Partitioning Real-Time Workloads on Multi-Core 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

Modern real-time virtual machines and containers are starting to make it possible to support the execution of real-time applications in virtualized environments. Real-time scheduling theory already provides techniques for analyzing the schedulability of real-time applications executed in virtual machines, but most of the previous work focused on global scheduling while, excluding a few exceptions, the problem of partitioning real-time workloads on multi-core VMs has not been properly investigated yet. This paper discusses and presents a set of partitioning algorithms, based on both mathematical optimization and some heuristics, to tackle the problem of online admission control and partitioning. An experimental evaluation shows that some of the heuristic algorithms can be effectively used in practical settings, being capable to partition complex task sets in short times and introducing an allocation overhead near to the optimum one.

Keywords: Real-time; virtual machines; hierarchical scheduling; cloud computing

1. Introduction

The recent improvements in CPU hardware and virtualization technology are starting to make virtual environments (such as the ones provided by cloud, fog, or edge computing) appealing even for time-critical applications, which were not traditionally considered candidates for execution in such environments. In particular, both the academic and the industrial communities are showing an increasing interest in virtualizing applications characterized by temporal constraints.

While applications with flexible and more relaxed temporal constraints are already supported in virtualized environments through appropriate resource provisioning and scaling (for example, consider cloud environments [1, 2, 3, 4]), novel applications with more strict timing requirements still lack adequate support and determine a series of challenges [5] to be tackled by research communities. Most of these challenges are introduced by unpredictable network latencies and by the fact that the physical servers hosting Virtual Machines (VMs) or containers do not provide temporal isolation or any kind of deterministic performance guarantees. Hence, the hosted virtual environments turn out to provide unstable processing times that are highly fluctuating depending on the executed workloads.

The networking issues, when present, can be addressed by moving the physical nodes that host the VMs closer to the application site so that the network connections are under control. This can be done (for example) by using *private cloud infrastructures* or by recurring to fog/edge architectures [6]. For instance, the usage of fog computing to respect the temporal constraints of served real-time applications has been recently investigated [7, 8].

The issues with the predictability of the nodes' performance, instead, can be addressed by using appropriate system-level support (real-time operating systems and real-time hypervisors) and management software. While modern VMs and hypervisors introduce latencies that are low enough for running real-time applications [9, 10, 11] and provide appropriate VM scheduling algorithms (RTDS [12] and `SCHED_DEADLINE` [13, 14]), the key issues in properly employing these scheduling algorithms have not been properly addressed yet. Such issues include the suitable configuration of scheduling parameters and the partitioning of real-time tasks upon the CPU cores.

Indeed, although some higher-level management stacks have been updated to support real-time features using either VMs [15] or containers [16, 17], they are still not providing tools and mechanisms for designing and configuring the scheduling parameters. Furthermore, while the design of time-critical VMs has been addressed in the literature, most of the solutions proposed for multi-core real-time VMs are either not effective in practice or force to over-allocate resources to VMs (as it will be shown in Section 4). In other words, the low-level system support provides adequate *scheduling mechanisms*, but higher-level management software does not implement *policies* for using them, mainly because such policies still have to be fully investigated.

1.1. Contributions

This work is a first step in investigating the design and resource allocation policies for using the low-level mechanisms provided by real-time scheduling algorithms already investigated in literature: after comparing the efficiency and effectiveness of various solutions (based on global and partitioned sche-

Email addresses: `luca.abeni@santannapisa.it` (Luca Abeni), `alessandro.biondi@santannapisa.it` (Alessandro Biondi), `bini@di.unito.it` (Enrico Bini) duling), this paper proposes and compares some approaches for partitioning the real-time workloads in multi-core VMs. Some of the proposed solutions are based on heuristic algorithms that are shown to be capable of providing comparable performance to optimal algorithms (based on mathematical optimization) while being characterized by a much smaller computational cost.

This result opens the way for online admission of real-time applications in dynamic VMs, such as the ones used in cloud/edge/fog computing.

The paper is organized as follows: Section 2 describes some related works; Section 3 provides some definitions and background information needed to understand the rest of the paper; Section 4 discusses some possible approaches for scheduling real-time tasks in a virtualized environment, also showing that the global scheduling approach performs much worse than the partitioned scheduling approach; Section 5 describes some algorithms for partitioning real-time tasks among virtual CPU cores; Section 6 describes the experiments performed to compare the various algorithms, and the obtained results. Finally, Section 7 states the paper's conclusions and presents some possible future work.

