacity for each server budget. Under such setups, the scheduling decision involves two levels, the hypervisor scheduler and the schedulers within the VMs, in a hierarchical manner.

To account for the interplay of servers and tasks, the applicability of such server-based approaches is based on the tightness of corresponding worst-case response time analyses. Note that a task system deployment under over pessimistic analyses may lead to unnecessarily low hardware utilization. For deferrable servers, Saewong et al. developed a sufficient schedulability analysis based on an assumption that server capacity is made available at the very end of the server's period in the worst case [28]. Davis and Burns further developed an exact test to ultimately optimize the schedulability of deferrable servers [8].

However, they also showed that the schedulability of deferrable servers is worse than the other server-based approaches, like periodic servers [29] and sporadic servers [32], due to the well-known phenomenon of *back-to-back* hits [8, 6], i.e., the interference introduced by the suspension of higher priority servers. As the state-of-the-art, Cuijpers and Bril in [7] discussed that the schedulability of deferrable servers is possible to outperform periodic servers and sporadic servers, if one deferrable server only serves one single task. Under such a constraint, the behavior of the deferrable server is no longer influenced by the presence of low-priority tasks. However, a general task model is considered based on real-time calculus [34].

As reported in an empirical study [1], the periodic task activation and the sporadic activation with minimum inter-arrival time are a common industry practice, i.e., 82% and 47% respectively over the investigated systems, and different types of task activation might be involved in the same systems. Thus, it is practically relevant whether the worst-case response time analysis from [7] can be further tightened for a periodic or sporadic task.

Contributions. In this work, we explore *deferrable servers for unikernel-based virtualization*, in which each server serves only one single sporadic task on a virtualized platform. Specifically, we develop the corresponding worst-case response time analysis under fixed-priority scheduling. In practice, we leverage the concept of *unikernel* to motivate and realize the proposed scheduling architecture. In a nutshell, the contributions of this work are as follows:

- We present why unikernel-based virtualization can facilitate the schedulability of deferrable servers. Specifically, we present how to convert the analysis proposed by Cuijpers and Bril in [7] to sporadic tasks served by deferrable servers. Under one practical scenario, our worst-case response time analysis dominates the state-of-the-art (see Section 4).

[1] `https://wiki.xenproject.org/wiki/Xen_Project_Schedulers`

Table 1 Notation used in this paper.

Symbol	Definition
$\tau_i = (T_i, C_i)$	Sporadic task
T_i	Minimum inter-arrival time
C_i	Worst-case execution time
$\mathrm{DS}_i = (P_i, Q_i)$	Deferrable server
P_i	Replenishment period
Q_i	Capacity
$R^{-}{}_i^{\mathrm{DS}}(x)$	Worst-case *response* time for x time units on DS_i
$R^{+}{}_i^{\mathrm{DS}}(x)$	Worst-case *resumed* time for x time units on DS_i

- Secondly, we explain how to realize our unikernel-based approach on top of the Xen hypervisor with a few design details. Under the proposed deferrable server model, we implement our own hypervisor scheduler and keep the routines of scheduler bookkeeping and budget replenishment as efficient as possible (see Section 5).
- Finally, we compare our analysis with the state-of-the-art [7] with synthetic periodic task systems. The results show that the applicability of deferrable servers indeed can be greatly improved (see Section 6.1). In addition, we also conduct a case study based on the Xen hypervisor with our deferrable server model and show that our approach of unikernel-based virtualization is feasible in practice (see Section 6.2).

2 Deferrable Server and Task model

We consider that deferrable servers [33] are adopted to preserve the required bandwidth of each virtualized CPS application, so-called virtual machine (VM). Since each VM is realized as a unikernel with one specific vCPU, each VM is treated as one deferrable server DS_i serving only one single sporadic task τ_i. A sporadic task τ_i releases an infinite number of task instances, called jobs, in which the worst-case execution time (WCET) of any of them is at most C_i and the arrival times of any two consecutive jobs of them must be separated by at least the minimum inter-arrival time T_i. The jobs of task τ_i are served based on the first-come first-serve policy within DS_i. Please note that our worst-case response time analysis does not require any limitation on the type of task deadlines, i.e., even arbitrary deadline tasks with deadline $D_i > T_i$ are allowed.

Each deferrable server DS_i is denoted as a tuple (P_i, Q_i), where P_i is its replenishment period and Q_i is its capacity. If the j-th replenishment time is t, then the next replenishment time is $t + P_i$. The budget of a deferrable server DS_i is set to Q_i initially and is consumed linearly while the corresponding task τ_i is served. When the budget becomes 0, the server DS_i needs to wait until the next replenishment time. At the time instant to replenish the server DS_i, the budget is replenished to Q_i and any unused time budget is lost at the end of each replenishment period. The first job release of every task τ_i is assumed to be after the first budget replenishment of DS_i.

Deferrable servers are scheduled based on preemptive fixed-priority (static-priority) scheduling. For a multiprocessor platform, it is possible to apply partitioned or global scheduling for the deferrable servers. Under a partitioned scheduling paradigm, a deferrable server (vCPU) is dedicated to one physical processor (pCPU). Under a global scheduling paradigm, a deferrable server (vCPU) can migrate from one physical processor to another.

3 Service Condition and Execution Scenarios of Deferrable Servers

In this section we discuss how the service condition of deferrable servers can be sufficiently tested on different systems. By analyzing the possible behaviors of a job served by a deferrable server, two lemmas are derived to provide some useful properties for the next section.

3.1 Service Condition of Deferrable Servers

Under the adopted scheduling paradigm, if the capacity of a deferrable server DS_k within the given replenishment period is feasible, we say that DS_k *fulfills its service condition*. That is, in this case, any request of Q_k amount of computation demand of its served task τ_k at the moment when the budget is fully replenished must be finished within P_k amount of time. This assumption is also required in the related work [7, 8, 28, 20]. Otherwise, one can provide an over-specified capacity Q_k that can never be guaranteed within one period P_k, and the worst-case analysis also has to investigate interplay with the virtualization scheduler.

As a deferrable server DS_i retains its unused budget until the next replenishment period when no job requests it to serve, it can be considered that DS_i voluntarily suspends its execution [6]. As a result, a DS_i may impose back-to-back interference to lower-priority servers (or tasks). Such back-to-back interference can be considered as bursty interference [20, 5], i.e., one additional job should be considered by extending the classical critical instant theorem, or release jitter of DS_i, which can be set to $P_i - Q_i$ [8].

A sufficient service condition test for DS_k is a sufficient test to validate whether DS_k fulfills its service condition or not. Therefore, under uniprocessor preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers, also stated in [8], a sufficient service condition test for the deferrable servers is

$$\forall\, \mathrm{DS}_k,\ \ \exists 0 < t \le P_k, \qquad Q_k + \sum_{\mathrm{DS}_i \in hp_k} \left\lceil \frac{t + P_i - Q_i}{P_i} \right\rceil Q_i \le t \tag{1}$$

where hp_k is the set of deferrable servers whose priorities are higher than DS_k. It has been shown that $\sum_{\mathrm{DS}_i} \frac{Q_i}{P_i} \le \ln \frac{3}{2} \approx 0.40546$ ensures that the condition stated in Eq. (1) holds [5, 20].[2]

For multiprocessor systems, under partitioned scheduling, the condition in Eq. (1) can be directly applied by defining hp_k as the set of higher-priority deferrable servers assigned on the same physical processor as DS_k. Under global preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers on m homogeneous (identical) physical processors, a sufficient service condition test can be written as[3]

$$\forall\, \mathrm{DS}_k,\ \ \exists 0 < t \le P_k, \qquad Q_k + \frac{\sum_{\mathrm{DS}_i \in hp_k} \left\lceil \frac{t+P_i-Q_i}{P_i} \right\rceil Q_i}{m} \le t \tag{2}$$

where hp_k is the set of deferrable servers whose priorities are higher than DS_k under global scheduling and m is the number of homogeneous processors.

Although the sufficient service condition tests in Eqs. (1) and (2) are valid for systems composed of only deferrable servers, they can be extended to consider the co-existence of typical sporadic tasks and other fixed-priority servers by adding corresponding interference terms. More specifically, let $I(t)$ be the worst-case interference of the higher-priority servers and/or tasks for an interval length t. For the rest of this paper, we assume that the sufficient service condition for DS_k is:

$$\exists 0 < t \le P_k, \qquad Q_k + I(t) \le t \tag{3}$$

[2] Their proof in [20] ensures a weaker condition: $\forall\, \mathrm{DS}_k,\ \exists 0 < t \le P_k,\ Q_k + \sum_{\mathrm{DS}_i \in hp_k} \left\lceil \frac{t+P_i}{P_i} \right\rceil Q_i \le t$.

