Hierarchical scheduling of real-time tasks over Linux-based virtual machines

Hierarchical Scheduling of Real-Time Tasks over Linux-based Virtual Machines

Luca Abeni[a], Alessandro Biondi[a], Enrico Bini[b]

[a] *Scuola Superiore S. Anna, Pisa, Italy*
[b] *Università di Torino, Torino, Italy*

Abstract

Virtualization has made feasible the full isolation of virtual machines (VMs) among each other. When applications running within VMs have real-time constraints, threads implementing the virtual cores must be scheduled in a predictable manner over the physical cores. In this paper, we propose a possible implementation of such a predictable VM scheduling based on Linux and kvm (a hosted hypervisor). The proposed implementation is based on vanilla Linux kernels and standard qemu/kvm, and does not require to apply any patch or to use custom software. We also show that previous work makes some assumptions that are unrealistic in practical situations. Motivated by these considerations, we finally propose a principled methodology to practically implement hierarchical scheduling with Linux. Finally, an extensive set of experiments based on Linux and kvm illustrates how the VMs and host scheduler can be set-up to match theoretical results with experiments.

Keywords: real-time; virtual machines; hierarchical scheduling; Linux; kvm

1. Introduction

In the last decade, there has been an increasing interest in Operating Systems (OSs) that are capable of scheduling groups of tasks by means of *hierarchical scheduling*. Under two-level hierarchical scheduling, a scheduler at

Email addresses: luca.abeni@santannapisa.it (Luca Abeni), alessandro.biondi@santannapisa.it (Alessandro Biondi), bini@di.unito.it (Enrico Bini)

the lower level (called *host* or *root* scheduler) selects which group of tasks to execute, while another scheduler at the higher level (called *guest* or *local* scheduler) selects the tasks within the group. Nowadays, some popular OSs provide out-of-the-box features to support such a scheduling scheme: for instance, the Linux kernel provides *control groups* (also known as the `cgroups` mechanism, which is originally inspired by resource containers [1]) that allow implementing hierarchical scheduling for processes and threads under the `SCHED_OTHER` and `SCHED_FIFO`/`SCHED_RR` scheduling classes.

The increasing interest in hierarchical scheduling is mostly driven by two main motivations. First, with the advent of cloud computing, various services or entire OSs can be remotely executed in virtualized environments. In this case, hierarchical scheduling is needed to control the Quality of Service experienced by the virtualized services or the guest OSs. Second, when adopting component-based design, if the software components to be integrated are composed of multiple tasks, then hierarchical scheduling is the natural solution to support their execution on the same physical machine.

As a consequence, building scheduling hierarchies is common across many different systems, ranging from large-scale servers (used in data centers) to small embedded systems, for which the composition of software developed by different vendors is a standard software engineering practice.

When some of the tasks are characterized by temporal constraints (such as deadlines to be respected), special care is needed to ensure that the schedulability of the hierarchy can be analyzed. In particular, it must be possible to provide *a priori* guarantees on respecting the temporal constraints for the tasks of a group/component, independently of the other groups of tasks running on the physical machine. In other words, when composing real-time systems, it is necessary that the real-time properties of the components are preserved during the integration phase. This can be achieved by adopting *reservation servers* [2], which allow reserving a periodically-recharged execution *budget* to processes.

Contribution. This paper presents a possible implementation of a two-level real-time scheduling hierarchy based on unmodified Linux-based OSs and kvm-based Virtual Machines. The proposal relies on stock software components, namely kvm, and the `SCHED_DEADLINE` scheduling policy of Linux: no custom patches for the Linux kernel are required. Components are executed inside kvm-based VMs, and the tasks of each component are scheduled by a local scheduler (also called the *guest scheduler* to indicate that it is the CPU scheduler of the guest kernel — this is a common terminology in the area of resource virtualization). Each VM represents a virtual execution platform that offers only a fraction of the computing capacity of the physical one. VMs are scheduled by the host scheduler and protected by reservation servers offered by `SCHED_DEADLINE`.

While implementing our hierarchical scheduling solution, we realized that most of the theoretical analysis techniques in the literature overlooked some practical details that become apparent when realizing a real system:

- To guarantee the component schedulability, most of the previous works either assume a local scheduler that has some knowledge on the status of the underlying virtual machine (and such knowledge is often not available if para-virtualized scheduling is not used), or are excessively pessimistic. This aspect is discussed in Section 3.

- Some previous works optimized the amount of resources allocated to the various components by only considering the real-time tasks running into them. Unfortunately, as discussed in Section 5.1 and experimentally verified in Section 6.2, this leads to quite optimistic assumptions.

- Previous works often did not consider the host system software (host kernel, virtual machine monitor, hypervisor, etc.) and only focused on guaranteeing the schedulability of the guest tasks. Unfortunately, the host system might have some important tasks (even non-real-time tasks) that should not be starved by the guests (for example, think about the Linux `kworker` or `ksoftirqd` kernel threads, or similar: if the VMs running on a CPU core starve these kernel threads, the host system can malfunction).

Hence, it might be necessary to limit the amount of CPU time consumed by the VMs on the physical cores. As it is shown in Section 3, this might be quite difficult with state-of-the-art techniques.

We addressed these issues by developing a new approach based on hierarchical partitioned scheduling (based on previous work) that is described in Section 4. Our implementation leverages previous theoretical research on hierarchical uniprocessor real-time scheduling: experimental results are finally presented to show that our implementation matches the corresponding theoretical results.

2. System model and Background

The overall system is composed of a set of *real-time components* $\{\mathcal{C}_1, \ldots, \mathcal{C}_\ell\}$ that run concurrently onto a multiprocessor that comprises M physical CPUs[1]. To enable composability and isolation, each component $\mathcal{C}_j$ is executed onto a dedicated *virtual machine* Π_j. The workload generated by component $\mathcal{C}_j$ is scheduled by a *local scheduler* (or *guest scheduler*) $\mathcal{S}_j$ over the machine Π_j.

The most important definitions used in this paper are summarized in Table 1, which is reported at the end of this section. Since we mostly focus on each component in isolation, to lighten the notation, from now on we drop the index "j" of the component. Hence, we denote the component by just $\mathcal{C}$, the virtual machine by Π, and the guest scheduler by $\mathcal{S}$.

A component $\mathcal{C}$ is a set of n independent real-time tasks $\{\tau_1, \ldots, \tau_n\}$[2]. Each task τ_i releases an infinite sequence of jobs $J_{i,1}, J_{i,2}, J_{i,3}, \ldots$. The h-th job $J_{i,h}$ belonging to task τ_i is released at time $r_{i,h}$, finishes at time $f_{i,h}$, and executes for $c_{i,h}$ time units. We assume the *sporadic* tasks model, i.e., the release instants

[1] The terms "CPU", "core" and "processor" are used interchangeably to mean a hardware component capable to perform computation.

[2] Interactions between different components are not considered as well; see Section 7 for a discussion about interactions between tasks and/or components.

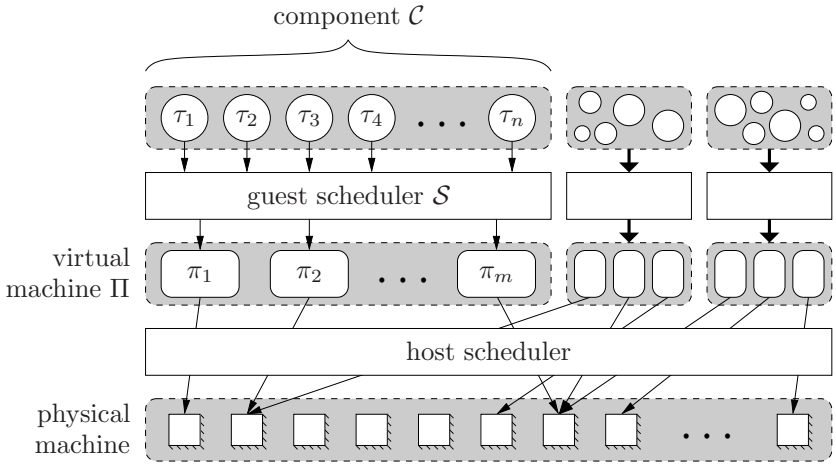


Figure 1: Architecture overview.

of two consecutive jobs of task τ_i are separated by at least the *minimum inter-arrival time* T_i, i.e., $\forall h = 1, 2, \ldots, r_{i,h+1} - r_{i,h} \geq T_i$. The execution time of each job $J_{i,h}$ is bounded by the worst-case execution time (WCET) C_i, that is $\forall h = 1, 2, \ldots, c_{i,h} \leq C_i$. Each job $J_{i,h}$ must be completed not later than its *absolute deadline*, that is $f_{i,h} \leq d_{i,h}$. The deadline $d_{i,h}$ of job $J_{i,h}$ is given by $d_{i,h} = r_{i,h} + D_i$, where D_i is the *relative deadline* of task τ_i. Finally, we assume a *constrained-deadline* model, that is $D_i \leq T_i$ for all tasks τ_i.

A *virtual machine* Π is composed of a set of m *virtual CPUs* (vCPUs) $\{\pi_1, \ldots, \pi_m\}$ on which the tasks of the served component can execute. The resulting system is characterized by a two-level scheduling hierarchy: a *root scheduler* — or *host scheduler* — selects a component to be executed, and then the component's guest scheduler $\mathcal{S}$ selects one or more of the component's tasks. The overall architecture is illustrated in Figure 1. Some of the most important virtualization technologies (with focus on Linux-based OSs and real-time performance) will be described in Section 5.

In real-time components, it is of key importance to guarantee that no deadline will be missed. This is enforced by a *schedulability test*, which is a boolean predicate that is TRUE if the component is *guaranteed* not to miss any deadline.

The schedulability analysis over virtual machines has been investigated by many authors [3, 4, 5, 6, 7, 8]. Leontyev and Anderson [3] proposed to use only the overall bandwidth requirement provided by Π to check the schedulability of a set of tasks. Bini et al. [4] introduced the richer notion of *parallel supply function* (PSF) to model the amount of computation provided by Π over intervals of any length. Easwaran et al. [5] defined the multiprocessor resource model (MPR) to characterize the resource provided by periodic and synchronized servers. Lipari and Bini [6] showed that if the synchronization among virtual CPUs is not feasible, then the worst-case resource supply of MPR can be worse than what assumed in [5]. It was then proposed the bounded-delay multi-partition (BDM) that fixes this issue at the price of some pessimism. Bini et al. [9] and Khalizad et al. [7] independently proposed a method to adapt the amount of resource allocated by a set of virtual CPUs. Burmyakov et al. [8] proposed the generalized multi-processor resource model (GMPR) showing that uneven budget distribution among the virtual CPUs is more favorable to the task set to be scheduled. Such an intuition was later proved by Yang and Anderson [10].

These methods only offer a theoretical analysis of hierarchical systems over virtual machines. A notable exception is RT-Xen [11], that relies on the widely-used Xen hypervisor and has been partly merged in the mainline version of Xen. Since Xen is a so-called bare-metal hypervisor, a natural question to be asked is if a predictable real-time scheduling hierarchy can be implemented using different virtualization technologies; for example, a hosted hypervisor such as kvm.

2.1. Reservation-based scheduling of virtual machines

A reservation-based scheduler is based on the idea of scheduling work for an amount of time Q every period P. When applying reservation-based scheduling to virtual machines, the time provided by the host scheduler to a virtual CPU π_k (and to the component's tasks selected to execute over π_k by $\mathcal{S}$) can be modeled by:

- a *budget* Q_k of time provided by the k-th virtual CPU π_k, and

- a *period* P_k of the allocation of the budget Q_k over time.

In this model, a virtual CPU π_k is viewed by the host scheduler as a task requiring a budget Q_k every period P_k.

A common method to analyze the component $\mathcal{C}$ over a virtual machine Π is to determine the minimum amount of computation provided by Π over any interval of length t. The notion of *supply function* fully captures this quantity for one single virtual CPU π, (or for virtual machines with composed by only one virtual CPU, $m = 1$) [12, 13, 14, 15, 16].

The *supply bound function* $\mathsf{sbf}_k(t)$ associated to the k-th virtual CPU π_k is a function such that [12, 14, 15]

$$\forall (a, b), \mathsf{sbf}_k(b - a) \leq \int_a^b \xi_k(t)\, dt \tag{1}$$

with $\xi_k(t)$ being the indicator function of the schedule of the k-th virtual CPU by the host scheduler, that is

$$\xi_k(t) = \begin{cases} 1 & \text{if } \pi_k \text{ is scheduled by the host scheduler at time } t \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

The property of (1) implies that $\mathsf{sbf}_k(t)$ is a lower bound for the amount of resource made available by the virtual CPU π_k in any interval of length t. Such a lower bound can be computed by considering all the possible time intervals of size t, and all the possible schedules generated by the host scheduler.