2. Related Work

Multi-processor/multi-core scheduling and hierarchical scheduling have been investigated in a vast portion of past real-time research. Multi-processor scheduling algorithms typically use a partitioned or a global scheduling approach:

- A global scheduler is free to migrate tasks among cores. Conceptually, the scheduler uses one single global ready queue containing all the tasks ready for execution.
- A partitioned scheduler, instead, does not migrate tasks among cores. Tasks are statically assigned to cores, and the scheduler is not allowed to change these assignments. In this way, the problem of scheduling tasks on multiple cores is reduced to multiple instances of a single-CPU scheduling problem (and single-processor real-time scheduling algorithms can be used). Here, the main challenge is how to partition the tasks among the available cores.

In theory, the absence of the tasks-to-cores mapping should favour global scheduling. In practice, however, the high complexity of exact schedulability tests for global scheduling [18, 19, 20, 21], together with the well-known pessimism of sufficient tests for both global EDF [22, 23] and global FP [24, 25], has kept the comparison between global and partitioned scheduling an open problem.

The global scheduling approach allows for developing optimal scheduling algorithms [26, 27, 28, 29, 30, 31]. However, they are more complex and introduce a high overhead due to the necessity to preempt jobs more frequently. In practice, to the best of our knowledge, none of the optimal algorithms above is implemented outside of the research context.

On the other hand, partitioned scheduling algorithms are simpler and introduce less overhead, but their efficiency depends on the partitioning step, which can be performed by solving a bin-packing problem that can be formulated as a Mixed Integer Linear Programming (MILP) problem. Instead of using a generic MILP solver, some well-known heuristics such as First Fit (FF) Worst Fit (WF), Best Fit (BF) and similar can be used [32].

In 1998, Oh and Baker [33] established lower and upper bounds to the utilization when allocating tasks over a multi-core until the Liu and Layland bound [34] on each core is reached. It has then been proved that if the taskset's utilization is smaller or equal than $(M + 1)/2$ (where M is the number of cores), then the taskset can be partitioned by using the FF heuristic and EDF [35].

Other MILP formulations have been used in literature to partition real-time tasks on multiple CPU cores [36, 37], but these previous works did not consider VMs or vCPUs scheduled with CPU reservations and did not address the problem of dimensioning the reservations when partitioning the taskset.

Hierarchical scheduling systems such as the one considered in this paper have been analyzed by first considering single-processor systems [38, 39, 40, 41, 42] and modelling the scheduling hierarchy through the time demanded by the taskset and the time provided by the root scheduler. This is the basis of the so-called Compositional Scheduling Framework (CSF) analysis.

Some previous works analyzed hierarchical scheduling systems when global algorithms are used for the guest scheduler. For example, the scheduling parameters for the various vCPUs can be designed by leveraging the Multiprocessor Periodic Resource model (MPR) [43, 44]. Lipari and Bini then showed that MPR analysis is based on some assumptions that are not always respected [45], and proposed the Bounded-Delay Multipartition (BDM) model to address this issue, at the price of some additional pessimism in the analysis. Other works extended the analysis of global guest schedulers by considering a more generic CPU allocation model [46] or considering more advanced scheduling algorithms [47, 48].

Partitioned scheduling in the guest has been investigated using a standard BF heuristic to assign tasks to vCPUs [12]. However, the work in [12] did not consider the scheduling hierarchy when partitioning the task set. A more advanced partitioning algorithm, taking into account virtual machine scheduling, has been then proposed [49]. Moreover, it has been noticed [14] that the schedulability analysis for global guest scheduling algorithms developed in previous works implicitly assumed some form of interaction between the guest scheduler and the host scheduler, which cannot be easily implemented in hypervisor-based VMs without using *para-virtualized scheduling*. The partitioned scheduling approach that has been proposed to address this issue will be referred to as the "$\alpha - 0$ model" in the rest of the paper. Even if OS-level virtualization (often known as container-based virtualization) allows using global guest scheduling without incurring the mentioned issue [50], global guest schedulers almost always result in a waste of computational resources (forcing to over-provision the VMs), as it will be shown in Section 4.