[3] A sketched proof is provided in Appendix for completeness.

In order to analyze the worst-case response time of a sporadic task τ_k served by DS_k, we further need two additional properties based on finer granularity of the service provided by DS_k. In particular, the worst-case response time $R^{-\text{DS}}_k(x)$ for requesting x amount of computation demand, for $0 \leq x \leq Q_k$, can be derived as:

$$R^{-\text{DS}}_k(x) = \inf \{t | x + I(t) = t\} \tag{4}$$

Moreover, right after finishing x amount of computation demand, the deferrable server DS_k may be preempted by other higher-priority activities. We further define the *worst-case resumed time* $R^{+\text{DS}}_k(x)$, for $0 \leq x < Q_k$, as the longest time that DS_k finishes x amount of computation demand and is scheduled to serve further demands if they exist. That is:

$$\begin{aligned} R^{+\text{DS}}_k(x) &= \inf \{t | (x + \epsilon) + I(t) = t\} \text{ for infinitesimal } \epsilon > 0 \\ &= \inf \{t | x + I(t) < t\} \end{aligned} \tag{5}$$

We note that $R^{+\text{DS}}_k(Q_k)$ is not defined above, as it is unnecessary in our analysis, and the proper definition involves more complications.

Since $I(t)$ is usually of the form $\sum_i \left\lceil \frac{t+j_i}{P_i} \right\rceil \rho_i$, where j_i and ρ_i are some real values like $j_i = (P_i - Q_i)$ and $\rho_i = Q_i$ in Equation (1), the exact value of $R^{-\text{DS}}_k(x)$ can be computed using fixed-point iterations, where t is increased gradually until $x + I(t) = t$ is reached. The exact value of $R^{+\text{DS}}_k(x)$ can be obtained by fixed-point iterations as well, if the standard interference function $I(t)$ is replaced by $\sum_i \left(\left\lfloor \frac{t+j_i}{P_i} \right\rfloor + 1 \right) \rho_i$.

Please note that the sufficient service condition test, i.e., Eq. (3), is only introduced to give an intuition behind the definition of $R^{-\text{DS}}_k(x)$ and $R^{+\text{DS}}_k(x)$. However, our analysis is not limited to the scenarios where Eq. (3) holds, but only to all possible scenarios where the service condition holds, i.e., any request of Q_k amount of computation demand must be finished within P_k amount of time when the budget is fully replenished. If this condition is ensured by any means, our timing analysis is applicable.

3.2 Serving one Task by a Deferrable Server

In this section we have a closer look at how a deferrable server DS_k serves a job J released by the task τ_k at time r_J. At the job release of J there may be unfinished backlog $L(r_J)$, i.e., unfinished execution demand from previously released jobs of τ_k. Moreover, we denote by $C(r_J) \leq C_k$ the computation demand of J at its release r_J. Whenever there is available budget $B(t) > 0$ of the server DS_k, the budget can be used to serve first the backlog $L(t)$ and if the backlog reaches $L(t) = 0$ then the budget is used to serve the computation demand $C(t)$ of J. The first time the computation demand reaches 0 is called the finish f_J of J, i.e., we have $C(f_J) = 0$.

If at the release r_J of the job J there is enough budget to complete both the backlog $L(r_J)$ and the computation demand $C(r_J)$ then the job J finishes as soon as additional $L(r_J) + C(r_J)$ budget is consumed.

▶ **Lemma 1.** *If a job J of task τ_k is released at time r_J and the remaining budget of* DS_k *is higher than the execution demand $C(r_J)$ of J at time r_J plus the backlog $L(r_J)$ from previous jobs of τ_k at time r_J, then J finishes within $R^{-\text{DS}}_k(L(r_J) + C_k)$ time units.*

Proof. If the budget is higher than the execution time plus the backlog at time r_J, then J and all previously released unfinished jobs can execute whenever the server DS_k is not interfered. The job J finishes when $L(r_J) + C(r_J)$ amount of computation demand is finished, which is after at most $R^{-\text{DS}}_k(L(r_J) + C_k)$ time units. ◀

In particular whenever there is no backlog and the remaining budget at time r_J is at least C_k time units, then the job J finishes within $R^{-\text{DS}}_{k}(C_k)$ time units.

However, there are cases in which the budget is not sufficient:

▶ **Definition 2** (Exhausted budget). *We say the budget B is exhausted by job J if there exists a point in time t such that the following conditions are met:*

- *There is remaining computation demand $C(t) > 0$ that wants to consume the budget.*
- *The budget $B(t) = 0$ has reached 0.*
- *Instead the processor idles or a lower priority server (or task) is served.*

In such a case the jobs have to wait until the next budget replenishment br to be served. If after the budget replenishment the budget $B(br) = Q_k$ is sufficient to finish the backlog and computation demand, i.e., $Q_k \geq L(br) + C(br)$, then the worst-case response time of J can be described by the following lemma.

▶ **Lemma 3.** *If after a budget replenishment at time br the remaining backlog $L(br)$ and the remaining computation demand $C(br)$ can be fully served, i.e., $Q_k \geq L(br) + C(br)$, then the job has a response time of at most $(br - r_J) + R^{-\text{DS}}_{k}(L(br) + C(br))$ time units.*

Proof. Similar to Lemma 1, at time br the remaining computation demand of $L(br) + C(br)$ has to be finished. Since the budget is high enough, i.e., $Q_k \geq L(br) + C(br)$, this takes at most $R^{-\text{DS}}_{k}(L(br) + C(br))$ time units. The response time of J is the result of the time it takes from the release of J until br plus the time to finish the remaining computation demand, i.e., $(br - r_J) + R^{-\text{DS}}_{k}(L(br) + C(br))$. ◀

4 Worst-Case Response Time Analysis for One Single Task

In this section, we provide a worst-case response time analysis for a sporadic task τ_k served by a deferrable server DS_k, which fulfills its service condition. That is, throughout this section, we implicitly assume that $R^{-\text{DS}}_{k}(Q_k) \leq P_k$. Furthermore, the utilization of task τ_k is assumed to be no more than the utilization of the deferrable server DS_k, i.e., $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$; otherwise, the worst-case response time of task τ_k is by definition unbounded.

We first explain how to convert the analysis in [7] based on real-time calculus to sporadic tasks served by DS_k. Then, we provide our analysis for a scenario that $T_k \geq P_k$ and $C_k \leq Q_k$, and demonstrate the dominance of our analysis over the analysis in [7] when considering sporadic tasks in Section 4.2. As our analysis requires to evaluate $\sup_{0 \leq x < C_k} R^{+\text{DS}}_{k}(x) + R^{-\text{DS}}_{k}(C_k - x)$, we explain how to implement this search in Section 4.3.

4.1 Existing Analysis Converting from Real-Time Calculus

We restate here the analysis from Cuijpers and Bril [7] based on real-time calculus for sporadic tasks. Note that we use a slightly different notation system from that in [7] due to notation discrepancy between real-time calculus and real-time scheduling theory. Suppose that $r_k(t)$ is the accumulated workload in time interval $[0, t)$ for task τ_k.

Let S be some $S > 0$ such that

$$r_k(s + S) - r_k(s) \leq \frac{Q_k}{P_k} \times S, \qquad \forall s \geq 0 \tag{6}$$

Under the assumption that $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$, setting S to $C_k \times \frac{P_k}{Q_k}$ ensures that

$$S = C_k \times \frac{P_k}{Q_k} \leq C_k \times \frac{T_k}{C_k} = T_k \text{ and } r_k(s + S) - r_k(s) \leq C_k = \frac{C_k}{S} \times S = \frac{Q_k}{P_k} \times S,$$

i.e., Eq. (6) holds.

As for the deferrable server DS_k, if it has a full budget at time t, its service provision from time interval t to $t + R^{-\mathrm{DS}}_{k}(h)$ is at least h. This notation is the inverse representation of the accumulative service in real-time calculus under the same assumption. In terms of real-time calculus, DS_k is guaranteed to provide at least h amount of service within an interval length of $R^{-\mathrm{DS}}_{k}(h)$.

Let h be the minimum value $\leq Q_k$ such that

$$\frac{h}{R^{-\mathrm{DS}}_{k}(h)} \geq \frac{r_k(s+S) - r_k(s)}{S}, \qquad \forall s \geq 0. \tag{7}$$

Let H denote $R^{-\mathrm{DS}}_{k}(h)$ for brevity. Cuijpers and Bril [7] showed that the worst-case response time of τ_k is upper bounded by $S + 2H$.