For a reservation-based host scheduler (with budget Q_k and period P_k), the worst-case resource allocation for a real-time task τ (which leads to the supply bound function $\mathsf{sbf}(t)$) happens when the root scheduler provides an amount of budget Q_k to lower priority tasks at the beginning of a reservation period and τ arrives Q_k time units after the beginning of the reservation period, immediately after the budget has been consumed [15]. Hence, the task cannot be scheduled on a physical CPU until the next reservation period (after a time interval equal to $P_k - Q_k$). If in the next reservation period the root scheduler provides the budget Q_k at the end of the period, then the total delay τ has to wait to be

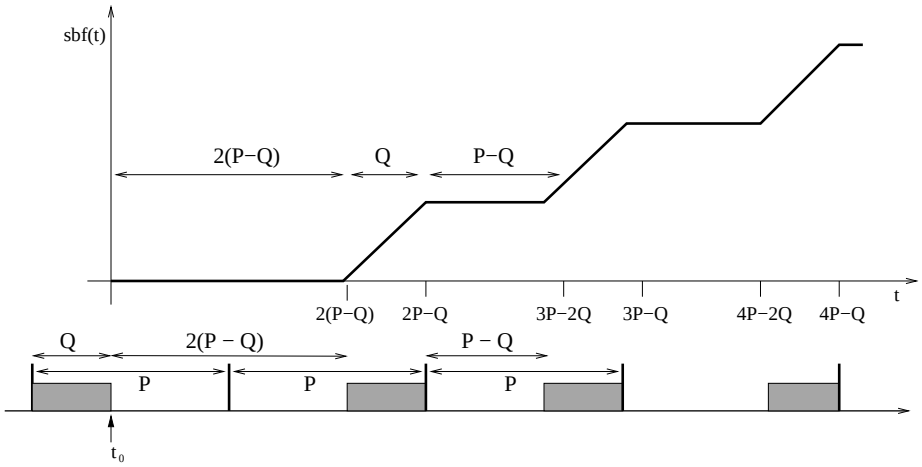


Figure 2: Supply bound function for a CPU reservation (upper part of the figure) and scenario leading to it (lower part of the figure).

scheduled is $2(P_k - Q_k)$. This scenario is shown in the lower part of Figure 2, where t_0 indicates the arrival time of τ (hence, the interval of size t considered by $\mathsf{sbf}_k(t)$ starts at time t_0). The upper part of the figure shows the shape of the resulting $\mathsf{sbf}_k(t)$: the function is equal to 0 for an amount of time $2(P_k - Q_k)$, then it increases with slope 1 for an amount of time Q_k (hence, it reaches value Q_k at time $2P_k - Q_k$, then it is flat for an amount of time $P_k - Q_k$, it increases with slope 1 for an amount of time Q_k, and so on. More formally, $\mathsf{sbf}_k(t)$ can be computed according to the following expression:

$$\mathsf{sbf}_k(t) = \begin{cases} 0 & \text{if } t < 2(P_k - Q_k) \\ (n-1)Q_k & \text{if } nP_k - Q_k \leq t < (n+1)P_k - 2Q_k \\ t - (n+1)(P_k - Q_k) & \text{if } (n+1)P_k - 2Q_k \leq t < (n+1)P_k - Q_k \end{cases} \tag{3}$$

As highlighted by many authors [12, 13, 14, 15], the supply function $\mathsf{sbf}_k(t)$ can be lower bounded by a line with equation $\alpha_k(t - \Delta_k)$, with:

- $\alpha_k = \frac{Q_k}{P_k}$, *bandwidth* of the virtual CPU π_k, and
- $\Delta_k = 2(P_k - Q_k)$, called *delay*, represents the longest interval in which no resource is made available by the host scheduler to the virtual CPU.

In multi-processor systems (assuming the presence of M physical CPUs), multiple virtual CPUs are generally assigned to each component Π; hence, Π is served by m reservations $\{(Q_0, P_0), (Q_1, P_1), ...(Q_{m-1}, P_{m-1})\}$, and both the host scheduler and the guest scheduler must schedule tasks on multiple CPUs (specifically, M CPUs for the host scheduler and m virtual CPUs for the guest scheduler of Π). A multiprocessor real-time scheduler can be implemented as a *global scheduler* or as a *partitioned scheduler*:

- a global scheduler is free to migrate tasks between different cores in accordance with the desired scheduling policy. Hence, conceptually, the scheduler uses one single global ready queue that contains all the tasks ready for execution,
- when a partitioned scheduler is used, tasks are statically assigned to cores by the system designer (or the application programmer), and the OS kernel is not allowed to migrate the tasks. As a consequence, the problem of scheduling tasks on m CPUs is reduced to m scheduling problems on a single CPU, and single-processor scheduling algorithms can be reused. Here, the main challenge is how to partition the tasks among the available cores.

3. Issues with global guest schedulers

The adoption of global scheduling algorithms for the guest scheduler was previously analyzed by several papers [4, 5, 6, 7, 8]. Some of these works, however, implicitly assumed some form of interaction between the guest scheduler and the host scheduler, which is not easily implementable with most of the state-of-the-art virtualization mechanisms. Specifically, popular global schedulers such as Global Earliest Deadline First (G-EDF) and Global Fixed-Priorities (G-FP) are characterized by the key *invariant* stating that, at each time, the x highest-priority ready tasks (with $x \leq M$) are scheduled on the M physical CPUs.

Symbol	Meaning
$\mathcal{C}$	Component (set of real-time tasks)
Π	Virtual machine (set of virtual CPUs on which a component executes)
π_k	k-th virtual CPU of the virtual machine
$\mathcal{S}$	Guest scheduler (CPU scheduler used to schedule tasks in the component)
τ_i	i-th task of the component
$J_{i,h}$	h-th job (activation) of task τ_i
$r_{i,h}$	arrival time of job $J_{i,h}$
$c_{i,h}$	execution time of job $J_{i,h}$
$f_{i,h}$	finishing time of job $J_{i,h}$
$d_{i,h}$	absolute deadline of job $J_{i,h}$ (respected if $f_{i,h} \leq d_{i,h}$)
T_i	minimum inter-arrival time of task τ_i ($r_{i,h+1} - r_{i,h} \geq T_i$)
C_i	worst-case execution time (WCET) of task τ_i ($r_{i,h+1} - r_{i,h} \geq T_i$)
D_i	relative deadline of task τ_i ($d_{i,h} = r_{i,h} + D_i$)
Q_k	maximum budget for virtual CPU π_k
P_k	reservation period for virtual CPU π_k
$\xi_k(t)$	indicator function for π_k ($\xi_k(t) = 1$ if π_k is scheduled on a physical CPU at time t)
$\mathsf{sbf}_k(t)$	supply bound function for virtual CPU π_k (minimum amount of time that π_k is guaranteed to receive in a time interval of size t)
α_k	bandwidth $\frac{Q_k}{P_k}$ of virtual CPU π_k (fraction of the physical CPU time reserved to π_k)
Δ_k	maximum allocation delay for virtual CPU π_k (size of the longest time interval in which π_k is not scheduled on physical CPUs)
Γ_k	set of component's tasks allocated to virtual CPU π_k in partitioned scheduling

Table 1: Definitions of the symbols used in the paper.

This assumption is also made by any job-level fixed-priority work-conserving scheduler.

When using a global scheduling algorithm as a guest scheduler, this invariant has to be extended to consider the combination of guest and host scheduler. In particular, a stronger invariant is needed to avoid priority inversion. Virtual

CPUs have to be considered in place of physical CPUs, paying attention at the fact that a virtual CPU may not always be able to provide service (e.g., when the budget of the corresponding reservation server is depleted). The resulting invariant should hence state that, at each time t, the x highest-priority ready tasks are scheduled on m' physical CPUs (with $x \leq m'$), where m' is the number of virtual CPUs that are scheduled on physical CPUs at time t. This means that the x highest-priority ready tasks must not be scheduled in m' random virtual CPUs, but must be scheduled exactly in the m' virtual CPUs that are currently scheduled on physical CPUs. Without this assumption, the hierarchical schedulability analysis has to be much more pessimistic; hence, most of the previous schedulability analysis of hierarchical scheduling systems with a global guest scheduler relies (more or less implicitly) on this assumption, even if it has not been explicitly mentioned.

The extended invariant mentioned above has an important implication: *the guest scheduler may be a sound global scheduler only if it is aware of the state of each virtual CPU at each time.* This poses considerable issues when there is strong isolation between the guest scheduler and the host scheduler, as in virtualized systems, where a guest operating system may totally be unaware of the fact that is running upon virtual CPUs. As a result, despite their intrinsic pessimism [17], most schedulability tests for global schedulers result to be incompatible with hierarchical scheduling if no proper strategies are adopted. In other words, to use a terminology much closer to virtualization techniques, a *para-virtualized guest scheduler* is often required to safely implement global scheduling. These claims can be verified with the following counterexamples.

3.1. Counterexamples

To better understand this issue, consider as an example a component $\mathcal{C}$ composed of two tasks: τ_1 with $C_1 = 60$ and $T_1 = D_1 = 100$ and τ_2 with $C_2 = 10$ and $T_2 = D_2 = 200$. τ_1 has higher priority than τ_2. The component $\mathcal{C}$ is scheduled over a virtual machine Π composed of two virtual CPUs π_1 and π_2, associated to two reservations with budget and period $Q_1 = 90$, $P_1 =$

100, and $Q_2 = 40$, $P_2 = 100$, respectively. Task τ_1 (the highest-priority task) is intuitively schedulable without missing any deadline if the guest scheduler always schedules it on π_1 (scheduling τ_2 on π_2). This can be confirmed by using the PSF analysis [4]. According to Theorem 1 in [6], task τ_i is schedulable if

$$\exists k \in \{1, 2, \ldots, m\} : kC_i + W_i \leq \mathsf{psf}_k(D_i) \tag{4}$$

where W_i is the interfering workload from higher-priority tasks and $\mathsf{psf}_k(t)$ is the level-k PSF for the component. Notice that since the guest scheduler is based on fixed priorities and τ_1 is the highest-priority task in the component, it holds that $W_1 = 0$. According to Lemma 1 (reported in Appendix A), the $\mathsf{psf}_1(t)$ for a component served by m periodic servers is $\mathsf{psf}_1(t) = \max_k\{\mathsf{sbf}_k(t)\}$. Hence, for component $\mathcal{C}$, $\mathsf{psf}_1(t)$ is the maximum between $\mathsf{sbf}_1(t)$ and $\mathsf{sbf}_2(t)$. Let us now check the schedulability of task τ_1 onto $\Pi = \{\pi_1, \pi_2\}$ when scheduled by G-FP. From Equation (3), we have that $\mathsf{sbf}_1(100) = 80$ (since $Q_1 = 90$ and $P_1 = 100$), hence $\mathsf{psf}_1(100) \geq 80$. Then, by applying the schedulability condition in (4), it holds that

$$C_1 + W_1 \leq \mathsf{psf}_1(D_1) \;\Leftrightarrow\; 60 + 0 \leq \mathsf{psf}_1(100) = 80,$$

from which it follows that τ_1 is schedulable according to PSF analysis. However, if the guest scheduler has no information about the available budgets for the two virtual CPUs, it can schedule τ_1 on π_2 and not migrate the task when the budget of π_2 is exhausted, with the result that τ_1 will miss its deadline. More specifically, consider the simple case in which π_2 is idle and is able to provide service as soon as τ_1 is released: in this case, the task will execute for 40 time units and then stopped, the server budget is recharged after 100 time units, and hence τ_1 will miss its deadline.

To avoid this issue, the guest scheduler should be informed about the amount of budget that is currently available to each virtual CPU, so that a high-priority task scheduled on a virtual CPU with a depleted budget can be *migrated* to another virtual CPU with a positive budget. This example has also been tested by means of an implementation on a real Linux-based system, which will be

later described in Section 5: not surprisingly, the experiments confirmed that the issue can really happen in practice. Please refer to Section 6.4 for further details.