3. Definitions and Background

This work considers a physical computing node hosting a set of *real-time Virtual Machines* (real-time VMs) $\{\Pi_1, \ldots, \Pi_\ell\}$ that run concurrently. These real-time VMs are commonly used in cloud/edge/fog systems, as well as in embedded systems. The physical node is a multi-core machine, also referred to as *host*, that comprises M *identical* physical cores[1]. This paper considers only identical multi-core systems [51], in which all the CPU cores have the same Instruction Set Architecture (ISA) and the same maximum speed. Extensions considering multiple cores with different speeds will be considered as future work (and we believe that the partitioned scheduling approach presented in this paper can make it simpler to support such architectures). Each VM Π_j disposes of m_j *virtual CPUs* (vCPUs) $\{\pi^j_0, \ldots, \pi^j_{m^j-1}\}$ and hosts a guest Operating System (OS), with a corresponding kernel that provides its own CPU scheduler $\mathcal{S}_j$ (simply named *guest scheduler* from now on) mapping guest tasks over the vCPUs of the VM Π_j. While in theory each VM Π_j can have a generic number of vCPUs m_j, which does not strictly depend on the number of physical CPUs M, in practice there are no reasons to set $m_j > M$. The only constraint on the vCPUs π^j_k and their scheduling parameters is that they should be *schedulable* on the host, as will be discussed at the end of Section 3.1.

When addressing a VM in isolation, it is convenient to omit the index "j" of the VM to simplify the notation, so that a VM can be denoted by just Π, its guest scheduler by $\mathcal{S}$, and its vCPUs by π_k with $k = 0, \ldots, m - 1$.

When a new real-time application arrives in the system, a software component is in charge of allocating resources to the application (by creating a multi-core VM on which the application will execute) so that its temporal constraints can be respected. In particular, this software component must:

1. Compute the number m of vCPUs that are required to serve the application, and create a real-time VM with m vCPUs.
2. Configure the guest scheduler to properly schedule the application's real-time tasks on the m vCPUs.
3. Design the scheduling parameters for scheduling the m vCPUs on the M physical CPUs of the host.

Notice that if a VM has already been created for other real-time applications and is not fully utilized, it cannot be re-used for the new application (otherwise, the isolation between applications would be at risk). In theory, the computational capacity not used by a VM could be donated to other VMs by using appropriate reclaiming algorithms [52, 53], but this would not affect the design of the scheduling parameters (which is based on worst-case analysis and hence consider the reclaimed CPU time as 0).

Every real-time application must provide a description of its requirements and temporal constraints. Based on this description the application is either accepted to the real-time virtual environment, by performing the above three steps, or rejected. When accepting a new real-time application, it is fundamental to *guarantee* that all the temporal constraints of the application will be respected. To understand how to express the application's requirements and temporal constraints and how to provide such a guarantee, it is first important to introduce definitions about real-time computing and VMs.

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

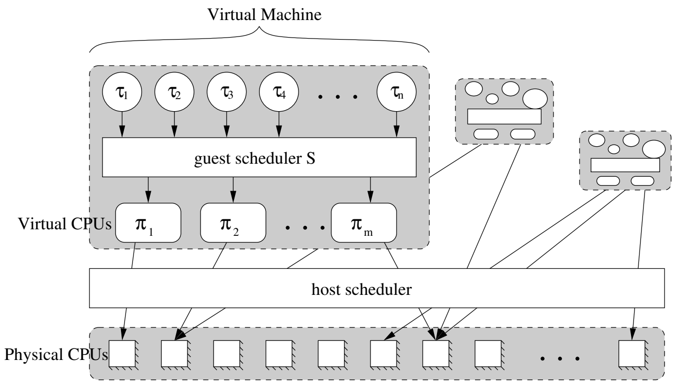


Figure 1: Hierarchical scheduling architecture overview.

This paper addresses the design and configuration of VMs and the partitioning of the application's tasks by considering the traditional task model [34]. The scheduling of real-time tasks with more complex structures executed in VMs [54] can then be analyzed based on the analysis of periodic/sporadic [55, 56] tasks.