With the above definitions of H and S, we can restate the worst-case response time analysis from Cuijpers and Bril [7] for sporadic tasks.

▶ **Theorem 4.** *Suppose that the deferrable server* DS_k *fulfills its service condition and that* $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$. *Then*

- S *can be set to* $C_k \times \frac{P_k}{Q_k} \leq T_k$
- H *is upper bounded by* $R^{-\mathrm{DS}}_{k}(Q_k) \leq P_k$

The worst-case response time of a sporadic task τ_k *served by* DS_k *is upper bounded by*

$$C_k \times \frac{P_k}{Q_k} + 2R^{-\mathrm{DS}}_{k}(Q_k) \leq C_k \times \frac{P_k}{Q_k} + 2P_k \leq T_k + 2P_k.$$

Proof. It comes directly from Theorem 1 in [7] and the above analysis of S and H. ◀

4.2 Our Analysis for Sporadic Tasks

One practical scenario is that the replenishment period P_k is set to its served task period T_k, and the capacity Q_k is set to C_k, i.e., $T_k = P_k$ and $C_k = Q_k$. Such a configuration might be out of intuition by expecting the period alignment is perfect. Assuming the timing behaviors of servers are the same as tasks, the Liu and Layland bound [19] might be applicable at the first glance. However, the response time of a task may still be interfered by a backlog of the previous unfinished job due to any potential misalignment. Hence, a corresponding timing analysis is still needed.

The analysis in Theorem 4 is applicable for sporadic tasks. However, for the scenario with $T_k \geq P_k$ and $C_k \leq Q_k$, in this section, we show that a tighter analysis can be achieved by examining the interplay between τ_k and DS_k more closely.

▶ **Definition 5** (Consecutive DS service interval)**.** *An interval* $G = [a, b) \subseteq \mathbb{R}$ *is called a* consecutive DS service interval *if it is a minimal interval such that the following properties are met:*

- a *and* b *are time instances where* DS_k *replenishes its budget.*
- *All jobs of* τ_k *that are released during* G *finish their execution during* G.
- *Only jobs of* τ_k *that are released during* G *can be executed.*

A consecutive DS service interval can be constructed in the following way: Let a be a time instant such that 1) DS_k replenishes its budget at a, 2) there is no unfinished job of τ_k at time a, and 3) a job of τ_k is released during $[a, a + P_k)$. Then, we set b to be the time of the next replenishment of DS_k such that there is again not unfinished job of τ_k at time b. The interval $[a, b)$ is a consecutive DS service interval. Please note that every job is inside a consecutive DS service interval as the above procedure to construct consecutive DS service intervals can be repeated for the whole time domain.

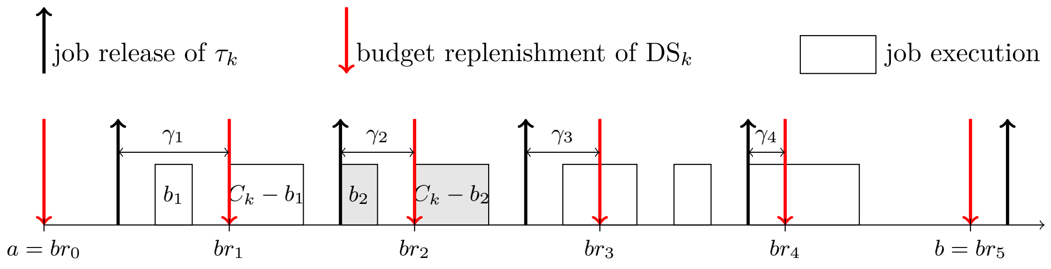


Figure 1 Analysis scenario for Theorem 7. We analyse the second job J_2 (marked in grey) in the consecutive DS service interval $[a, b)$.

The first job in any consecutive DS service interval receives budget Q_k. Under the assumption that $C_k \leq Q_k$ this job finishes after at most $R^{-\mathrm{DS}}_k(C_k)$ time units.

▶ **Lemma 6.** *Under the assumption that $C_k \leq Q_k$ and $T_k \geq P_k$, the following holds:*

1. *Between any two budget replenishments there is at most one job release.*
2. *Every job finishes until the second replenishment period after the job release.*
3. *There is at most one previous unfinished job of τ_k at any job release of τ_k.*
4. *There are at most two jobs of τ_k executed between two consecutive budget replenishments of* DS_k*.*

Proof. **1**: If there would be two job releases between two budget replenishments, then $T_k < P_k$, which contradicts our assumption.
2: By contradiction: Assume that J is the earliest job such that it does not finish until the second replenishment period after its release. Let br_{i-1} and br_i the two consecutive budget replenishments before and after the release of J. By **1**, the previous job J' is released before br_{i-1}. Moreover, since J is the first job such that **2** does not hold, J' finishes until br_i. At time br_i there is no backlog from J' or from earlier jobs. The computation demand at br_i by J is at most $C_k \leq Q_k$. By Lemma 3 the job J finishes until $br_i + R^{-\mathrm{DS}}_k(C_k) \leq br_i + P_k$ by the sufficient service assumption stated in Section 3.1.
3 and **4**: Follow directly from **1** and **2**. ◀

▶ **Theorem 7.** *Suppose that the deferrable server* DS_k *fulfills its service condition and that* $\frac{C_k}{T_k} \leq \frac{Q_k}{P_k}$*. If $C_k \leq Q_k$ and $T_k \geq P_k$, then*

$$R^{\tau}_k \leq \max\left((P_k - T_k) + \sup_{0 \leq x < C_k} (R^{+\mathrm{DS}}_k(x) + R^{-\mathrm{DS}}_k(C_k - x)),\ R^{-\mathrm{DS}}_k(C_k)\right) \tag{8}$$

Proof. We prove this theorem over induction of the jobs in each consecutive DS service interval. Let G be some consecutive DS service interval. We show by induction that for all jobs that are released during G, the upper bound on the worst-case response time given by Eq. (8) holds. We denote by J_i the i-th job that is released by τ_k during G. Moreover, we denote by γ_i the time between the release of J_i and the subsequent budget replenishment br_i. Due to the assumption that $T_k \geq P_k$, we have $\gamma_1 \geq \gamma_2 + (T_k - P_k) \geq \gamma_3 + 2(T_k - P_k) \geq \dots$ as demonstrated in Figure 1. By Lemma 6 there is at least one budget replenishment between any two consecutive job releases. Indeed, there is exactly one budget replenishment between two consecutive job releases since otherwise the earlier job would finish until the second budget replenishment after its release and the consecutive DS service interval would end. Therefore J_i is always released during br_{i-1} and br_i. We define b_i as the time that J_i is executed before br_i and $R^{\tau}_{k,i}$ be the response time of the job J_i. For the intermediate jobs J_i in the consecutive DS service interval we show that the following holds:

	backlog	no backlog
budget not exhausted between release of J_i and br_i	(not possible) **(Case 2)**	$R^{\tau}_{k,i} \leq R^{-\mathrm{DS}}_{k}(C_k)$ $\gamma_i \leq R^{+\mathrm{DS}}_{k}(b_i)$ **(Case 1)**
budget exhausted between release of J_i and br_i	$R^{\tau}_{k,i} \leq (P_k - T_k) + R^{+\mathrm{DS}}_{k}(b_i) + R^{-\mathrm{DS}}_{k}(C_k - b_i)$ $\gamma_i \leq R^{+\mathrm{DS}}_{k}(b_i)$ **(Case 3)**	

Moreover, for the last job J_i in consecutive DS service intervals (if a last job exists) we show that the following holds:

	backlog	no backlog
budget not exhausted between release of J_i and br_i	$R^{\tau}_{k,i} \leq R^{\tau}_{k,i-1}$ **(Case 2)**	$R^{-\mathrm{DS}}_{k}(C_k)$ **(Case 1)**
budget exhausted between release of J_i and br_i	$R^{\tau}_{k,i} \leq (P_k - T_k) + R^{+\mathrm{DS}}_{k}(b_i) + R^{-\mathrm{DS}}_{k}(C_k - b_i)$ $\gamma_i \leq R^{+\mathrm{DS}}_{k}(b_i)$ **(Case 3)**	

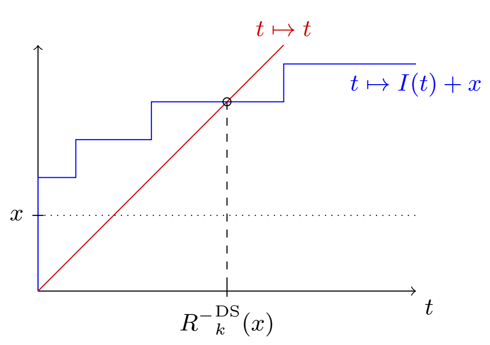
First job J_1. At the release of J_1 there is $Q_k \geq C_k$ amount of budget and no backlog from previous jobs. Therefore, the budget is not exhausted between the release of J_1 and the next budget replenishment. Hence, the first job is always in Case 1. By Lemma 1 the response time $R^{\tau}_{k,1}$ of J_1 is upper bounded by $R^{-\mathrm{DS}}_{k}(C_k)$. If J_1 is an intermediate job, then it does not finish before br_1. In particular an execution demand of b_1 time units could be served between the release of J_1 and br_1, although there was enough budget available. We conclude that the worst-case resumed time $R^{+\mathrm{DS}}_{k}(b_1)$ has to be at least γ_1.