The MPR analysis is more pessimistic than the PSF analysis. Indeed, if the previous component $\mathcal{C} = \{(60, 100), (10, 200)\}$ is assumed to be executed upon a MPR interface $\{(90, 100), (40, 100)\}$ (i.e., with two virtual processors), the corresponding MPR schedulability test [5] for G-EDF scheduling deems the component unschedulable. Nevertheless, note that the MPR explicitly assumes that the component disposes of all the computing supply offered by the interface [5]. In fact, with the following counterexample it can be shown that also MPR is affected by this issue when the interface is implemented with reservation servers configured with asymmetric parameters. Consider a component $\mathcal{C}$ composed of two tasks: τ_1 with $C_1 = 50$ and $T_1 = D_1 = 100$ and τ_2 with $C_2 = 2$ and $T_2 = D_2 = 100000$. According to the MPR analysis for G-EDF, this component results schedulable upon a MPR interface with budget 151, period 100, and two virtual CPUs, namely π_1 and π_2. This interface can be implemented with two reservation servers $(Q_1 = 51, P_1 = 100)$ and $(Q_2 = 100, P_2 = 100)$. If τ_1 is scheduled on π_1, it can suffer a worst-case service delay $\Delta_1 = 2(P_1 - Q_1) = 98$. The task will then miss its deadline after two time units. Note that, if the guest scheduler would be informed about the state of π_1, it can avoid assigning τ_1 to π_1 when its budget is exhausted, hence guaranteeing its schedulability. Indeed, the two tasks would also be schedulable if the guest scheduler only uses π_2, but this is an information the scheduler cannot know if it is not informed of the state of the virtual CPUs. Note that this task set is also schedulable with G-EDF upon a MPR interface with budget 125, period 80, and two virtual processors, which can be implemented with two servers $(Q_1 = 45, P_1 = 80)$ and $(Q_2 = 80, P_2 = 80)$: in this case, τ_1 can miss its deadline even if it incurs a worst-case service delay $P_1 - Q_1$.

3.2. Other issues

Despite the problem described above, global scheduling in the guest carries other issues because of its pessimistic analysis, as it is highlighted by the schedulability analysis of MPR interfaces. For instance, consider a component $\mathcal{C} = \{(21, 23), (2, 23), (2, 27), (24, 367), (31, 419)\}$[3]. The total utilization of the taskset is about $\sum_i \frac{C_i}{T_i} = 1.21$, but the taskset requires an MPR interface with utilization at least 4.9 (5 CPUs) to be scheduled. If some copies of the last task (having $C = 31$, $T = 419$ and a utilization $31/419 = 0.07$) are added to the component, the MPR interface requires one additional CPU for every added task. These results have been confirmed by using the CARTS tool for compositional scheduling analysis [18] (`https://rtg.cis.upenn.edu/carts`). Section 6.5 presents a more detailed investigation of this issue.

This issue is particularly relevant in practice when the host system needs to leave some spare CPU time on each core. An intuitive way to achieve this result can be to increase the number of virtual CPUs used in the virtual platform. Unfortunately, this approach does not work well with global guest scheduling, because increasing the number of virtual CPUs also increases the amount of CPU time needed to correctly schedule the component. For instance, this can be shown with the following example. Consider a physical platform (host) composed of $M = 4$ physical CPU cores, and a component $\mathcal{C} = \{(2, 73), (21, 118), (11, 53), (12, 132), (9, 48), (23, 86), (26, 229), (81, 278)\}$[4]. According to the theoretical analysis in [5], the minimal MPR interface needed to schedule this component with period 10 has utilization 1.7 on $m = 2$ virtual CPUs.

Now, assume that the host system needs 30% of the CPU time on every core for its own tasks: in this case, the MPR interface cannot be allocated

[3] This is a simplification of one of the tasksets randomly generated for the experiments of Section 6.5.

[4] While the example has been simplified for making it more understandable, a large number of experiments with randomly generated tasksets have been performed, changing the taskset utilization, the number of tasks, and the number of physical and virtual CPUs. All the experiments generated results consistent with the ones reported here

on the physical CPUs. A system designer can try to increase the number of virtual CPUs, hoping to decrease the per-core CPU load, but unfortunately, an MPR interface with 4 virtual CPUs requires a utilization 2.9 to schedule the component. Again, at least one of the periodic servers used to implement the MPR interface must consume more than 70% of the physical core, making the MPR interface not possible to allocate to physical cores.

Conversely, if adopting a partitioned scheduling approach in the guest, it is possible to allocate tasks $\{(9, 48), (21, 118)\}$ on virtual CPU 0, tasks $\{(23, 86), (26, 229)\}$ on virtual CPU 1, tasks $\{(11, 53), (12, 132)\}$ on virtual CPU 2, and tasks $\{(2, 73), (81, 278)\}$ on virtual CPU 3. This allocation can be scheduled with CPUs configures with budgets and periods $(8, 18)$ (utilization 0.44), $(8, 18)$, $(5, 14)$ (utilization 0.36) and $(9, 26)$ (utilization 0.34), hence leaving available the desired spare bandwidth on each physical CPU.

Because of the issues discussed in this section, and to dispose of proper theoretical foundations for the analysis of guest schedulers, this paper adopts a solution based on *partitioned* guest schedulers, discussed in the next section (which provides a way to automatically partition the tasks as in the previous example).

4. The proposed scheduling approach

Motivated by the issues identified in the previous section, this paper adopts *partitioned fixed-priority* (P-FP) scheduling for the guest scheduler. Under P-FP, each task is assigned a unique static priority and we denote by $\mathsf{hp}(i) \subseteq \mathcal{C}$ the subset of tasks with higher priority than the one of τ_i. Also, each task is statically allocated to a single virtual processor π_k and we denote by Γ_k the set of tasks allocated to virtual processor π_k.

In partitioned scheduling, the schedulability of the component can be checked by applying single-processor analysis to each subset Γ_k allocated to the virtual processor π_k. According to several results from the literature [12, 13, 14, 15, 16,

19], the tasks allocated to virtual processor π_k are schedulable if

$$\forall \tau_i \in \Gamma_k,\ \exists t \in \mathsf{tSet}_i, \qquad C_i + \sum_{\tau_j \in \mathsf{hp}(i) \cap \Gamma_k} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq \mathsf{sbf}_k(t), \tag{5}$$

where tSet_i denotes the set of schedulability points at which the schedulability condition is tested.

Several definitions of tSet_i were proposed, ranging from proposals that enable an exact (i.e., necessary and sufficient) schedulability test, such as those presented by Lehoczky et al. [20] later refined by Bini and Buttazzo [21], up to proposals that allow performing an efficient polynomial-time schedulability test with near-optimal empirical performance, such as the ones presented by Park and Park [22]. It is worth observing that the approaches proposed in this paper are compatible with any definition of tSet_i. Nevertheless, to make the paper self-consistent, the original definition proposed by Lehoczky et al. [20] is reported: $\mathsf{tSet}_i = \{zT_j \mid \tau_j \in hp(i) \cup \tau_i \ \wedge z = 1, \ldots, \lfloor T_i/T_j \rfloor\}$.

4.1. Task partitioning and virtual processor design

Designing a partitioned guest scheduling algorithm requires two steps:

1. deciding which virtual processor each task must be allocated to (partitioning); and
2. designing the reservation parameters (i.e., budget and period) for each virtual processor such that the served tasks are schedulable.

Furthermore, it is necessary to specify the design objectives to select a solution among all the feasible ones.

The joint consideration of these aspects, to develop a holistic methodology, requires solving complex non-linear optimization problems that are possibly intractable for practical scenarios. To overcome these limitations, this work adopts a two-stage approach inspired by the methodology presented in [23], which is based on decoupling the task partitioning from the design of the reservation servers. This methodology is individually applied to each component $\mathcal{C}$ in isolation, i.e., it is independent of the tasks and the server parameters of the other

components. First, given a design objective, the tasks in $\mathcal{C}$ are *optimally* partitioned upon a set of "fluid" virtual processors, which are abstracted by the only bandwidth α_k. Later, given the partitioning obtained at the first stage, the server parameters (budget Q_k and period P_k) of each virtual processor are designed such that the server bandwidth is *minimized* while the served tasks are guaranteed to be schedulable.

4.2. Task partitioning

The first stage is performed by setting up a *mixed-integer linear program* (MILP). The MILP formulation uses the following optimization variables:

- $x_{i,k} \in \{0,1\}$, a binary variable such that $x_{i,k} = 1$ if and only if task τ_i is allocated to virtual processor π_k;
- $\alpha_k \in [0,1]$, a real variable that expresses the bandwidth of virtual processor π_k;
- $p_{i,q} \in \{0,1\}$, a binary variable such that $p_{i,q} = 1$ if the schedulability condition of task τ_i is verified at the q-th scheduling point of the set tSet_i.

Variables $x_{i,k}$ are constrained as follows to ensure that each task is allocated to exactly one virtual processor.

Constraint 1.

$$\forall \tau_i \in \mathcal{C},\ \sum_{k=1}^{m} x_{i,k} = 1.$$

The component schedulability is enforced with the following constraint, which is based on Equation (5) assuming a fluid supply bound function, i.e., $\mathsf{sbf}_k(t) = \alpha_k\ t$.

Constraint 2. $\forall k = 1,\ldots,m,\ \forall \tau_i \in \mathcal{C},\ \forall q = 1,\ldots,|\mathsf{tSet}_i|,$

$$C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \alpha_k \cdot \mathsf{tSet}_i[q] + \mathcal{M} \cdot (2 - p_{i,q} - x_{i,k}), \quad (6)$$

where $\mathsf{tSet}_i[q]$ *denotes the q-th scheduling point in set* tSet_i *and* $\mathcal{M}$ *is a large constant always larger than any LHS of the inequality.*

Please refer to Appendix B for a proof of the above constraint.

Finally, to ensure that the schedulability condition provided by Equation (5) is verified in at least one of the scheduling points in tSet_i, the following constraint is enforced.

Constraint 3.

$$\forall \tau_i \in \mathcal{C}, \quad \sum_{q=1}^{|\mathsf{tSet}_i|} p_{i,q} \geq 1.$$

To solve this MILP formulation, two design objectives are considered:

A) to minimize the overall bandwidth required by the virtual processors of the component, i.e., $\mathsf{minimize} \ \sum_{k=1}^{m} \alpha_k$. This strategy tends to allocate the tasks on a small set of virtual processors with high bandwidth.

B) to minimize the maximum bandwidth required by the virtual processor of a component, i.e., $\mathsf{minimize} \ \max_{k=1,\ldots,m} \alpha_k$. This strategy facilitates the packing of vCPUs since it is minimized the size of the largest needed vCPU.

The complete MILP formulation is summarized in Equation (7).

$$\textbf{minimize} \sum_{k=1}^{m} \alpha_k \text{ (objective A)} \quad \textbf{or} \quad \max_{k=1,\ldots,m} \alpha_k \text{ (objective B)}$$

subject to

$$\begin{aligned}
&\forall \tau_i \in \mathcal{C}, \ \sum_{k=1}^{m} x_{i,k} = 1 \\
&\forall k = 1, \ldots, m, \ \forall \tau_i \in \mathcal{C}, \ \forall q = 1, \ldots, |\mathsf{tSet}_i|, \\
&C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \alpha_k \cdot \mathsf{tSet}_i[q] + \mathcal{M} \cdot (2 - p_{i,q} - x_{i,k}) \\
&\forall \tau_i \in \mathcal{C}, \ \sum_{q=1}^{|\mathsf{tSet}_i|} p_{i,q} \geq 1
\end{aligned} \tag{7}$$

Note that strategy B tends to reduce the per-physical-CPU utilization, hence solving one of the issues pointed out in Section 3.2 by leaving some spare CPU

bandwidth on each core to execute host threads. A more explicit control of this issue can be achieved by introducing additional constraints to the MILP formulation that enforce the server bandwidths to be lower than some given thresholds.

4.3. Server design

Given the task partitioning resulting from the first stage, the server parameters for each virtual processor are designed. This step is performed by means of a branch-and-bound optimization algorithm that relies on the schedulability test of Equation (5), but this time adopting the exact supply bound function of Equation (3). To the end of minimizing the server bandwidth while still guaranteeing the schedulability of the served task set, the design algorithm explores the space of feasible budgets Q_k and periods P_k with given granularities and minimum values for the two parameters. Since given the same server bandwidth larger server periods are preferable (e.g., to reduce the intervention of the scheduler when enforcing the budgeting mechanism), the design algorithm considers $(Q_k + \sigma)/P_k$ as objective function to minimize, with σ being a tunable parameter that models the scheduling overhead. The design algorithm exploits that such an objective function decreases as the server period increases, thus limiting the number of branches to be explored.