A real-time application running inside a VM is hence modeled as a set of n independent real-time tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$. Each task τ_i generates an infinite sequence of jobs (i.e., activation instances) $J_{i,1}, J_{i,2}, J_{i,3}, \ldots, J_{i,h}, \ldots$, where each job $J_{i,h}$ becomes ready for execution at time $r_{i,h}$ and finishes at time $f_{i,h}$ after executing for a time $c_{i,h}$. The behavior of tasks is subject to timing constraints. The release times of two consecutive jobs of task τ_i, i.e., $r_{i,h+1}$ and $r_{i,h}$, must be separated by at least the *minimum inter-arrival time* T_i, formally stated by $\forall h = 1, 2, \ldots, r_{i,h+1} - r_{i,h} \geq T_i$. The execution times of all the jobs $J_{i,h}$ must be smaller than τ_i's *worst-case execution time (WCET)* C_i, i.e., $\forall h = 1, 2, \ldots, c_{i,h} \leq C_i$. Finally, every job $J_{i,h}$ must complete no later than its *absolute deadline* $d_{i,h}$, i.e., $\forall h = 1, 2, \ldots, f_{i,h} \leq d_{i,h}$. The absolute deadline $d_{i,h}$ of job $J_{i,h}$ is set by $d_{i,h} = r_{i,h} + D_i$, where D_i is the *relative deadline* of task τ_i. In this paper, the *constrained-deadline* model is assumed, that is $D_i \leq T_i$ for all tasks τ_i.

3.1. Hierarchical scheduling framework

As shown in Figure 1, the architecture considered in this work results in a two-level scheduling hierarchy: a *host scheduler* selects the vCPUs π_k to be executed on the host's physical CPUs, and then the VM's guest scheduler $\mathcal{S}$ selects a task τ_i for each vCPU. In these hierarchical scheduling systems, a common approach to guarantee that no deadline is missed relies on guaranteeing that each vCPU of the VM can execute for a minimum guaranteed amount of time, which can be guaranteed by using *resource reservation* algorithms in the host scheduler.

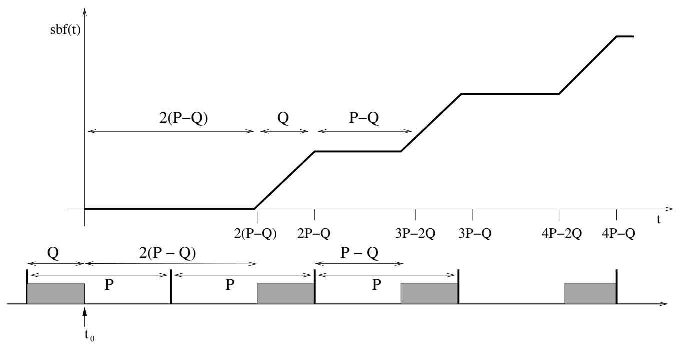


Figure 2: Computation of the supply bound function for a CPU reservation.

A reservation-based scheduler schedules some entity (a thread, process, or in this case a vCPU) for an amount of time Q, typically called *budget*, every *period* P. When applying this concept to real-time VMs, the host scheduler provides each vCPU π_k with a budget Q_k every period P_k. Note that enforcing a budget for π_k also means reserving a portion of time for the tasks selected to execute over π_k by the guest scheduler $\mathcal{S}$.

The schedulability of real-time tasks running in a VM Π can be analyzed by computing the minimum amount of CPU time provided by the host scheduler to Π over any interval of length t. For a single vCPU π_k, this quantity can be modeled through the so-called *supply bound function* $\mathsf{sbf}_k(t)$, representing a lower bound to the amount of CPU time that a vCPU π_k is guaranteed to receive in any time interval of length t. The logic of the computation of $\mathsf{sbf}_k(t)$ for a reservation (Q_k, P_k) is shown in Figure 2, where t_0 indicates the arrival time of any task τ (hence, the time interval of size t considered by $\mathsf{sbf}_k(t)$ starts at time t_0). The shape of the supply bound function is shown in the upper part of the figure. In the worst case, the task τ may be released exactly at the instant of budget exhaustion and then receive no budget for $2(P_k - Q_k)$. This implies that $\mathsf{sbf}_k(2(P_k - Q_k)) = 0$. At time $2(P_k - Q_k)$, the function $\mathsf{sbf}_k(t)$ must necessarily increase with slope 1 for an amount of time Q_k, reaching the value Q_k at time $2P_k - Q_k$. Then it is flat for an amount of time $P_k - Q_k$, and so on. More formally, the function is defined as [40]:

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

Due to the complexity of this equation, various authors [38, 39, 40, 41] proposed a linear lower bound for the supply function $\mathsf{sbf}_k(t)$ as $\mathsf{sbf}_k(t) \geq \alpha_k(t - \Delta_k)$, where:

- $\alpha_k = \frac{Q_k}{P_k}$ is the so-called *bandwidth* of vCPU π_k, and
- $\Delta_k = 2(P_k - Q_k)$ is the so-called *allocation delay* experienced by vCPU π_k, representing the longest time interval in which π_k receives no CPU time.