Induction step $J_{i-1} \rightarrow J_i$. Under the assumption that J_{i-1} is an intermediate job, we show that for J_i the bounds presented in the tables still hold.

Case 1: The budget is *not* exhausted between the release of J_i and br_i, and there is *no* backlog from J_{i-1}. This job behaves analogously to the first job J_1. The worst-case response time $R^{\tau}_{k,i}$ is upper bounded by $R^{-\mathrm{DS}}_{k}(C_k)$ due to Lemma 1. If J_i is the last job in G, then this is the only bound from the table to be proven. Otherwise, $\gamma_i \leq R^{+\mathrm{DS}}_{k}(b_i)$ holds since only an execution demand of b_i time units during the interval of length γ_i could be served.

Case 2: The budget is *not* exhausted between the release of J_i and br_i, and there is backlog from J_{i-1}. In this case there is demand at all times during br_{i-1} and the release of J_i. If J_i would not finish before br_i, then the budget would be exhausted no later than at time $br_{i-1} + R^{-\mathrm{DS}}_{k}(Q_k) \leq br_i$. This is not possible by the assumption of this case. Therefore in this case J_i must finish before br_i and is therefore always the last job in the consecutive DS service interval G. Since the job finishes before br_i we have $R^{\tau}_{k,i} \leq \gamma_i \leq \gamma_{i-1}$ which is upper bounded by $R^{\tau}_{k,i-1}$ since J_{i-1} has to finish after br_{i-1} to produce a backlog for J_i.

Case 3: The budget is exhausted between the release of J_i and br_i. For this case, it is irrelevant if J_i is an intermediate or the last job since the same properties have to be proven. Since by Lemma 6 the job finishes until the second budget replenishment after its release, this means at the replenishment time br_i there is no backlog from the previous job J_{i-1}. Moreover, the remaining computation demand from J_i is at most $C_k - b_i \leq Q_k$. By Lemma 3 the response time of J_i is upper bounded by $\gamma_i + R^{-\mathrm{DS}}_{k}(C_k - b_i)$. Moreover, we know that $\gamma_i \leq (P_k - T_k) + \gamma_{i-1} \leq (P_k - T_k) + R^{-\mathrm{DS}}_{k}(b_{i-1})$. Since J_{i-1} has to be served by DS_k for $Q_k - b_i$ time units after br_{i-1} such that J_i exhausts the budget of



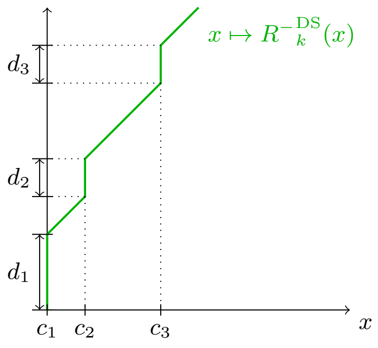


Figure 2 Computation of $R^{-\text{DS}}_{k}(x)$ by finding the intersection of two functions.

Figure 3 Shape of $R^{-\text{DS}}_{k}(x)$.

DS_k before br_i, we have $b_{i-1} \leq C_k - (Q_k - b_i)$, resulting in $b_{i-1} \leq b_i$. We conclude that $\gamma_i \leq (P_k - T_k) + \gamma_{i-1} \leq (P_k - T_k) + R^{-\text{DS}}_{k}(b_{i-1}) \leq (P_k - T_k) + R^{-\text{DS}}_{k}(b_i) \leq R^{-\text{DS}}_{k}(b_i)$ and the worst-case response time is upper bounded by $(P_k - T_k) + R^{+\text{DS}}_{k}(b_i) + R^{-\text{DS}}_{k}(C_k - b_i)$.

Conclusion. By induction we have proven that the bounds from the above stated tables hold. Since $b_i < C_k$ holds for those jobs with exhausted budget, the response time bound in Eq. (8) holds for all jobs by analyzing all consecutive DS service intervals. ◀

▶ **Theorem 8** (Dominance discussion). *The worst-case response time bound presented in Theorem 7 dominates the bound from Theorem 4 when $C_k \leq Q_k$ and $T_k \geq P_k$.*

Proof. As $(P_k - T_k) \leq 0$ by assumption, the worst-case response time bound provided in Theorem 7 is upper bounded by $\sup_{0 \leq x < C_k} R^{+\text{DS}}_{k}(x) + R^{-\text{DS}}_{k}(C_k)$. Since $R^{+\text{DS}}_{k}(x) \leq R^{-\text{DS}}_{k}(C_k)$ for all $x < C_k$, the bound from Theorem 7 is also upper bounded by $2R^{-\text{DS}}_{k}(C_k) \leq C_k \cdot \frac{P_k}{Q_k} + 2R^{-\text{DS}}_{k}(Q_k)$ which is the bound from Theorem 4. ◀

4.3 Efficient Computation of Worst-Case Response Time Bound

For the computation of the worst-case response time upper bound presented in Theorem 7 the supremum

$$\sup_{0 \leq x < C_k} R^{+\text{DS}}_{k}(x) + R^{-\text{DS}}_{k}(C_k - x) \tag{9}$$

has to be computed. In this section we discuss a method to do this efficiently without computing the values for $R^{+\text{DS}}_{k}$ and $R^{-\text{DS}}_{k}$ at every point using fixed-point iterations. As presented in Section 3.1, $I(t)$ is of the form $\sum_i \left\lceil \frac{t+j_i}{P_i} \right\rceil \rho_i$ for some positive real values j_i, P_i and ρ_i. Fixed-point iterations can be used to compute the value of $R^{-\text{DS}}_{k}(x)$, in particular the intersection between the functions $t \mapsto t$ and $t \mapsto I(t) + x$ is computed, as presented in Figure 2.

The efficient presentation and formulation presented in this section is based on the observation that $R^{-\text{DS}}_{k}(x)$ and $R^{+\text{DS}}_{k}(x)$ coincide and grow linearly if the intersection with $I + x$ is on a plateau, i.e., if I is constant during the interval $(R^{-\text{DS}}_{k}(x) - \delta, R^{-\text{DS}}_{k}(x) + \delta)$ then $R^{-\text{DS}}_{k}(y) = R^{-\text{DS}}_{k}(x) + (y - x)$ for all $y \in (x - \delta, x + \delta)$. At those points x where there is a jump of I at $R^{-\text{DS}}_{k}(x)$, the values of $R^{+\text{DS}}_{k}(x)$ and $R^{-\text{DS}}_{k}(x)$ are computed using fixed-point

Algorithm 1 Computation of all values in $\mathcal{S}$ with $c_i \leq C_k$.

```
    Input: $I(t) = \sum_i \left\lceil \frac{t+j_i}{P_i} \right\rceil \cdot \rho_i$
    Output: The set $\mathcal{S}$ with all values $(c_i, d_i)$ where $c_i \leq C_k$.

1: $\mathcal{S} := [\,]$; $x := 0$
2: while $x \leq C_k$ do
3:     Compute $R^{-DS}_k(x)$ and $R^{+DS}_k(x)$ by fixed-point iterations.
4:     $c := x$; $d := R^{+DS}_k(x) - R^{-DS}_k(x)$
5:     Add $(c, d)$ to the set $\mathcal{S}$
6:     $x := x + \min_i (-(x + j_i) \mod P_i)$          ▷ Time until next jump.
7: return $\mathcal{S}$
```

iterations. The shape of R^{-DS}_k is presented in Figure 3. Please note that the shape of R^{-DS}_k and R^{+DS}_k coincide during the linear parts and only at the jumps $(c_1, c_2, \dots)$ the function R^{-DS}_k takes the lower value and R^{+DS}_k takes the higher value.