Algorithm 1 reports the pseudo-code of the proposed design strategy. The algorithm takes as input the partition $\Gamma_k \subseteq \mathcal{C}$ of the task set $\mathcal{C}$ that has been allocated to virtual processor π_k by the solving the MILP formulation reported in the previous section. Furthermore, the algorithm returns the pair (Q_k, P_k) representing the budget and the period of the designed reservation server. When the algorithm is not able to produce a suitable design (e.g., due to a task set with very large utilization), **unschedulable** is returned. With the two loops at lines 6 and 7, the algorithm explores the space of parameters α and Δ for the server with given granularities α^{step} and Δ^{step}, respectively. Parameter α (the server bandwidth) must clearly be larger than the task set utilization U and lower or equal to the whole processor supply ($\alpha = 1$). Parameter Δ is explored

Algorithm 1 Server design algorithm.

Input: subset $\Gamma_k \subseteq \mathcal{C}$ including tasks allocated to π_k

Output: budget and period (Q_k, P_k) of the server, or **unschedulable**.

```
1: procedure DESIGNSERVER(Γ_k)
2:     (Q_k, P_k) ← (2, 1)
3:     α* ← 2
4:     U = Σ_{τ_i ∈ Γ_k} C_i/T_i
5:     Δ^UB = max_{τ_i ∈ Γ_k} {D_i}
6:     for α = U to α = 1 with step α^step do
7:         for Δ = Δ^UB to Δ = Δ^step with step Δ^step do
8:             P ← Δ/(2(1 − α))
9:             Q ← αP
10:            (Q, P) ← round_values(Q, P)
11:            if is_schedulable(Γ_k, Q, P) then
12:                if (Q + σ)/P < α* then
13:                    α* ← (Q + σ)/P
14:                    (Q_k, P_k) ← (Q, P)
15:                    break
16:                end if
17:            end if
18:        end for
19:    end for
20:    if (Q_k, P_k) == (2,1) then
21:        return unschedulable
22:    else
23:        return (Q_k, P_k)
24:    end if
25: end procedure
```

up to its minimum granularity Δ^{step} by starting from an upper bound Δ^{UB}. By looking at Equation (5) and by recalling that the definitions of sets tSet_i do not include values larger than the maximum relative deadlines of the tasks into $\mathcal{C}_k$, it follows that $\max_{\tau_i \in \Gamma_k}\{D_i\}$ yields a safe upper bound for Δ.

The algorithm proceeds by computing the budget and the period corresponding to the analyzed pair (α, Δ), having care of rounding their values as discussed above (line 10). If the task set Γ_k is schedulable with a reservation server configured with the current pair of values (Equation (5) is used at this stage), then the algorithm verifies whether the minimum server bandwidth α^* can be reduced with the current parameters. If yes, the current parameters are stored in the pair (Q_k, P_k). In addition, since fixed a value of α the objective function $(Q + \sigma)/P$ cannot improve with lower values of Δ, the inner loop is stopped as soon as the algorithm finds a pair of parameters that allows deeming the task set schedulable, hence pruning unnecessary branches.

4.4. Putting the pieces together

For the sake of completeness, Algorithm 2 reports the pseudo-code of the main algorithm that shows how to integrate the MILP formulation of Equation (7) with Algorithm 1. The algorithm takes as input the task set $\mathcal{C}$ and first solves the MILP formulation summarized in Equation (7). This step allows assigning tasks to virtual processor π_k, with $k = 1, \ldots, m$, hence defining the subsets Γ_k. Then, for each subset Γ_k, a corresponding server is designed by means of Algorithm 1. If all the server designs succeed, the resulting budgets and periods of each virtual processor are returned. Otherwise, **unschedulable** is returned.

4.5. Example

This section reports a simple example to concisely illustrate the results of the steps described in the previous sections. Consider the task set reported in Table 2 to be partitioned upon a virtual machine composed of $m = 2$ virtual processors π_1 and π_2.

After solving the MILP formulation of Equation (7), the following partitioning are obtained:

- In the case objective A is selected, τ_1, τ_2, and τ_4 are allocated to π_1, while τ_3 is allocated to π_2. The corresponding values of variables α_k are 0.62

Algorithm 2 Task partitioning and server design.

Input: task set $\mathcal{C}$

Output: m pairs of budget and period (Q_k, P_k), or **unschedulable**.

1: **procedure** PARTITIONINGANDDESIGN($\mathcal{C}$)
2: Solve the MILP formulation of Equation (7) for $\mathcal{C}$
3: **for** $k = 1 \ldots, m$ **do**
4: $\Gamma_k \leftarrow \{\tau_i \in \mathcal{C} \mid x_{i,k} = 1\}$
5: $(Q_k, P_k) \leftarrow$ DESIGNSERVER(Γ_k)
6: **if** $(Q_k, P_k) ==$ **unschedulable then**
7: **return unschedulable**
8: **end if**
9: **end for**
10: **return** $\{(Q_1, P_1), \ldots, (Q_m, P_m)\}$
11: **end procedure**

Table 2: Example task set.

Task	C_i	T_i	D_i
τ_1	2	10	10
τ_2	3	25	25
τ_3	14	35	35
τ_4	15	50	50

and 0.4, respectively.

- In the case objective B is selected, τ_1 and τ_4 are allocated to π_1, while τ_2 and τ_3 are allocated to π_2. The corresponding values of variables α_k are both 0.57.

Then, the server design strategy presented in Section 4.3 can be applied. For the objective A, the approach produces two reservation servers with budget and period $(7, 10)$ and $(7.5, 14)$, respectively. Conversely, for the objective B the approach produces two servers with budget and period $(6, 10)$ and $(7.5, 11)$, respectively. Note that the total bandwidth consumed by the servers is 1.24 in

the first case, and 1.28 in the second case.

5. Hierarchical Scheduling on Linux

A scheduling hierarchy such as the one described in the previous sections can be implemented in many different ways. One of the simplest and most effective approaches is based on Virtual Machines (VMs), and consists in implementing a two-level hierarchical scheduling system by executing each component $\mathcal{C}$ over a dedicated VM. The various VMs can be scheduled by a hypervisor (such as Xen [24]) or a *host OS* (if solutions similar to kvm [25] are used), and the OSs running inside the VMs are referred to as *guests*. The root scheduler is then responsible for selecting a VM for execution, and the local scheduler (responsible for selecting the actual task to be executed) is implemented by the scheduler in the guest OS kernel.

Today, several kinds of virtualization techniques are available for different architectures. A first distinction can be made between *full hardware virtualization* and *container-based virtualization* (also known as *OS-level virtualization*). In the case of full hardware virtualization, the VM implements all the hardware details of a real machine (including all the needed I/O devices). In this way, a guest OS, including its own kernel, can execute in it as if it would execute on real hardware; if commonly available I/O devices are emulated, any unmodified OS can run in the VM without being aware of the fact that it is not running on real hardware. Conversely, if some "special" virtual devices are provided, the guest OS needs to be aware that it is running inside a VM (and must know how to handle the virtual devices), but can achieve better performance. The term *para-virtualization* is used to indicate when a guest OS is aware of running in a VM (and its kernel is modified to improve the virtualization performance, to exploit some features of the VM, or similar).

OS-level virtualization, instead, virtualizes only the OS kernel, and not the whole hardware on which it is running. In this case, there is one single OS kernel (the host kernel) that virtualizes the offered services to provide some form of

isolation and security between different guests. Hence, host OS and guest OS share the kernel, and must run on the same hardware architecture (note that it is not possible to execute Windows guests on Linux hosts or to run ARM-based guest OSs on an Intel-based host). For example, it is possible to use the same hardware to run multiple distributions based on the same Linux kernel (or multiple Linux applications/containers) on the same server. Every container or OS distribution will be isolated from the others, having the impression to be the only one running on the kernel. On Linux-based systems, this kind of OS-level virtualization can be implemented by using *containers*, *control groups*, and *namespaces* to isolate various kernel resources (other OSs provide different mechanisms; for example, BSD-based OSs provide *jails*). A user-space tool such as `lxc` or `docker` is responsible for setting up the kernel containers, control groups and namespaces, and executing the guest OS inside them.

In this paper, we focus on full hardware virtualization. Whenever possible, the performance of full hardware virtualization can be improved by allowing the guest machine language instructions to execute on the host CPUs [26] directly. The software component that is responsible for controlling the execution of guests is named *hypervisor*. A hypervisor can directly execute on the hardware (bare-metal hypervisor, like Xen [24]) or can rely on an OS kernel (hosted hypervisor, such as kvm). While a bare-metal hypervisor must also contain a VM scheduler (the host scheduler in our hierarchy), a hosted hypervisor relies on the host kernel scheduler for scheduling the VMs, which are seen by the host kernel as processes or threads.

Some of the previous work in the literature focused on implementing hierarchical real-time scheduling based on bare-metal hypervisors. For example, RT-Xen [11] modified the Xen VM scheduler to implement deferrable servers [27, 28] or polling servers (based on fixed priorities or on EDF), that can provide CPU reservations. Hence, the various virtual CPUs can be assigned budgets and periods as done in this work. Based on this scheduler, RT-Xen allows providing real-time guarantees to real-time tasks scheduled in the guests using partitioned or global strategies [29]. However, it is not clear how the problem mentioned

in Section 2 is addressed: when using global scheduling in the guest, real-time guarantees are provided using the MPR interface [5], which also requires some form of para-virtualized scheduling (like our previous example). The deferrable server mechanism implemented in RT-Xen has been integrated in the vanilla Xen hypervisor, hence no additional patches are needed to use it. In the context of large-scale cloud systems the overhead introduced by the hypervisor scheduler might be too high, hence has also been proposed to use a static, table-driven, approach in the Xen scheduler [30].

Other works [31] focused on using a μkernel, such as Fiasco [32] as a bare-metal hypervisor, and a para-virtualized version of Linux (named l4linux [33]) as a guest kernel. The schedulability analysis for these hierarchical schedulers has been limited to only one single virtual CPU.

In the context of hosted hypervisors, some previous works used a modified Linux scheduler to schedule qemu/kvm [19, 34], but again focused on VMs with one single virtual CPU.

5.1. Proposed architecture

In this work, we are interested in using a vanilla Linux scheduler (without any additional patches) for scheduling the VMs, focusing on multi-processor issues.

The most commonly-used hosted hypervisor on Linux is based on the kvm (kernel-based virtual machine) module, that allows using some specific hardware features to virtualize the CPU. For example, it is well known that the Intel ISA was traditionally not virtualizable (in the sense of [26]), but the "Intel VT-x" extensions (`vmx` in the CPU flags) introduce a virtualizable ISA to allow for a safe and controlled execution of all the guest code. Kvm allows using this technology. This possibility is then used by some user-space code (generally `qemu` [35]), that is in charge of setting up the VM and emulating the I/O devices.

When using kvm to execute guest code, qemu creates a thread for each virtual CPU π_k of the VM (named vCPU thread). As a result, each virtual CPU corresponds to a thread in the host system, which can be scheduled by using

one of the scheduling policies provided by Linux. As discussed in Section 4.3, we need to schedule each virtual CPU π_k with a reservation-based scheduler, assigning a budget Q_k and a period P_k to it. By default, a vanilla Linux kernel (without any additional patch) provides the `SCHED_DEADLINE` [36] scheduling class that implements a reservation-based scheduler [2], which allows assigning the desired scheduling parameters (Q_k, P_k) to each scheduled task. Hence, it can be used as a host scheduler to implement the approach described in Section 4.

Wrapping up, the hierarchical scheduling solution proposing in this paper is based on:

- running each component $\mathcal{C}$ in a dedicated kvm-based VM (using the qemu program) with m virtual CPUs;
- using the `SCHED_FIFO` scheduling policy in the guest Linux kernel to schedule the component's real-time tasks with fixed priorities;
- partitioning the real-time tasks of the components between the m virtual CPUs as explained in Section 4.2, which means that each task is assigned to a virtual CPU by setting its affinity to a single CPU; and
- scheduling each one of the m vCPU threads π_k of the VM with the host's `SCHED_DEADLINE` policy and the scheduling parameters (Q_k, P_k) computed by Algorithm 1.

The `SCHED_DEADLINE` policy in the host kernel can schedule the vCPU threads on multiple physical CPUs by using a partitioned or a global approach. If global scheduling is used, then a global admission test [37, 38] must be used; otherwise, the vCPU threads can be partitioned between the physical CPUs by using some bin-packing algorithm, and by enforcing that the sum of the utilizations $\sum_k \frac{Q_k}{P_k}$ on each physical CPU is smaller than 1[5].

[5] Actually, a threshold smaller than 1 has to be chosen - the default is 0.95 - to leave some unused execution time for `SCHED_OTHER` tasks on the host OS.