This is the so-called "$\alpha - \Delta$" model that will be used later in this paper.

Whenever a VM Π has multiple vCPUs $\pi_1, ...\pi_m$, it is served by m reservations $\{(Q_1, P_1), (Q_2, P_2), ...(Q_m, P_m)\}$ and the guest scheduler $\mathcal{S}$ must schedule tasks on more than one vCPU. In this case, $\mathcal{S}$ typically uses a *global* or a *partitioned* scheduling approach as mentioned in Section 2.

Summing up, accepting a new real-time application (according to the 3 items previously highlighted) requires to first

1. *select a guest scheduler* $\mathcal{S}$ (since the following steps depend on $\mathcal{S}$), then
2. *design the VM* scheduling (by computing the number of vCPUs m and the (Q_k, P_k) reservations needed to serve the application without missing deadlines), and finally
3. *check if the* (Q_k, P_k) *reservations are schedulable* on the host's physical CPUs.

In the design step at item 2, the number of vCPUs m is first decided (remember that there are no strict constraints on the value of m, but it is generally selected as $m \leq M$). As will be shown and discussed in Section 6, for some partitioning algorithms the choice of a large value of m does not impact the efficiency of the design (because the algorithm is able to allocate tasks only on some of the m possible vCPUs, if needed), while for other partitioning algorithms it is important to select a reasonable value for m (because the algorithm otherwise risks to spread the tasks over all the possible vCPUs, at the cost of resulting in an over-allocation of CPU time and generating reservations that are not schedulable on the host). For this second class of algorithms, m can be initially selected as the smallest integer number larger than the taskset's utilization, and iteratively increased if the design fails.

The check at item 3 depends on the actual reservation-based algorithm used by the host scheduler (for example, if the reservations are implemented by using partitioned EDF, then they are schedulable if the sum of the reservations' bandwidths on each core is smaller or equal than 1). If the check is passed, then the real-time application can be accepted in the system. Some of the partitioning algorithms discussed in this paper allow to take the existing real-time load into account when designing the CPU reservations at item 2 so that they can pass the schedulability check at item 3; see the discussion in Section 5.4.
.

4. Scheduling Real-Time Tasks in VMs

As previously mentioned, real-time VMs are executed by scheduling each vCPU π_k through a (Q_k, P_k) reservation (meaning that vCPU π_k is reserved an amount of time Q_k every period P_k). In bare metal hypervisors such as Xen [57], the hypervisor implements a vCPU scheduler that must support resource reservations; in hosted hypervisors, the vCPUs are generally seen by the host OS as threads and the host OS kernel is responsible for implementing the CPU reservation mechanism in its CPU scheduler; for OS-level virtualization (for example, Docker[2], Linux containers[3] or similar), the OS kernel provides some form of real-time containers.

[2]`www.docker.com`
[3]`www.linuxcontainers.org`

In any case, the temporal constraints of the real-time tasks running inside the VM are respected if the reservations' parameters Q_k and P_k are properly configured. While designing the reservation for VMs composed of a single vCPU is a widely studied problem (see Section 2), the analysis and design of multi-vCPU, reservation-based VMs is more challenging as it adds the difficulty of scheduling vCPUs to the host to the traditional server design problem. In particular, the design of reservations heavily depends on the guest scheduling strategy, which, as previously mentioned, can be either global or partitioned.