Based on the shape of the functions R^{-DS}_k and R^{+DS}_k, there exists a set of tuples $\mathcal{S} = \{(c_1, d_1), (c_2, d_2), (c_3, d_3), \dots\}$ such that

$$R^{-DS}_k(x) = x + \sum_{(c_i,d_i)\in\mathcal{S}} \chi_{x>c_i} \cdot d_i \qquad \text{and} \qquad R^{+DS}_k(x) = x + \sum_{(c_i,d_i)\in\mathcal{S}} \chi_{x\geq c_i} \cdot d_i \tag{10}$$

where $\chi_{x>c_i}$ is 1 if $x > c_i$ and 0 else, and $\chi_{x\geq c_i}$ is 1 if $x \geq c_i$ and 0 else. The computation of the values of $\mathcal{S}$ is presented in Algorithm 1. In each step of the while-loop the size of a jump is computed by the difference between R^{-DS}_k and R^{+DS}_k, and the corresponding tuple is added to the set $\mathcal{S}$. Afterwards, the time until the next jump is computed by finding the next point in time where the intersection t with $x + I(t)$ reaches a jump, i.e., after $\min_i (-(x + j_i) \mod P_i)$ additional time units.

With the representation of R^{-DS}_k and R^{+DS}_k achieved in Eq. (10), the supremum in Eq. (9) can be rewritten as

$$\sup_{0\leq x<C_k} R^{+DS}_k(x) + R^{-DS}_k(C_k - x) \tag{11}$$

$$= x + \sum_{(c_i,d_i)\in\mathcal{S}} \chi_{x\geq c_i} \cdot d_i + \sup_{0\leq x<C_k} x + \sum_{(c_i,d_i)\in\mathcal{S}} \chi_{C_k-x>c_i} \cdot d_i \tag{12}$$

$$= C_k + \sum_{(c_i,d_i)\in\mathcal{S}} (\chi_{x\geq c_i} + \chi_{x<C_k-c_i}) \cdot d_i \tag{13}$$

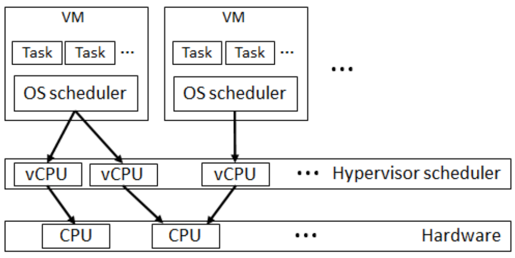
In particular, only finitely many cases have to be checked to find the exact solution of the supremum formulated in Eq. (9).

5 Architecture Model

In this section, we present our unikernel-based architecture for hosting virtualized CPS applications. First, we introduce the concept of unikernel-based CPS applications. Next, we shorty describe the Xen architecture as well as its implications for scheduling and compare it with our approach. Finally, we provide some design and implementation details.

5.1 Unikernel-based CPS Applications

In data-centers and cloud environments, each VM is expected to host a general purpose operating system to ease the effort of porting legacy software with a lot of inherent libraries and functionalities. However, most CPS applications are functionally dedicated, single-purposed and thus not dependent on additional functionality. Due to the encapsulation of



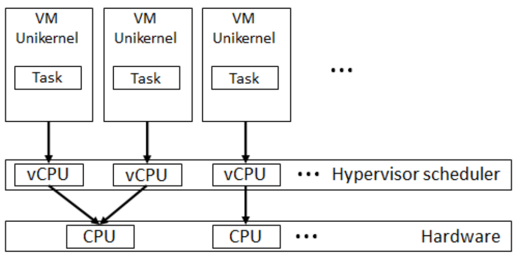


Figure 4 depicts the typical scheduling architecture in platform virtualization: The scheduling decision is split into two layers, the OS scheduler of the guest and the hypervisor scheduler.

Figure 5 illustrates our approach where scheduling decisions are reduced to one layer only, due to the adaptation of unikernels.

superfluous libraries and the execution of non-essential processes that are not related to the task of interest, deploying real-time tasks of CPS on a general-purpose OS in fact arises various issues like resource efficiency and timing predictability.

Towards this, several efforts have been made in recent years for the realization of single-purpose appliances [21, 22, 24, 26], so called *unikernels*. These are sealed, single-purpose VM images that can be constructed using the concept of library operating systems (LibOS) [12, 13]. LibOS's allow for the tailoring of an OS code base to the particular needs of a given task, by which only those parts of the OS API are included into the VM image.

This approach has several advantages. Unikernels are characterized by a minimal VM image size which highly increases their security properties, due to the minimal attack surface for malicious code injections This also translates to a substantial reduction of overall system resource usage. Moreover, unikernels can be instantiated and become fully functional within only a few milliseconds. It has further been shown that unikernels are more efficient and safer than modern container technologies [23]. Please note that while unikernels have indeed a minimal attack surface in comparison with full-fledged operating systems and are fully isolated from other guests, depending on the given scenario, some additional security defence mechanisms would have to be adopted, but considered out of scope in this work.

Considering these benefits of unikernels, as well as the fact that most CPS applications are specialized and functionally dedicated tasks and can be implemented as single-purpose appliances, CPS applications provide an excellent target for unikernels.

5.2 Scheduling Architecture

The case-study results presented in this paper are linked to the Xen hypervisor [2], which we have chosen to realize our unikernel-based approach for hosting virtualized CPS applications. Xen is a type 1 hypervisor and allows for the consolidation of multiple systems on a single platform. Xen runs directly on host's hardware and is the first software layer to execute after the bootloader. The hypervisor is responsible for managing hardware resources, including CPUs and memory. It also handles timers and the scheduling of VMs. Specific to Xen is a privileged VM called Domain 0. It is the first VM to load under Xen, it holds the drivers to the underlying hardware and this is also where the toolstack resides that enables management of further VMs. Typically Domain 0 is deployed on Linux. In order to make use of the existing drivers in Domain 0, Xen uses an approach called paravirtualization, which exposes an API to the guest VM for delegating privileged instructions, including driver calls. This approach is much more efficient than emulation. The backbone of the paravirtualization driver concept under Xen is the split device driver model. The drivers consist of two parts: the front-end and the back-end. The front-end is situated in the guest VM while the back-end resides in Domain 0. Both parts are isolated and communicate through shared memory.

As shown in Figure 4, the scheduling decision in Xen is split into two tiers. The bottom tier is constituted by the hypervisor scheduler which assigns virtual CPUs (vCPU) to physical CPUs (pCPU). The second tier consists of the guest operating system schedulers within virtual machines, which in turn assign their threads to vCPUs. This split is needed for two reasons. The first being the possibility to abstract physical resources (e.g. pCPUs) into logical resources (e.g. vCPUs) which is a premise for achieving better hardware utilization. Secondly, it allows the hypervisor to enforce timing isolation between the concurrently running VMs. For systems that comprise of task sets without strict timing requirements and that implement a fair share scheduler, this architecture allows for a high resource utilization while at the same time preventing any faulty or compromised VM from hijacking system resources, and by this from negatively influencing the behavior of other VMs in the system. However, when it comes to the need of providing timing guarantees, this scheduling architecture also complicates the corresponding schedulability analysis. As discussed in Section 3.1, the suspension behavior of deferrable servers additionally interferes with the response time of lower priority servers or tasks in the worst case. While improving the applicability of deferrable servers in Section 4.3, our approach, i.e., one sporadic task per server, aligns well with the concept of unikernels, by which CPS applications are deployed as single tasked VMs. As shown in Figure 5, this renders the scheduler instances inside the VMs obsolete. Due to the fact that each of our unikernels hosts a single task, there is no point of assigning more than one vCPU per unikernel.

5.2.1 Real-Time Networking

As in practice, CPS applications commonly assume distributed architectures, the I/O processing of network packets is a matter of particular importance. Xen handles packet processing in Domain 0 where the network driver resides. Each instantiated VM under Xen is connected to a dedicated virtual network interface (VIF) and a corresponding dedicated VIF-thread. This is the context where the actual packet processing takes place. Unfortunately, by default, the VIF-threads in Xen are scheduled independently of the priority of their VMs. This can lead to priority violation, i.e., the order of packet processing mismatches the priorities of vCPUs. In order to solve this issue, we align the priorities of the packet processing threads with the corresponding priorities of the vCPUs in the hypervisor scheduler.