5.2. Taking non-real-time tasks into account

Notice that the previously-described design is based on the "potentially pessimistic" worst-case supply bound function $\mathsf{sbf}(t)$ from Equation (3), which comes with an allocation delay $\Delta_k = 2(P_k - Q_k)$. This expression of $\mathsf{sbf}(t)$ results from the assumption that the worst-case task arrival pattern shown in Figure 2 can really happen. However, some authors showed that this assumption might be too pessimistic [39, 40]. Basically, previous work showed that, for some sets of tasks, the mentioned worst-case arrival pattern cannot happen, with the result that a value of Δ_k smaller than $2(P_k - Q_k)$ could be computed and used to obtain a less-pessimistic analysis. However, this result is based on the assumption that the VM contains only real-time tasks (in other words, real-time tasks are the only tasks that can consume the budget reserved for the virtual CPU). In a real kvm-based VM, the reservation budget can be consumed by both the real-time (fixed priority) and non-real-time (`SCHED_OTHER`) tasks. Hence, a job of a real-time task can arrive immediately after the budget of a reservation has been fully discharged by non-real-time tasks (as shown in Figure 2), even if this situation does not look possible by looking at real-time tasks only. As a result, considering $\Delta_k = 2(P_k - Q_k)$ is not too pessimistic. This is something that has not been explicitly considered in previous theoretical research about hierarchical real-time scheduling but can happen in practical implementations as it will be shown in Section 6.2.

To use a more optimistic $\mathsf{sbf}(t)$ (as described in the papers mentioned above), the budget of a reservation used to schedule a vCPU thread must be decreased only when a real-time task is scheduled on the corresponding virtual CPU. Similarly to the usage of global scheduling in the guest, this requires to use some para-virtualized scheduling [41] or to use some OS-level virtualization [40].

6. Experiments

The effectiveness of the hierarchical scheduling system described in Section 5 has been tested through an extensive set of experiments, as reported in the

following.

6.1. Experimental setup

In each experiment, a set of periodic real-time tasks $\tau_i \in \mathcal{C}$ has been executed in a VM (remember that in the proposed approach each component $\mathcal{C}$ has a dedicated VM). The periodic real-time tasks have then been implemented by using the `periodic_thread` application[6], which creates a set of periodic threads having the desired periods and execution times. The periodic behavior is implemented by using the `clock_nanosleep()` function, and the job body of each task is implemented as a busy loop that consumes the desired amount of time; hence, the periodic tasks are mainly CPU-intensive and the execution times of the jobs of each task are almost constant.

The set of real-time tasks τ_i running in component $\mathcal{C}$ has been randomly generated using the Randfixedsum algorithm [42] and the tasks have been assumed to have relative deadlines equal to the periods (*implicit deadlines*); this means that to respect the component's temporal constraints, each job must be completed before the time at which the next job of the task is released. The set of programs and scripts used for the experiments can be downloaded from `http://retis.santannapisa.it/~luca/RTVM`.

Each `periodic_thread` application is executed inside a VM, i.e., it is a thread executed by a guest. The `SCHED_FIFO` scheduling policy is adopted within guests to schedule the various threads with fixed priorities, which are assigned according to Rate Monotonic [43]. In order to check that the temporal constraints are not violated, the *lateness* $f_{i,h} - d_{i,h}$ of all the jobs of all the tasks have been measured[7]. To aggregate the results from different real-time tasks executing in the same VM, the lateness value can be normalized to the task period, so obtaining the value $\frac{f_{i,h}-d_{i,h}}{P_i}$ that will be referred to as *normal-*

[6] See `https://gitlab.retis.santannapisa.it/l.abeni/PeriodicTask`

[7] this measure is useful in order to check if the temporal constraints have been respected, because a positive value of the lateness indicates that the finishing time of the job is larger than the absolute deadline and hence corresponds to a deadline miss.

ized lateness in the following. Hence, the real-time performance of the tasks $\tau_i \in \mathcal{C}$ can then be expressed by the Cumulative Distribution Function (CDF) $cdf(l) = Pr\left\{\frac{(f_{i,h}-d_{i,h})}{P_i} \leq l\right\}$ for the normalized lateness of their jobs. For example, $cdf(0)$ indicates the probability to have no deadline misses. Another metric that is used to evaluate the following experimental results is introduced: the *reservation cost* of a component is defined as the difference between the sum of all the utilizations of the m reservation servers that are used to implement the virtual CPUs of the component, and the corresponding task set utilization. This cost is formally defined as $\sum_{k=1}^{m} Q_k/P_k - \sum_{\tau_i \in \mathcal{C}} C_i/T_i$ and models the extra CPU bandwidth required by the component interface.

The set of experiments reported in this section shows how the measured response times match with theoretical analysis results. These experiments have been performed by using various versions of the Linux kernel (4.4.0-112 from Ubuntu, 4.13.8-100.fc25 from Fedora, 4.13.0-16 from Ubuntu, 4.13.16 compiled from source) and different Linux distributions (Ubuntu 18.04 LTS, Ubuntu 17.10, Fedora 25, Ubuntu 16.04.3 LTS, and Linux Mint 18.3). Multiple CPU models (including Intel(R) Core(TM) i5-5200U, Intel(R) Core(TM) i7-4790K, Intel(R) Xeon(R) CPU E5-2640 and AMD Ryzen Threadripper 1950X) have been tested obtaining consistent results, with the only exception of a few runs on the Core i5 that showed some unexpected overhead; see Section 7 for more details. As previously explained, qemu/kvm has been used for the VMs and the vCPU threads have been scheduled by the host scheduler using the `SCHED_DEADLINE` policy. For CPUs having more than four cores, the vCPU threads have been scheduled on four cores only (using both the global EDF algorithm provided by `SCHED_DEADLINE` and the partitioned EDF obtained by explicitly setting the vCPU threads affinities), using the isolated cpusets mechanism.

To obtain repeatable results and to impose a more precise execution time, the CPU frequency has been fixed by setting a constant power state of the Intel `pstate` driver.

6.2. Reservation design in the presence of non-real-time tasks

A first set of experiments has been carried out to investigate the design of the reservation parameters for VMs with a single virtual CPU by using Algorithm 1. Note that the algorithm assumes an allocation delay $\Delta = 2(P - Q)$ that, as previously explained in Section 5, is a tight bound on the service delay suffered by real-time tasks when non-real-time tasks (`SCHED_OTHER`) can consume the server budget. To experimentally confirm this observation, a periodic task τ with period $50ms$ and execution time $25ms$ has been executed inside a single-CPU kvm-based VM. The corresponding vCPU thread has been scheduled by `SCHED_DEADLINE` using budget $Q = 30ms$ and period $P = 50ms$. Since τ is the only real-time task inside the VM, there are no other real-time tasks that can consume the server budget before the release of τ's jobs (in theory, this is not even an example of hierarchical scheduling). The schedulability of tasks running upon reservation servers offered by `SCHED_DEADLINE` can be checked with Lemma 1 of [2], which states that a periodic real-time task (C, T) does not miss any deadline when scheduled by a reservation (Q, P) if $Q \geq C$ and $P \leq T$. In this case, τ results schedulable. Since τ is the only task running upon the server (also denoted as a "bound task" according to [39]), it incurs in a worst-case service delay equal to $P - Q < \Delta$.

This analysis seems to be confirmed by a first experiment, in which the task is scheduled by the `SCHED_FIFO` policy in the guest without a significant non real-time workload, and no deadline miss has been observed. However, when the task has been scheduled as `SCHED_FIFO` in the guest together with some CPU-consuming tasks scheduled by the `SCHED_OTHER` policy, i.e., non-real-time tasks, about 30% of its jobs miss their deadline. This happens because the `SCHED_OTHER` tasks can consume the budget reserved to the vCPU thread, leading to the worst-case situation of Figure 2, which is not considered in Lemma 1 of [2].

Repeating the experiment with a CPU reservation designed with Algorithm 1 (that is, assuming a worst-case delay $\Delta = 2(P - Q)$, which leads to reserving a budget $Q = 37.5$ when $P = 50$), all the jobs respected their deadlines even in

Task	C_i	T_i
τ_1	$7.284ms$	$55ms$
τ_2	$4.799ms$	$66ms$
τ_3	$23.150ms$	$213ms$
τ_4	$24.938ms$	$451ms$
τ_5	$5.898ms$	$191ms$

Table 3: Example of task set generated by Randfixedsum, with five tasks and $\sum_{\tau_i \in \mathcal{C}} \frac{C_i}{T_i} = 0.4$

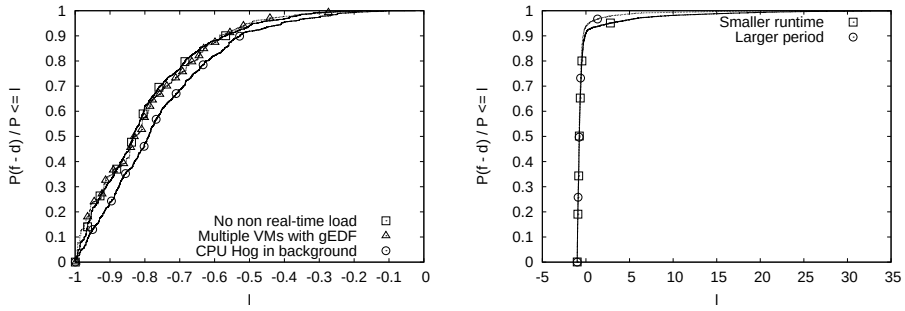
the presence of several CPU-consuming `SCHED_OTHER` tasks in the guest.

6.3. Single-CPU VMs — Design of the reservation parameters

Another set of experiments has been carried out by randomly-generating (with the Randfixedsum algorithm) various task sets with utilization ranging from 0.3 to 0.85, periods in the range $[20ms, 500ms]$ (with uniform distribution), and composed of $N \in [4, 10]$ tasks. Each of the generated task sets has been used to populate a component executed in a kvm-based VM with a single virtual CPU. The corresponding vCPU thread has been scheduled by `SCHED_DEADLINE`. The experiments have been performed by running a single VM or multiple VMs (multiple concurrent components) upon the same host system.

The server design has been performed by adding two additional constraints: the budget Q must be larger or equal than $1ms$, and the reservation period P must be in the interval $[10ms, 500ms]$. As an example, Table 3 shows one of the generated task sets, for which the design algorithm produced a $(Q = 7ms, P = 16ms)$ reservation. Note that the utilization of this task sets is $U = \sum_{\tau_i \in \mathcal{C}} \frac{C_i}{T_i} = 0.4$, while the fraction of CPU-time reserved to the vCPU thread is $Q/P = 7/16 = 0.4375$; hence, the reservation cost for scheduling the container in a VM is $0.0375 = 3.75\%$.

For each pair (U, N), 100 different sets of tasks have been generated and the normalized lateness of every job of every task has been measured and used to estimate a CDF. As an example, Figure 3 shows the experimental CDF of



(a) (b)

Figure 3: CDF of the normalized lateness ($Pr\left\{\frac{(f_{i,h}-d_{i,h})}{P_i} \leq l\right\}$). The figure on the left (a) refers to the case of a component scheduled upon a single virtual CPU served by a correctly-designed reservation (Q_k and P_k computed as discussed in Section 4.3). Notice that the probability of missing a deadline ($\frac{(f_{i,h}-d_{i,h})}{P_i} > 0$) is 0. On the right (b), the results for a component scheduled on one single virtual CPU served by badly-designed reservations. Notice that in this case there is a non-null probability of missing a deadline ($Pr\left\{\frac{(f_{i,h}-d_{i,h})}{P_i} > 0\right\} > 0$).

the normalized lateness measured with $N = 5$ tasks per component $\mathcal{C}$, and an utilization $U = \sum_{\tau_i \in \mathcal{C}} \frac{C_i}{T_i} = 0.4$ for each component[8]. The results presented in this figure have been measured both when there is only one VM running on the host (i.e., the case in which all the 100 generated components for the pair $(U = 0.4, N = 5)$ have been sequentially executed) and when multiple VMs have been concurrently executed on the host by means of global EDF scheduling (in general, 5 or 6 different components were admitted to run concurrently on the 4 CPU cores). Each experiment has been performed two times: first by running only the real-time tasks in an otherwise idle VM, and then by running a heavy CPU load in background (as `SCHED_OTHER` non-real-time tasks). The second setup makes it more likely to have a job for a real-time task arriving when the budget is exhausted.

Figure 3(a) shows some results obtained when the VM is scheduled with

[8]Experiments with a different number of tasks per component and different utilizations have been performed, leading to similar results.

parameters designed with Algorithm 1. Since the CDF of the normalized lateness arrives at 1 before the normalized lateness arrives to 0 (in other words, $Pr\left\{\frac{(f_{i,h}-d_{i,h})}{P_i} \leq 0\right\} = 1$), all the deadlines are respected, thus consistently matching the theory.