5.3 Design Principles

In our model, each vCPU is implemented as a deferrable server and is described by its *capacity* and *replenishment period*. Due to our adaptation of unikernels, each vCPU has only one task assigned to it. The vCPU is released under the sporadic task activation. The budget denotes the amount of time a vCPU can consume for its execution during the replenishment period. The budget of the vCPU is set to the given capacity at its replenishment periodically. The vCPU can either be runnable or blocked. While running, the vCPU consumes its budget. A vCPU with a depleted budget will not be scheduled. If there are no eligible (runnable and with budget) vCPUs, the hypervisor will schedule an idle vCPU. As vCPUs are implemented as deferrable servers, they can defer their budget to be used at a later time. However, the budget cannot be preserved and transferred into the next period. Budgets that were not consumed during their current periods are lost.

Our implementation relies on partitioned queues, i.e., each pCPU possesses and manages its own run queue of vCPUs. That is, the scheduling model is under (multiprocessor) partitioned scheduling paradigm. The priorities are statically assigned to the vCPUs according to their replenishment periods, following the *rate-monotonic* (RM) policy. As mentioned in Section 3.1, the utilization bound $\ln(3/2) \approx 0.40546$ guarantees the sufficient service condition of the deferrable servers on one physical processor, whereas a tighter analysis can be achieved by adopting Eq. (1). Each time a vCPU is assigned to a run queue, the vCPU is inserted accordingly to its priority. In the case of the RM algorithm, the highest priority is given to the VM with the shortest replenishment period. In this process also the priorities of all lower prioritized vCPUs are updated. Scenarios where a vCPU is assigned to a run queue include the instantiation of a new VM or an existing VM becoming runnable. Blocked vCPUs are removed from the run queue.

5.4 Implementation on Xen

The Xen hypervisor provides an interface to schedulers by exposing an abstract scheduler struct which contains pointers to functions which have to be implemented when adding a new scheduler to Xen. The scheduler policy independent code is situated in the scheduler.c file. All of the functions defined in the interface have analogs in this file. It comprises scheduler policy independent code which after execution calls the specialized functions from a specific scheduler implementation. The most important function in schedule.c is *schedule()*. As the name suggests, this function is executed when a scheduling decision is needed. In order to choose the next vCPU to run, it deschedules the currently running vCPU and calls the specialized *do_schedule()* function from the custom scheduler file. The default scheduling policy for Xen is implemented in the *Credit Scheduler* which is a fair-share scheduler and therefore not suitable for scheduling tasks with real-time constraints. We have extended Xen with our own scheduler that implements our deferrable server model. In the following, we shorty describe some of its implementation details.

The *do_schedule()* function is critical to the performance of the scheduler, as it is invoked very often. Therefore, we have designed it to be fast and simple. In order to achieve this, we keep our run queue sorted. The sorting process is conducted while inserting or reinserting vCPUs after instantiation or unblocking. As the run queue is already sorted with respect to vCPUs priorities, our *do_schedule()* function has to conduct only two operations. First, it updates the bookkeeping (consumed budget) for the currently running vCPU. Secondly, it chooses the next vCPU to run from the top of the run queue. In the case that there are no eligible vCPUs to run, the algorithm returns the idle vCPU. In our implementation, the scheduling quantum is set to 100 µs, at which the server budget is updated. The pseudocode for our function is depicted in Algorithm 2.

Another aspect worth mentioning is the implementation of the budget replenishment. One way to implement budget replenishment is inside the *do_schedule()* function. However, this would break the efficiency of this function, due to the unnecessarily high amount of budget validations. In most cases, the *do_schedule()* will be invoked more often than budget replenishment is necessary. Therefore, we transferred this functionality to timers. For each of the vCPUs a timer is instantiated with a period that equals the replenishment period of the task. The replenishment itself is happening inside the timer handler. We have extended our toolstack for the scheduler interface in the management domain with the possibility to migrate replenishment timers to other pCPUs in the system. This allows for testing and fine-tuning of the impact of timer interrupts on scheduling.

Algorithm 2 Pseudocode of the do_schedule() function from our deferrable server scheduler.

```
1: next = null
2: consumeBudget(getCurrentVCPU)
3: for each vCPU in RunQ do
4:     if vCPU.isRunnable then
5:         if vCPU.hasBudget then
6:             next ← vCPU
7:             break
8: if vCPU == null then
9:     next ← idleVCPU
10: return next
```

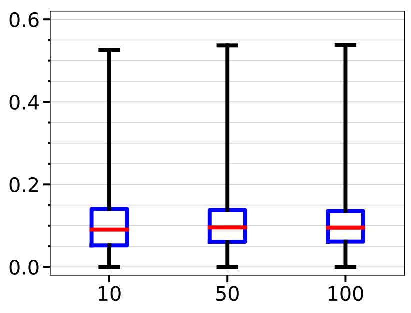


Figure 6 $\frac{WCRT_{Our}}{WCRT_{SOTA}}$ for different number n of servers per test.

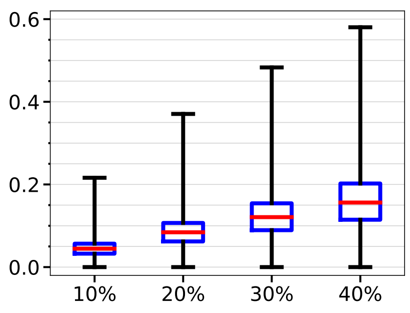


Figure 7 $\frac{WCRT_{Our}}{WCRT_{SOTA}}$ for different total utilization U on one processor.

6 Evaluation and Discussion

Firstly we evaluate the tightness of our response time analysis in comparison to the state of the art [7]. Then, we conduct an empirical case study on a previously synthesized task set. With this case study we aim at showing that our approach of unikernel-based virtualization under deferrable servers is feasible in practice.

6.1 Numerical Simulation

We conducted numerical evaluation to compare our worst-case response time bound (Theorem 7) with the upper bound derived from [7] (Theorem 4). We present two experiments: 1) we distinguish different number of servers, and 2) we distinguish different total utilization of the servers, i.e., $U = \sum_{\mathrm{DS}_i} \frac{Q_i}{P_i}$. To synthesize a system with n servers with a total utilization of U we applied 4 steps:

- We generated n utilization values $U_i \in (0, 1)$ that add up to the total utilization U by applying the UUniFast method presented in [4].
- We generated n replenishment periods P_i by drawing them log-uniformly from the interval $[1, 100]$[ms] as it is suggested in [11] for the generation of task sets.
- We generated n servers $\mathrm{DS}_1, \ldots, \mathrm{DS}_n$ by specifying the replenishment periods P_i from above and by setting the capacity Q_i to $P_i \cdot U_i$. Priorities are set in rate-monotonic order.
- We generated the tasks τ_i by drawing the minimum inter-arrival time T_i uniformly at random from the interval $[1.0P_i, 1.5P_i]$ and by drawing the worst-case execution time C_i uniformly at random from the interval $[0.5Q_i, 1.0Q_i]$.

In the first experiment we generated 1000 sets of servers and their tasks at random using the procedure specified at the beginning of this section for each given number of servers in $\{10, 50, 100\}$. For each system in the first experiment, the utilization was drawn uniformly from the interval $[0.1, 0.4]$ since it has been shown in [5] or Theorem 8 in [20] that $U \leq \ln \frac{3}{2} \approx 0.40546$ guarantees that the service condition stated in Eq. (1) holds.

In the second experiment we generated 1000 sets of servers and their tasks at random using the same procedure specified previously for each given total utilization in $\{0.1, 0.2, 0.3, 0.4\}$. For each system in the second experiment, the number of servers n was drawn uniformly at random from the interval $[10, 100]$.

We applied our worst-case response time analysis ($WCRT_{Our}$) and the converted analysis, i.e., Theorem 4, derived from the real-time calculus based worst-case response time analysis in [7] ($WCRT_{SOTA}$). In Figures 6 and 7 we present the values of $\frac{WCRT_{Our}}{WCRT_{SOTA}}$ for all tasks in the tests as boxplots. In particular the lower the value, the better the performance of our analysis is. The medians are depicted by a horizontal line, the boxes mark a quartile of the data points, and the whiskers present all values.

We observe in Figure 6 that our bound is in median 90.5% smaller than the worst-case response time bound from Theorem 4. Although the number of servers per test does not make a difference in the performance of our method compared to the state of the art, in Figure 7 we can see that the utilization has an impact. The higher the utilization, the closer our bound gets to the bound of the state of the art. However, even with 40% utilization, our bound is in median still 84.4% smaller than the bound derived by the state of the art.