To double-check the correctness of the analysis, the experiments have been repeated by using a smaller budget ($1ms$ smaller than the "correct budget" used in Fig. 3(a)) or a larger period ($1ms$ larger than the "correct period" used in Fig. 3(a)); the results are reported in Figure 3(b) and show that, in this case, a noticeable number of deadlines are missed—i.e., there is a significant probability of measuring a normalized lateness larger than 0.

The experiments have been then repeated by running multiple VMs in parallel on the same host. When running multiple VMs to serve a set of components $\{\mathcal{C}_j\}$, it is crucial to guarantee that the host scheduler is able to respect the (Q_j, P_j) interface provided to each vCPU. This can be done by running an appropriate admission test. If the host scheduler is partitioned, then, after statically assigning virtual CPUs to physical CPUs, a standard single-processor admission test can be used. Otherwise, if global scheduling is used in the host, then a more advanced (and pessimistic [17]) admission test must be used [37, 38]. The experiments have been performed by using $M = 4$ cores in the host system, and the VMs have been scheduled by using a global EDF scheduler (and running the admission test [37, 38] in user space) or a partitioned scheduler (explicitly setting the affinity of each vCPU thread to one single CPU) in the host. The admission test and the number of admitted VMs changed depending on the usage of a global or partitioned scheduler, but the normalized lateness results were similar.

In all the cases, when the reservations were properly designed with Algorithm 1, no missed deadline has been detected, thus confirming that the proposed approach allows to properly serve real-time tasks in VM guests, even when multiple VMs are executed simultaneously. This also indicates that both partitioned or global scheduling can be used at the host level without compromising the tasks schedulability. As already discussed (and as it will be confirmed by

the next experiments), this is not true for the guest scheduler.

6.4. Verifying the issues with global guest scheduling

As shown in Section 3, VMs with multiple virtual CPUs present some issues that have not generally been considered in the literature. To experimentally verify these issues, the tasks set $\Gamma = \{\tau_1 = (60ms, 100ms), \tau_2 = (10ms, 200ms)\}$ from the example in Section 3 has been executed in a kvm-based VM with 2 virtual CPUs and the 2 vCPU threads scheduled by `SCHED_DEADLINE` with the budgets and periods mentioned in the example ($(90ms, 100ms)$ and $(40ms, 100ms)$).

The two tasks τ_1 and τ_2 are scheduled by the guest kernel using the `SCHED_FIFO` scheduling policy (which uses global scheduling by default on Linux); remember that, as previously discussed, from the theoretical analysis the two tasks should not miss any deadline. However, repeating the experiment 10 times it has been noticed that in some cases the tasks are correctly scheduled (without missing any deadlines), while in other cases the response times for τ_1 continuously increase and all the absolute deadlines of this task are missed. As previously explained, this happens because the guest Linux kernel ends up in scheduling τ_1 on the vCPU with a depleted budget, breaking one of the assumptions made in the theoretical analysis.

This experiment clearly shows that some form of para-virtualized scheduler [41] is required to use global scheduling in the guest. The next experiments will focus on partitioned scheduling in the guest as proposed in this paper.

6.5. Multi-CPU VMs

In the next experiments, we tested several task sets generated by the Randfixedsum algorithm with utilization $U \in [1.0, 3.0]$, composed of $N \in [5, 30]$ tasks, and periods in the range $[10ms, 500ms]$ (with uniform distribution). Each task set (representing a component) has been scheduled in a VM with 4 virtual CPUs. The tasks have been statically partitioned upon the 4 virtual CPUs by solving the MILP formulation described in Equation (7), where one the definitions proposed by Park and Park (specifically, the one of Theorem 5 in [22])

Task	C_i	T_i
τ_1	$5.022ms$	$26ms$
τ_2	$13.262ms$	$93ms$
τ_3	$11.446ms$	$121ms$
τ_4	$36.846ms$	$122ms$
τ_5	$10.319ms$	$145ms$
τ_6	$5.219ms$	$181ms$
τ_7	$23.142ms$	$302ms$
τ_8	$96.506ms$	$354ms$
τ_9	$60.679ms$	$437ms$
τ_{10}	$187.492ms$	$494ms$

Table 4: Example of task set generated by Randfixedsum, with 10 tasks and $U = 1.7$

has been used to build the sets tSet_i. Each vCPU thread of the VM has been scheduled using `SCHED_DEADLINE` with the reservation parameters designed by means of Algorithm 1, enforcing a minimum budget of $1ms$ and a minimum period of $10ms$. Furthermore, the budget and the periods have been rounded to multiples of $500\mu s$ and $1ms$, respectively.

As an example, consider one of the generated task sets with utilization $U = 1.7$ and $N = 10$ tasks as reported in Table 4. Using the first optimization goal from Equation (7) (objective A: minimize $\sum_k \frac{Q_k}{P_k}$), the partitioning algorithm produces the task subsets $\Gamma_0 = \{\tau_0, \tau_9\}$ (tasks τ_0 and τ_9 assigned to the first virtual CPU), $\Gamma_1 = \{\tau_3, \tau_4, \tau_6, \tau_8\}$, $\Gamma_2 = \{\tau_2, \tau_{10}\}$, and $\Gamma_3 = \{\tau_5, \tau_7\}$. The first vCPU thread is scheduled with parameters $(6ms, 16ms)$, the second vCPU thread is scheduled with parameters $(16ms, 22ms)$, the third vCPU thread is scheduled with parameters $(13.5ms, 24ms)$, and the fourth vCPU thread is scheduled with parameters $(4ms, 24ms)$, with a total utilization of $\frac{6}{16} + \frac{16}{22} + \frac{13.5}{24} + \frac{4}{24} = 1.8314$. The corresponding reservation cost is hence $1.8314 - 1.7 = 0.1314$. For reference, with MPR the minimum CPU bandwidth to schedule component is 2.72, with a reservation cost of about 1, i.e.,

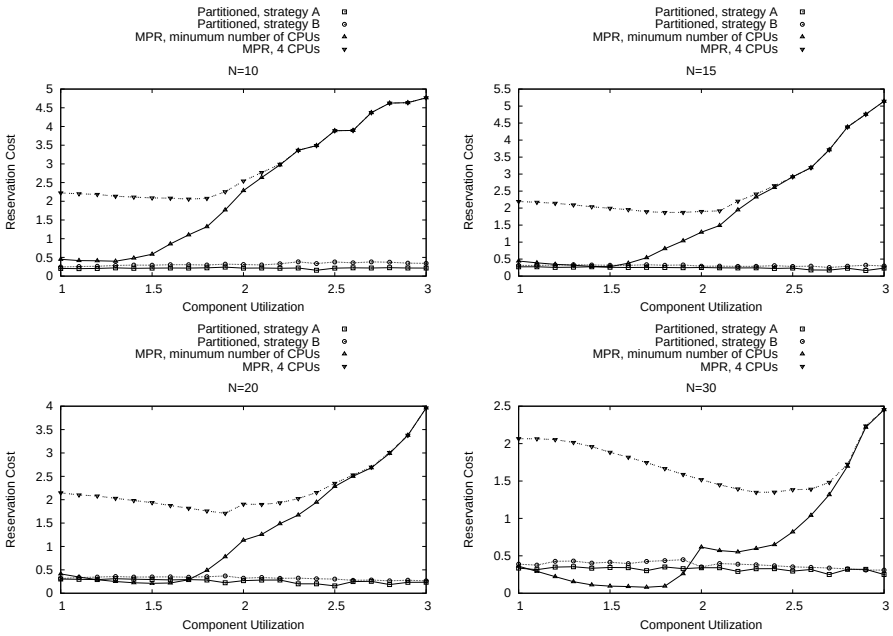


Figure 4: Reservation cost for executing components in VMs for MPR and the proposed partitioning approach, as a function of the component utilization.

the bandwidth of an entire CPU.

Figure 4 compares the reservation cost for executing a component in a VM when using the proposed partitioned approach (with optimization goals A or B) and MPR. For MPR, the design has been performed both using the minimum number of virtual CPUs and distributing the component over 4 virtual CPUs or more; the second design (4 virtual CPUs or more) has been performed in order to decrease the CPU bandwidth allocated on each physical CPU (so that some CPU time can be left for host system software as explained in Section 1). Unfortunately, as the figures show this approach does not work well since increasing the number of virtual CPUs the allocated bandwidth also increases (as already noticed in the example at the end of Section 3.2 — these experiments just confirm that the previous example is not just a pathological case).

The reservation cost has been computed for task sets with different numbers of tasks ($N \in \{10, 15, 20, 30\}$) and different utilizations (the plots show the

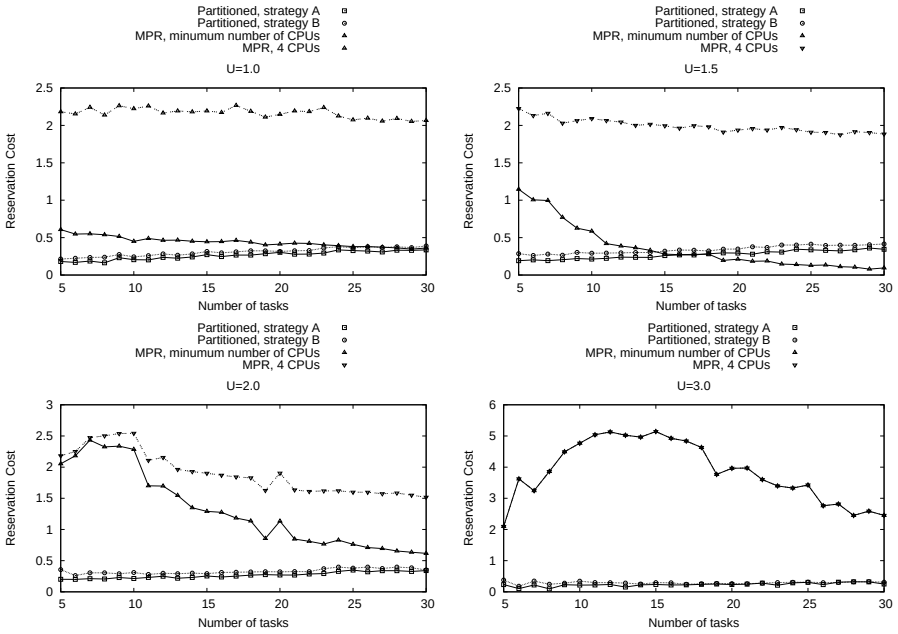


Figure 5: Reservation cost for executing components in VMs for MPR and the proposed partitioning approach, as a function of the number of tasks.

cost as a function of the utilization). For each different configuration (a pair of number of tasks and utilization), 100 different task sets have been analyzed, and the figure reports the average of the collected reservation costs. From the figure, it is possible to see that MPR with the minimum number of virtual CPUs tends to perform better when the component's tasks have a small utilization (low values of the total component's utilization, and a high number of tasks), while its cost tends to increase when the component's utilization increases. With 30 tasks and a utilization smaller than 1.9, MPR with 2 CPUs performs better than the proposed partitioning approach. This is probably due to the behavior of global scheduling. On the other hand, MPR on 4 or more virtual CPUs is always outperformed by the proposed approach. The partitioning approach shows a cost that does not depend on the component's utilization, and is almost independent of the number of tasks.

Figure 5 reports the same results (for utilizations $U \in \{1.0, 1.5, 2.0, 2.5\}$) as a function of the number of tasks in the component. The figure basically confirms the previous observations, but also highlights a strange behavior of MPR for $N = 20$ tasks and utilization $U = 2.0$. Further investigation revealed that this behavior is due to the fact that one of the 100 task sets generated for $N = 20$ and $U = 2.0$ exhibits the issue discussed at the end of Section 3, requiring a CPU bandwidth equal to 14.9 (much larger than the task set utilization $U = 2.0$).

Note that all the generated components have also been executed under the considered kvm-based experimental setup and by following the approach proposed in this paper. Since components with different numbers of tasks and different utilizations generated similar results, we report, in Figure 6, the experimental CDF of the normalized lateness only for two representative configurations: components with $N = 10$ tasks and utilization $U = 1.2$; and components with $N = 21$ tasks and utilization $U = 3.0$. As in the single-vCPU experiments, the figure has been generated by collecting data from 100 randomly-generated task sets for each configuration. Again, it is possible to appreciate how the normalized lateness is never larger than 0, so no deadline has been missed.