Please note that we limited our experiments for $U \leq \ln \frac{3}{2} \approx 0.40546$ so that the service condition of a deferrable server is always fulfilled. Otherwise, the service condition of a deferrable server cannot be always guaranteed and the focus of the evaluation would be drifted to the service condition tests of deferrable servers. However, our analysis in Theorems 4 and 7 is applicable as long as the deferrable server can fulfill its service condition.

6.2 Case Study

In this subsection, we describe our practical case study. We provide the experiment setup, the measurement methodology and present the results. Similar to the numerical simulation, we synthesize four periodic tasks with a total utilization 40%, and specify the capacity and replenishment period for each deferrable server. Their parameters are shown in Table 2. Note that we stick to periodic tasks to focus on the incurring overhead. They are deployed in form of four unikernels, which are instantiated on the same single processor core. Domain 0 receives one separate core.

The experiments were conducted on a commercial off-the-shelf hardware, a barebone Intel NUC Kit with an Intel i5-8259U 4-core processor running at a constant speed of 2.3 GHz. Turbo Mode, hyper-threading as well as all power management features were disabled. Domain 0 ran on a 64-Bit version of Ubuntu 20.04.1 LTS Server with a para-virtualized Linux kernel 5.4.0-59-generic. The used hypervisor was Xen in version 4.14.1 extended with our deferrable server scheduler and our toolstack.

6.2.1 Benchmark

For the purpose of this case study a User Datagram Protocol (UDP)-based client-server benchmark has been implemented in the programming language C. The client-server model fits the distributed nature of CPS's. The benchmark servers represent our CPS tasks (e.g. protection or control algorithms) deployed as unikernels. The computational workloads of

Table 2 Four periodic tasks and their corresponding deferrable servers for our case study. Note that the time unit is [ms] and the WCRT of each task is derived from our analysis.

Periodic Task	Deferrable Server	WCRT
$T_1 = 12,\ C_1 = 1$	$P_1 = 10,\ Q_1 = 2$	1
$T_2 = 20,\ C_2 = 4$	$P_2 = 20,\ Q_2 = 4$	12
$T_3 = 60,\ C_3 = 8$	$P_3 = 50,\ Q_3 = 10$	26
$T_4 = 130,\ C_4 = 9$	$P_4 = 100,\ Q_4 = 10$	79

the tasks have been configured to fit the parameters of the analyzed task set as Table 2. The benchmark clients are used for triggering the computation inside the unikernels by generating requests (network packets). The requests can be interpreted as, for example, sensor values for a given task. After completing its computation, each task sends a response packet to the corresponding client. The responses can be seen as control commands destined for an actuator in a given CPS. The architecture of our benchmark is depicted in Figure 8.

During benchmarking we collect three different types of latencies for every single request/response pair. The most important one is the task response time – a) in Figure 8. We define it as the time interval between the moment when a network packet (request) destined for a given unikernel (task) arrives at the bottom of the Linux TCP/IP network stack in Domain 0 and the time-stamp at which a corresponding departing packet is delegated to the network adapter for a response transmission. In other words, the packet arrival time-stamp and the corresponding packet departure time-stamp correlate with the task release and task finish times from our schedulability analysis. In order to collect these values, we hook into the TCP/IP stack layer-2 kernel functions using systemtap[4] [27] and log the appropriate time-stamps. Another latency type is the task execution time – b) in Figure 8. It allows for quantifying the amount of processor cycles a task needed for completing is workload from the moment it has been scheduled to the point of time where it finishes its computation. We use it to estimate or verify the actual computational workload of our task on the given hardware. To this end, we implemented a clock cycle precise measurement technique as in [25]. This type is not suited for measuring the task response time, as it does not account for the time a given task may remain blocked after its release, e.g. due to its lower priority. On the client, we can also measure the round-trip times for the request/response pairs – c) in Figure 8. We use them to check the plausibility of the other latency types.

Last but not least, before we can present and discuss the results, we have to address the aspect of latencies related to networking. The schedulability analysis from Section 4 currently concentrates on local scheduling and does not account for latencies imposed by networking. However, the response times measured by the means of our benchmark include delays related the entire software network stack of the host. Therefore, for the purpose of the following case study, we empirically estimated the latencies related to the network stack and adjusted our computed worst-case response time bounds with the worst-case network processing time. In the course of our experiments, the worst-case latency for the network stack did not exceed 250 µs and amounted on average to 157 µs.

[4] `https://sourceware.org/systemtap`

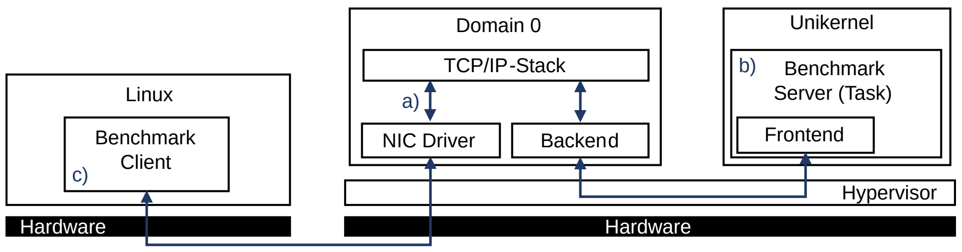


Figure 8 Benchmark architecture and the latencies measurement locations.

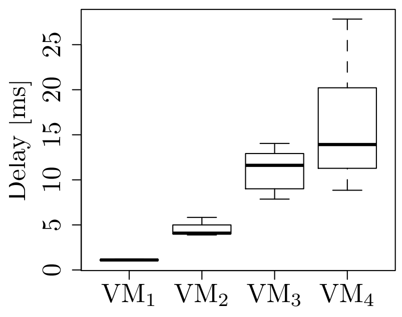


Figure 9 Boxplots of the response time values for the different VMs.

Table 3 The minimum value, arithmetic mean, standard deviation, worst-case response time and the worst-case response time bound for the different VMs.

VM	*Min.*	$\overline{x}$	σ	*WCRT*	*WCRT(B)*
VM_1	1 ms	1.11 ms	0.047 ms	1.215 ms	1.25 ms
VM_2	3.88 ms	4.42 ms	0.49 ms	5.83 ms	12.25 ms
VM_3	7.86 ms	10.99 ms	2.02 ms	14.03 ms	26.25 ms
VM_4	8.84 ms	15.54 ms	5.14 ms	27.83 ms	79.25 ms

6.2.2 Case Study Results

In the following, we present and discuss the results of our case study. In order to estimate the response times of the evaluated VMs, a total of twenty-two thousand measurements have been conducted. The results are depicted in form of four boxplots in Figure 9 and the corresponding numerical values are presented in Table 3. Please note that the estimated worst case delay, i.e., 250 µs, as discussed above is added into each WCRT(B) of VMs.

As can be seen in Figure 9, our deferrable server services the four VMs accordingly to their priorities. Further, there are no outliers. This means that during the entire experiment none of the VMs experienced any spikes in their latencies. This in turn translates to a deterministic execution behavior. The standard deviation values from Table 3 support this evidence. In the case of the highest prioritized VM_1, σ amounts to 47 µs – and this includes the variances caused by the network stack. For comparison, the σ computed from the task execution times collected inside VM_1 (see Figure 8, collection point b) amounts to 12 µs.

Next, in the case study none of the empirically measured WCRTs exceeds the computed WCRTs bounds, i.e., $WCRT(B)$, from our analysis. This information can be gathered by comparing the WCRTs in Table 3. The comparison also shows that for decreasing VM priorities the computed bounds become more and more conservative.

Our case study leads to the conclusion that our approach is not only feasible but can be efficiently and safely realized in practice even on common-off-the-shelf hardware.

7 Related Work

Various virtualization techniques have been developed as a promising approach for embedded systems and latency-sensitive cloud computing [14]. The scheduling of VMs often implies a hierarchy of schedulers [9]. For hierarchical scheduling, server-based approaches are often considered to represent each virtual machine with its own real-time tasks as a single entity, which is scheduled by the hypervisor. Under fixed-priority scheduling, there are several server-based approaches, e.g., periodic server [29], sporadic server [32], deferrable servers [33]. The concept of deferrable servers was first introduced in [33]. It was originally designated to handle aperiodic activation in hard real-time systems. Afterwards, it has been utilized for resource budgets in [8, 28], where most of timing analyses assume each server may serve multiple tasks. As shown in [8], the lower priority tasks served by deferrable servers, however, may suffer from the back-to-back interference of higher priority tasks, resulting in a lower schedulability than the other periodic server approaches even under an exact schedulability test. Cuijpers and Bril in [7] showed that the response timing analysis proposed in [8] is no longer exact, in the absence of a low-priority task. When a deferrable server only serves a single task, the applicability of deferrable servers could be better. The result was based on real-time calculus [34], which considers a general model of task activation. In this work, we focus on sporadic tasks to derive a tighter worst-case response time analysis.

For using server-based approaches in virtualization, Shin and Lee in [31] developed the compositional scheduling framework (CSF) to compute the capacity of each server, given the replenishment period and the properties of tasks and their scheduler. RT-Xen was introduced in [35], adopting server-based approaches to schedule VMs. The CSF was introduced to RT-Xen for assuring the schedulability of tasks [17]. Both global and partitioned EDF schedulers were implemented for supporting multicore scheduling at the host-level [36]. In this work, we also adopt the deferrable server. However, as RT-Xen was implemented to comply with the compositional scheduling theory and addresses an architecture where scheduling decisions are divided between the hypervisor and the guest OS level, it is not suitable to realize our unikernel-based scheduling model. Therefore, we have extended the Xen hypervisor with our own scheduling infrastructure that implements our deferrable server model.

Lackorzyński et al. were the first to introduce the concept of flattening the hierarchical scheduling [16], where the task information is exported to the hypervisor scheduler. Drescher et al. further proposed to abandon the usage of server-based approaches to achieve ExVM [10]. However, as also noted in [10, 18], such flattening mechanisms might either break the temporal isolation between VMs, or expose task-specific information from a VM, which might not be appealing to the purpose of using virtualization for CPS's. Similarly, RTVirt [37] proposed to enable a cross-layer communication over the schedulers in two different tiers. Although this mechanism can adapt the scheduling decisions according to the dynamic changes, the potential concern is similar to the aforementioned flattening mechanisms. In this work, the main insight is to constrain the number of served real-time tasks of each deferrable server to a single one, without violating any important properties.

8 Conclusions

In this work, we proposed to leverage the scheduling architecture of unikernel-based virtualization to facilitate the schedulability of deferrable servers, in which each server only serves one sporadic task. We presented how to derive the worst-case response time analysis under practical scenarios. The evaluation results show that our analysis outperforms the restated analysis based on the state-of-the-art [7]. In addition, we demonstrated that the unikernel-based architecture can be effectively implemented on top of the Xen hypervisor [2] and conducted a case study to evaluate the applicability of the proposed approach.

In future work we plan to investigate more scenarios over different relationships between a sporadic task and its deferrable server, e.g., scenarios with $T_k \leq P_k$. Further exploration may unleash the full power of the deferrable servers and eventually benefit more CPS applications under real-time virtualization.

References

1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020. doi:10.1109/RTSS49844.2020.00012.

2 Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. doi:10.1145/1165389.945462.

3 Sanjoy Baruah and Nathan Fisher. Component-based design in multiprocessor real-time systems. In *2009 International Conference on Embedded Software and Systems*, pages 209–214, 2009. doi:10.1109/ICESS.2009.71.

4 Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005. doi:10.1007/s11241-005-0507-9.

5 Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Real-Time Systems Symposium (RTSS)*, pages 107–118, 2015. doi:10.1109/RTSS.2015.18.

6 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn B. Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil C. Audsley, Raj Rajkumar, Dionisio de Niz, and Georg von der Brüggen. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real Time Syst.*, 55(1):144–207, 2019. doi:10.1007/s11241-018-9316-9.

7 Pieter J. L. Cuijpers and Reinder J. Bril. Towards budgeting in real-time calculus: Deferrable servers. In *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'07, pages 98–113, Berlin, Heidelberg, 2007. Springer-Verlag.

8 Robert I. Davis and Alan Burns. Hierarchical fixed priority pre-emptive scheduling. In *RTSS*, pages 389–398, 2005. doi:10.1109/RTSS.2005.25.

9 Z. Deng and J.W.-S. Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium (RTSS)*, pages 308–319, 1997. doi:10.1109/REAL.1997.641292.

10 Michael Drescher, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. A flattened hierarchical scheduler for real-time virtualization. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2968478.2968501.

11 Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS*, pages 6–11, 2010.

12 D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, 1995. doi:10.1145/224056.224076.

13 Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for kernel and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 38–51, 1997.

14 Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726–740, 2014. doi:10.1016/j.sysarc.2014.07.004.

15 Boguslaw Jablkowski and Olaf Spinczyk. Cps-xen: A virtual execution environment for cyber-physical applications. In Luís Miguel Pinho, Wolfgang Karl, Albert Cohen, and Uwe Brinkschulte, editors, *Architecture of Computing Systems - ARCS 2015 - 28th International Conference, Porto, Portugal, March 24-27, 2015, Proceedings*, volume 9017 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2015. doi:10.1007/978-3-319-16086-3_9.

16 Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 93–102, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2380356.2380376.

17 Jaewoo Lee, Sisu Xi, Sanjian Chen, Linh T. X. Phan, Chris Gill, Insup Lee, Chenyang Lu, and Oleg Sokolsky. Realizing compositional scheduling through virtualization. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 13–22, USA, 2012. IEEE Computer Society. doi:10.1109/RTAS.2012.20.

18 Haoran Li, Meng Xu, Chong Li, Chenyang Lu, Christopher Gill, Linh Phan, Insup Lee, and Oleg Sokolsky. Multi-mode virtualization for soft real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 117–128, 2018. doi:10.1109/RTAS.2018.00022.

19 C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

20 Cong Liu and Jian-Jia Chen. Bursty-interference analysis techniques for analyzing complex real-time task models. In *2014 IEEE Real-Time Systems Symposium*, pages 173–183, 2014. doi:10.1109/RTSS.2014.10.

21 Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 559–573, USA, 2015. USENIX Association.

22 Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGARCH Comput. Archit. News*, 41(1):461–472, March 2013. doi:10.1145/2490301.2451167.

23 Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132763.

24 Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation*, pages 459–473, 2014.

25 Gabriele Paoloni. How to benchmark code execution times on intel® ia-32 and ia-64 instruction set architectures, 2010.

26 Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304. ACM, 2011. doi:10.1145/1950365.1950399.

27 Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and J Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64, 2005.

28 Saowanee Saewong, Ragunathan Rajkumar, John P. Lehoczky, and Mark H. Klein. Analysis of hierarchical fixed-priority scheduling. In *ECRTS*, pages 173–181, 2002. URL: http://csdl.computer.org/comp/proceedings/ecrts/2002/1665/00/16650173abs.htm.

29 Lui Sha, John P. Lehoczky, and Ragunathan Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 181–191, 1986.

30 Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 181–190, 2008. doi:10.1109/ECRTS.2008.28.

31 Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*, pages 57–67, 2004. doi:10.1109/REAL.2004.15.

32 Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

33 J.K. Strosnider, J.P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995. doi:10.1109/12.368008.

34 Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *IEEE International Symposium on Circuits and Systems, Emerging Technologies for the 21st Century*, pages 101–104, 2000. doi:10.1109/ISCAS.2000.858698.

35 Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 39–48, 2011.

36 Sisu Xi, Meng Xu, Chenyang Lu, Linh T. X. Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in xen. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2656045.2656066.

37 Ming Zhao and Jorge Cabrera. Rtvirt: Enabling time-sensitive computing on virtualized systems through cross-layer cpu scheduling. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3190508.3190527.

A Appendix

▶ **Lemma 9.** *Under global preemptive fixed-priority scheduling of systems composed of only fixed-priority deferrable servers on m homogeneous (identical) physical processors, a sufficient service condition test can be written as Eq.* (2)*.*

Proof. This can be proved by contrapositive. Suppose that the service condition to provide capacity Q_k for DS_k cannot be fulfilled during a period from a to $a + P_k$ for contrapositive. Under multiprocessor global preemptive fixed-priority scheduling, it implies that whenever DS_k is not served, the m processors are busy serving other higher-priority workload (i.e., servers in this case). The amount of execution time a higher-priority DS_i can be executed from a to $a + t$ is upper-bounded by $\left\lceil \frac{t+P_i-Q_i}{P_i} \right\rceil Q_i$, by considering the back-to-back hitting. Therefore, we know that $\exists\, \text{DS}_k, \forall 0 < t \leq P_k, Q_k + \frac{\sum_{\text{DS}_i \in hp_k} \left\lceil \frac{t+P_i-Q_i}{P_i} \right\rceil Q_i}{m} > t$. By taking the negation of the above condition, we reach the conclusion. ◀