For the case of components with 10 tasks and utilization 1.2, the average of

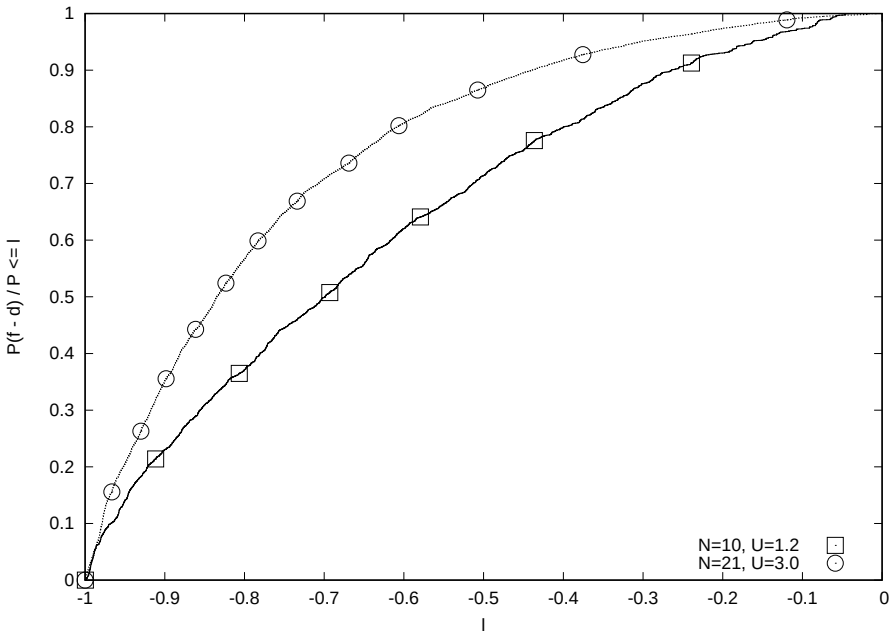


Figure 6: CDF of the normalized lateness ($Pr\left\{\frac{(f_{i,h}-d_{i,h})}{P_i} \leq l\right\}$) for a component scheduled on multiple virtual CPUs, using a partitioned guest scheduler as discussed in Section 4. Notice that all the deadlines are respected ($\frac{(f_{i,h}-d_{i,h})}{P_i} > 0$).

l	$P\{L \geq l\}$	
	without hrtick	with hrtick
-0.053294	0.9995	0.9888
-0.053088	0.9996	0.9902
-0.051792	0.9996	0.9998
-0.051745	0.9996	0.9999
-0.051676	0.9996	1.0000
0.004794	0.9997	1.0000
0.004853	0.9998	1.0000
0.005029	0.9999	1.0000
0.005412	1.0000	1.0000

Table 5: Tails of the normalized lateness CDFs (values of the CDFs for $l > 0.053$) when the `hrtick` kernel feature is disabled (without hrtick) or enabled (with hrtick). When `hrtick` is not enabled, some deadlines are missed (there is a non null probability to have $l > 0$).

the sums of the bandwidths reserved to the 4 vCPU threads resulted equal to 1.464. Repeating the design with MPR, it turned out that the average CPU bandwidth to be reserved to the VM is 1.609 when using 2 vCPUs, and 3.385 when using 4 vCPUs. For the other configuration with 21 tasks and utilization 3.0, the partitioning approach needed to reserve an average CPU bandwidth of 3.26 to the vCPU threads, while MPR would have needed an average bandwidth of 6.97 (at least 7 physical CPUs).

It is worth noticing that, in these experiments, the partitioning and design algorithm proposed in this paper splits the tasks on 4 different vCPUs, and for some sets, the utilization on some vCPUs was quite small. As a result, the corresponding reservations were designed with a small budget (around 1 or 2 milliseconds). Since the Linux kernel accounts for the budget at every periodic tick, a vCPU can experience an overrun up to one tick, which is typically comparable with the budget in the unfortunate cases mentioned above. As a result, with the default kernel configuration, some task sets were experiencing sporadic deadline misses: the CDF of the normalized lateness was similar to the

one represented in Figure 6, but the maximum normalized tardiness value was larger than 0, indicating some missed deadline. In more details, the maximum experienced value for the normalized lateness was 0.005412, and a normalized lateness larger than 0 was measured with probability $4 \cdot 10^{-4}$. This issue has been fixed by enabling a kernel mechanism called `hrtick`, that forces the kernel to perform exact accounting and enforcement of the budget by using additional one-shot timers instead of the periodic tick timer. After enabling `hrtick`, no missed deadline has been experienced, as it is visible in Table 5 that shows the tails of the normalized lateness CDFs when `hrtick` is enabled and when it is not.

7. Additional Considerations

The previous discussion mainly focused on scheduling algorithms, assuming that the host scheduler and the guest scheduler can precisely implement them. However, due to implementation issues in the OS kernel[9], a real scheduler often introduces some additional delays with respect to the theoretical CPU allocation. These delays are generally known as *kernel latencies* [44].

Due to the Linux kernel latency, some non-real-time workload is able to generate unexpected deadline misses. For example, threads or processes scheduled by `SCHED_OTHER` in the host can cause deadline misses in some guests (by triggering high kernel latencies in the host kernel) even if an appropriate amount of CPU time is reserved for vCPU threads according to the theoretical analysis. In a standard Linux kernel, the kernel latency can be larger than $10ms$ (and is technically equivalent to a *blocking time* for the CPU reservations).

This issue can be addressed by using a real-time version of the Linux kernel, such as Preempt-RT [45, 46]. It has been previously shown that using Preempt-RT, and properly configuring the host and the guest, it is possible to achieve a $20\mu s$ worst-case kernel latency [47], achieving a CPU allocation very similar to

[9] Or in the hypervisor, in case of bare metal hypervisors.

the theoretical one. While we plan to perform more tests with real-time features in the kernel and in the hypervisor, some preliminary experiments seem to show that using Preempt-RT in the host is of paramount importance, while using it in the guest is generally less critical.

Another source of unexpected missed deadlines came from some hardware issues related to the Intel Core i5 CPU we tested. All the other CPUs we tested allowed to achieve a perfect consistency between experimental results and theoretical analysis, but some of the experiments performed on the i5 CPU resulted in response times larger than expected. In particular, some of the task sets scheduled on VMs with multiple virtual CPUs (Figure 6) sporadically missed some deadlines. Since this issue only happened with a specific CPU model and did not depend on the kernel version and configuration, we suspect that it is due to the hardware design of the CPU (e.g., related to some hardware resources that are shared between different cores), but some investigation is still needed. For example, it is possible to check if the issue is due to caches shared between CPU cores by repeating the experiments with cache coloring techniques [48, 49].

From a more theoretical point of view, an aspect that deserves to be investigated in more details is the usage of global scheduling in the host. In our previous experiments, we used a hard real-time admission test that is quite pessimistic [17, 37, 38]. A less pessimistic admission control can be used, but it does not guarantee the hard respect of every deadline; it only guarantees an upper bound on the tardiness [50]. Hence, Algorithm 1 (server design) must be improved to account for such a tardiness. In particular, the worst-case allocation delay Δ in the `sbf` must be accordingly increased so that the reservation parameters Q and P can be properly computed in order to avoid breaking the guarantees in the guest. A possible way to do this can be based on the work by Erickson et al. [51], who proposed a linear programming approach to assign deadlines to global EDF tasks such that their tardiness is minimized, reducing Δ.

We are also currently analyzing the usage of a global scheduling algorithm

in the guest, either using a para-virtualized scheduler or scheduling all the vCPU threads with the same budget and period. In this regard, an interesting line of research that deserves to be investigated is about OS-level virtualization (container-based virtualization on Linux). In fact, scheduling para-virtualization can be easily implemented by modifying the so-called "real-time control group scheduler" in the Linux kernel, which is used in container-based VMs such as lxc[10] or Docker[11]. At this point, some preliminary work on this direction has been performed using lxc [52], but can be easily extended to Docker and similar tools.

Finally, all the experiments presented in this paper focused on CPU-intensive tasks and did not consider system calls, interactions between different components (or between tasks in a single component), I/O or access to (virtual) devices. When the workload running in the guest performs system calls, it generates some additional latencies that must be accounted for in the schedulability analysis. The usage of different technologies for implementing virtual devices (and for connecting them with physical devices or other virtual devices) can have a huge impact on the performance, both in terms of throughput [53] and latencies [54]. Some investigation has already been performed in literature, but the results still have to be integrated into schedulability analysis (and in the reservation design algorithm). Interaction between tasks (belonging to the same component or to different components) introduce additional blocking times that, again, have to be considered in the analysis; from the practical point of view, interactions between components can be implemented by using the `ivshmem` virtual device [55] provided by qemu, or by using the `vhost-user`/`virtio-user` [56, 57] functionality.

All the blocking times mentioned above (due to latencies or to interactions between tasks or components) can be considered in the design of the vCPU reservations by adding some constraints to Equation 7 (and Algorithm 1), sim-

[10] See `https://linuxcontainers.org/`.
[11] See `https://www.docker.com/`.

ilarly to what has been done in [23].

8. Conclusions

This paper presented the implementation of a theoretically-sound two-level hierarchical scheduling system based on Linux and kvm. First, some implementation-related issues with previous analysis from literature have been highlighted, and then a solution based on kvm and `SCHED_DEADLINE` (using partitioned scheduling in the guest) has been presented.

Experiments on real hardware showed both that (i) the issues discussed in the paper can really happen in practice, as they can be easily reproduced in practical configurations; and (ii) that the presented solution is sound, as the experimental results matched with the theoretical analysis used in this paper.

As a future work, we plan to investigate the points raised in Section 7 (see that section for details) and to analyze how the proposed technique can be applied in real-world clouds (considering multiple hosts) and in industrial environments.

Acknowledgement

This work has been partially funded by the European Commission through the RETINA (EUROSTARS E10171) project.

[1] G. Banga, P. Druschel, J. C. Mogul, Resource containers: A new facility for resource management in server systems, in: OSDI, Vol. 99, 1999, pp. 45–58.

[2] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proc. of 19th IEEE Real-Time Systems Symposium, 1998, pp. 4–13.

[3] H. Leontyev, J. H. Anderson, A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees, Real-Time Systems 43 (1) (2009) 60–92.

[4] E. Bini, M. Bertogna, S. Baruah, Virtual multiprocessor platforms: Specification and use, in: Proc. of 30th IEEE Real-Time Systems Symposium, 2009, pp. 437–446.

[5] A. Easwaran, I. Shin, I. Lee, Optimal virtual cluster-based multiprocessor scheduling, Real-Time Systems 43 (1) (2009) 25–59.

[6] G. Lipari, E. Bini, A framework for hierarchical scheduling on multiprocessors: from application requirements to run-time allocation, in: Proc. of 31st IEEE Real-Time Systems Symposium, 2010, pp. 249–258.

[7] N. M. Khalilzad, M. Behnam, T. Nolte, Multi-level adaptive hierarchical scheduling framework for composing real-time systems, in: Proc. of 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013, pp. 320–329.

[8] A. Burmyakov, E. Bini, E. Tovar, Compositional multiprocessor scheduling: the GMPR interface, Real-Time Systems 50 (3) (2014) 342–376.

[9] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzén, V. Romero-Segovia, C. Scordino, Resource management on multicore systems: The ACTORS approach, IEEE Micro 31 (3) (2011) 72–81.

[10] K. Yang, J. H. Anderson, On the dominance of minimum-parallelism multiprocessor supply, in: Proc. of 37th IEEE Real-Time Systems Symposium, 2016, pp. 215–226.

[11] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, O. Sokolsky, Realizing compositional scheduling through virtualization, in: Proc. of 18th IEEE Real Time and Embedded Technology and Applications Symposium, 2012, pp. 13–22.

[12] A. K. Mok, X. Feng, D. Chen, Resource partition for real-time systems, in: Proc. of 7th IEEE Real-Time Technology and Applications Symposium, 2001, pp. 75–84.

[13] X. Feng, A. K. Mok, A model of hierarchical real-time virtual resources, in: Proc. of 23rd IEEE Real-Time Systems Symposium, 2002, pp. 26–35.

[14] G. Lipari, E. Bini, Resource partitioning among real-time applications, in: Proc. of 15th Euromicro Conference on Real-Time Systems, 2003, pp. 151–158.

[15] I. Shin, I. Lee, Periodic resource model for compositional real-time guarantees, in: Proceedings of 24th IEEE Real-Time Systems Symposium, 2003, pp. 2–13.

[16] L. Almeida, P. Pedreiras, Scheduling within temporal partitions: response-time analysis and server design, in: Proc. of 4th ACM International Conference on Embedded Software, 2004, pp. 95–103.

[17] A. Biondi, Y. Sun, On the ineffectiveness of 1/m-based interference bounds in the analysis of global edf and fifo scheduling, Real-Time Systems.

[18] L. T. X. Phan, J. Lee, A. Easwaran, V. Ramaswamy, S. Chen, I. Lee, O. Sokolsky, CARTS: A tool for compositional analysis of real-time systems, SIGBED Review 8 (1) (2011) 62–63.

[19] T. Cucinotta, G. Anastasi, L. Abeni, Respecting temporal constraints in virtualised services, in: Proc. of 33rd IEEE International Computer Software and Applications Conference, 2009, pp. 73–78.

[20] J. P. Lehoczky, L. Sha, Y. Ding, The rate-monotonic scheduling algorithm: Exact characterization and average case behavior, in: Proc. of 10th IEEE Real-Time Systems Symposium, 1989, pp. 166–171.

[21] E. Bini, G. C. Buttazzo, Schedulability analysis of periodic fixed priority systems, IEEE Transactions on Computers 53 (11) (2004) 1462–1473.

[22] M. Park, H. Park, An efficient test method for rate monotonic schedulability, IEEE Transactions on Computers 63 (5) (2014) 1309–1315.

[23] A. Biondi, G. Buttazzo, M. Bertogna, Partitioning and interface synthesis in hierarchical multiprocessor real-time systems, in: Proc. of 24th International Conference on Real-Time Networks and Systems, 2016, pp. 257–266.

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, SIGOPS operating systems review 37 (5) (2003) 164–177.

[25] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: the linux virtual machine monitor, in: Proceedings of the Linux symposium, Vol. 1, 2007, pp. 225–230.

[26] G. J. Popek, R. P. Goldberg, Formal requirements for virtualizable third generation architectures, Communications of the ACM 17 (7) (1974) 412–421.

[27] J. K. Strosnider, J. P. Lehoczky, L. Sha, The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments, IEEE Transactions on Computers 44 (1) (1995) 73–91.

[28] T. M. Ghazalie, T. P. Baker, Aperiodic servers in a deadline scheduling environment, Real-Time Systems 9 (1) (1995) 31–67.

[29] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, I. Lee, Real-time multi-core virtual machine scheduling in Xen, in: Proc. of 2014 International Conference on Embedded Software (EMSOFT), 2014, pp. 1–10.

[30] M. Vanga, A. Gujarati, B. B. Brandenburg, Tableau: A high-throughput and predictable vm scheduler for high-density workloads, in: Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, ACM, New York, NY, USA, 2018, pp. 28:1–28:16.

[31] J. Yang, H. Kim, S. Park, C. Hong, I. Shin, Implementation of compositional scheduling framework on virtualization, SIGBED Rev. 8 (1) (2011) 30–37.

[32] M. Hohmuth, H. Härtig, Pragmatic nonblocking synchronization for real-time systems, in: Proc. of USENIX Annual Technical Conference, 2001, pp. 217–230.

[33] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, S. Schönberg, The performance of μ-kernel-based systems, in: Proc. of 16th ACM Symposium on Operating Systems Principles, 1997, pp. 66–77.

[34] T. Cucinotta, D. Giani, D. Faggioli, F. Checconi, Providing performance guarantees to virtual machines using real-time scheduling, in: European Conference on Parallel Processing, 2010, pp. 657–664.

[35] F. Bellard, QEMU, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41–46.

[36] J. Lelli, C. Scordino, L. Abeni, D. Faggioli, Deadline scheduling in the linux kernel, Software: Practice and Experience 46 (6) (2016) 821–839.

[37] M. Bertogna, M. Cirinei, G. Lipari, Improved schedulability analysis of EDF on multiprocessor platforms, in: Proc. of 17th Euromicro Conference on Real-Time Systems, 2005, pp. 209–218.

[38] J. Goossens, S. Funk, S. Baruah, Priority-driven scheduling of periodic task systems on multiprocessors, Real-Time Systems 25 (2) (2003) 187–205.

[39] R. I. Davis, A. Burns, Hierarchical fixed priority pre-emptive scheduling, in: Proc. of 26th IEEE Real-Time Systems Symposium, 2005, pp. 389–398.

[40] L. Abeni, T. Cucinotta, Efficient virtualisation of real-time activities, in: Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2011), Irvine, CA, 2011, pp. 1–4.

[41] J. Kiszka, Towards linux as a real-time hypervisor, in: Eleventh Real-Time Linux Workshop, Dresden, Germany, 2009, p. 205.

[42] P. Emberson, R. Stafford, R. I. Davis, Techniques for the synthesis of multiprocessor tasksets, in: Proceedings 1st International Workshop on Analysis

Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010), 2010, pp. 6–11.

[43] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the Association for Computing Machinery 20 (1) (1973) 46–61.

[44] L. Abeni, A. Goel, C. Krasic, J. Snow, J. Walpole, A measurement-based analysis of the real-time performance of linux, in: Proc. of 8th IEEE Real-Time and Embedded Technology and Applications Symposium, 2002, pp. 133–142.

[45] S. Rostedt, Internals of the RT patch, in: Proceedings of the Linux Symposium, Ottawa, Canada, 2007, pp. 161–172.

[46] P. McKenney, A realtime preemption overview, `https://lwn.net/Articles/146861/` (August 2005).

[47] R. V. Riel, Real-time kvm from the ground up, in: KVM Forum 2015, 2015.

[48] P. Modica, A. Biondi, G. Buttazzo, A. Patel, Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms, in: Proceedings of the 18th IEEE International Conference on Industrial Technology (ICIT 2018), 2018.

[49] H. Kim, R. Rajkumar, Predictable shared cache management for multi-core real-time virtualization, ACM Transactions on Embededded Computing Systems 17 (1) (2017) 22:1–22:27.

[50] U. C. Devi, J. H. Anderson, Tardiness bounds under global EDF scheduling on a multiprocessor, Real-Time Systems 38 (2008) 133–189.

[51] J. P. Erickson, J. H. Anderson, B. C. Ward, Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling, Real-Time Systems 50 (1) (2014) 5–47.

[52] L. Abeni, A. Balsini, T. Cucinotta, Container-based real-time scheduling in the linux kernel, in: Proceedings of the Embedded Operating System Workshop 2018 (EWiLi'18), Torino, Italy, 2018.

[53] L. Abeni, C. Kiraly, N. Li, A. Bianco, On the performance of kvm-based virtual routers, Computer Communications 70 (Supplement C) (2015) 40–53.

[54] C. Li, S. Xi, C. Lu, C. D. Gill, R. Guerin, Prioritizing soft real-time network traffic in virtualized hosts based on xen, in: Proc. of 21st IEEE Real-Time and Embedded Technology and Applications Symposium, 2015, pp. 145–156.

[55] A. C. Macdonell, Shared-memory optimizations for virtual machines, Ph.D. thesis, University of Alberta (2011).

[56] J. Tan, C. Liang, H. Xie, Q. Xu, J. Hu, H. Zhu, Y. Liu, Virtio-user: A new versatile channel for kernel-bypass networks, in: Proceedings of the Workshop on Kernel-Bypass Networks, KBNets '17, ACM, New York, NY, USA, 2017, pp. 13–18.

[57] M. Paolino, N. Nikolaev, J. Fanguede, D. Raho, Snabbswitch user space virtual switch benchmark and performance optimization for NFV, in: 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), 2015, pp. 86–92.

Appendix A. Computing the Level-1 Parallel Supply Function

To guarantee tasks over virtual machines, the most general abstraction was proposed by Bini et al. [4], who proposed the *parallel supply function* (PSF). This notion extends the concept of supply function of Equation (1) to machines on which components may use the computing power provided by any of the available vCPUs. In brief, the PSF abstraction of m vCPUs is a collection of m function $\{\mathsf{psf}_1(t), \ldots, \mathsf{psf}_m(t)\}$. Each function $\mathsf{psf}_k(t)$ represents the minimum

computing capacity made available by any k vCPUs. Formally (Equations (6) and (13) of [4]),

$$\mathsf{psf}_k(t) = \min_{[\xi_1,\ldots,\xi_m]} \min_{t_0} \int_{t_0}^{t_0+t} \min\left\{k, \sum_{j=1}^{m} \xi_j(s)\right\} ds \tag{A.1}$$

with $[\xi_1, \ldots, \xi_m]$ being all possible schedule functions of the vCPUs. Hence, the index k of $\mathsf{psf}_k(t)$ is **not** linked to the index of the k-th vCPU. Rather, it represents the parallelism of the computing power offered by Π. The interested reader may find more details in [4] and [6].

Below, we report an original result linking the set of supply bound functions from $\mathsf{sbf}_1(t)$ to $\mathsf{sbf}_m(t)$ of each vCPU of Π to the parallel supply function $\mathsf{psf}_1(t)$, which represents the amount of computing power provided by at most one vCPU among the m available.

Lemma 1. *Given a virtual machine* Π *composed by* m *vCPUs, with the* k*-th vCPU characterized by the supply function* $\mathsf{sbf}_k(t)$*. Then, the parallel supply function* $\mathsf{psf}_1(t)$ *of parallelism* 1 *of the whole machine* Π *is such that*

$$\forall t \geq 0, \qquad \mathsf{psf}_1(t) \geq \max_k \{\mathsf{sbf}_k(t)\} \tag{A.2}$$

Proof. Let $[\xi_1^*, \ldots, \xi_m^*]$ and t_0^* be the indicator functions of the vCPU schedules and the value of t_0 that determine the minimum for $\mathsf{psf}_1(t)$. Hence from the definition of Eq. (A.1)

$$\mathsf{psf}_1(t) = \int_{t_0^*}^{t_0^*+t} \min\left\{1, \sum_{j=1}^{m} \xi_j^*(s)\right\} ds.$$

Now we observe that it must be

$$\forall t, k, \quad \min\left\{1, \sum_{j=1}^{m} \xi_j^*(t)\right\} \geq \xi_k^*(t).$$

In fact:

- both the LHS and the RHS are either 0 or 1

- if the LHS is 0 then no $\xi_k(t)$ can be 1.

Hence

$$\forall t, k, \quad \mathsf{psf}_1(t) = \int_{t_0^*}^{t_0^*+t} \min\Big\{1, \sum_{j=1}^{m} \xi_j^*(s)\Big\} ds \geq \int_{t_0^*}^{t_0^*+t} \xi_k^*(s) ds \geq \mathsf{sbf}_k(t)$$

by definition of worst-case resource supply of the the k-th vCPU.

Hence

$$\forall t, k, \quad \mathsf{psf}_1(t) \geq \mathsf{sbf}_k(t) \qquad \Rightarrow \qquad \forall t, \quad \mathsf{psf}_1(t) \geq \max_k \{\mathsf{sbf}_k(t)\}$$

as required.

Appendix B. Proof of Constraint 2

Lemma 2. *When $\mathsf{sbf}_k(t) = \alpha_k \cdot t$, Constraint 2 correctly enforces the schedulability test provided in Equation* (5)*.*

Proof. Consider an arbitrary task $\tau_i \in \mathcal{C}$ and an arbitrary virtual processor π_k. First note that, if τ_i is not allocated to π_k, i.e., $x_{i,k} = 0$, then $\mathcal{M} \cdot (2 - p_{i,q} - x_{i,k}) \geq \mathcal{M}$ idenependently of the value of binary variables $p_{i,q}$. Consequently, by definition of $\mathcal{M}$, the constraint has *no effect* independently of the LHS of Equation (6). Now, consider the case in which τ_i is allocated to π_k, i.e., $x_{i,k} = 1$. By definition of variables $x_{j,k}$, Equation (6) can be rewritten as

$$C_i + \sum_{\tau_j \in \mathsf{hp}(i) \cap \Gamma_k} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil C_j \leq \alpha_k \cdot \mathsf{tSet}_i[q] + \mathcal{M} \cdot (1 - p_{i,q}). \tag{B.1}$$

By definition of variables $p_{i,q}$, if $p_{i,q} = 1$ then the schedulability condition for τ_i must be verified at the q-th point of set tSet_i. By replacing $p_{i,q} = 1$ in Equation (B.1) we get

$$C_i + \sum_{\tau_j \in \mathsf{hp}(i) \cap \Gamma_k} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil C_j \leq \alpha_k \cdot \mathsf{tSet}_i[q]. \tag{B.2}$$

Since $\mathsf{sbf}_k(t) = \alpha_k \cdot t$, note that the above equation matches the inequality provided by Equation (5), thus correctly enforcing the schedualability condition at point $\mathsf{tSet}_i[q]$. Conversely, if $p_{i,q} = 0$, the constraint has no effect for the same reasons discussed above in the case for $x_{i,k} = 0$. Hence the lemma follows. □