4.1. Global guest scheduling

If the guest OS uses a global scheduler (for example, a global fixed priority scheduler or global EDF), the CPU reservations (Q_k, P_k) can be designed by leveraging the MPR model. According to MPR, each real-time VM is reserved a total amount of CPU time Θ every period Φ, to be allocated over at most m virtual CPUs. Hence, the scheduling parameters of the VM can be expressed as (Θ, Φ, m); obviously, the constraint $\Theta \leq m\Phi$ must be always respected (since the maximum amount of time that can be provided by m vCPUs on a period Φ is $m\Phi$). In other words, the VM is associated with a multi-core reservation (Θ, Φ, m) that can be implemented by using m single-core reservations (Q_k, Φ) with $\sum_{k=0}^{m-1} Q_k = \Theta$ (all the vCPU reservations have the same period Φ of the multi-core reservation and the sum of their budgets Q_k is equal to the multi-core budget Θ). The schedulability of the real-time workload running inside a VM served by a multi-core reservation (Θ, Φ, m) is checked by computing a multi-core supply function for (Θ, Φ, m) and by comparing it with the amount of time demanded by the global guest scheduler to respect all the temporal constraints of the workload.

The computation of the multi-core supply function used by MPR assumes that the multi-core reservation (Θ, Φ, m) is implemented by using *m synchronized* single-core reservations (the reservation periods of all the m reservations start simultaneously), and if this assumption is not respected the resource supply bound for MPR can be smaller than the one computed in the original MPR analysis. The Bounded-Delay Multi-partition (BDM) model has then been proposed to address this issue, at the price of some additional pessimism in the analysis (as it will be shown in Figure 3).

Under the BDM approach, the supply bound function of each vCPU is approximated by using the $\alpha - \Delta$ model, and the constraint that all the vCPUs have the same allocation delay Δ is imposed. The vCPU reservations are correctly configured if all the tasks $\tau_i \in \Gamma$ are schedulable. A task τ_i is schedulable if

$$\exists k = 1, \ldots, m \qquad \sum_{j=0}^{k-1} \max\{0, \alpha_j(D_i - \Delta)\} \geq kC_i + W_i, \quad (2)$$

where W_i is the interference of other tasks, computed as in [58].

The optimal bandwidth values α_k can be found by solving the following optimization problem:

$$\begin{array}{l}
\textbf{minimize } \sum_{k=0}^{m-1} \alpha_k \text{ (objective A) } \textbf{or } \max_k\{\alpha_k\} \text{ (objective B)} \\
\textbf{subject to} \\
\forall \tau_i \in \Gamma, \forall k = 1, \ldots, m, \\
\quad L \cdot (1 - q_{i,k}) + \sum_{j=0}^{k-1} \max\{0, \alpha_j(D_i - \Delta)\} \geq kC_i + W_i \\
\forall \tau_i \in \Gamma, \sum_{k=1}^{m} q_{i,k} \geq 1
\end{array} \quad (3)$$

where $q_{i,k}$ is a binary variable with $q_{i,k} = 1$ indicating that task τ_i satisfies the schedulability constraint of Equation (2) for the integer k, and L is a constant large enough to make the inequality always true when $q_{i,k} = 0$. The N "$\sum_{k=0}^{m-1} q_{i,k} \geq 1$" constraints indicate that for each task at least one of the k schedulability constraints must be satisfied, as required by Equation (2). This results in a MILP with $2 \cdot N \cdot m$ constraints, which can be solved by means of standard software tools, either commercial (for example, CPLEX) or open-source (such as GLPK).

4.2. Partitioned guest scheduling

Partitioned scheduling in the guest is generally performed by first partitioning the tasks on the m vCPUs, and then designing the reservation parameters for each vCPU. The first step (partitioning) can, for example, be implemented by solving the following optimization problem [14]:

$$\begin{array}{l}
\textbf{minimize } \sum_{k=0}^{m-1} \alpha_k \text{ (objective A) } \textbf{or } \max_k\{\alpha_k\} \text{ (objective B)} \\
\textbf{subject to} \\
\forall \tau_i \in \Gamma, \forall k = 1, \ldots, m,, \forall 1 < q \leq |\mathsf{tSet}_i|, \\
\quad C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \\
\qquad \alpha_{k-1} \cdot \mathsf{tSet}_i[q] + L \cdot (2 - p_{i,q} - x_{i,k}) \\
\forall \tau_i \in \Gamma, \sum_{k=1}^{m} x_{i,k} = 1 \\
\forall \tau_i \in \Gamma, \sum_{q=1}^{|\mathsf{tSet}_i|} p_{i,q} \geq 1
\end{array} \quad (4)$$

where tSet_i is the set of schedulability check-points of task τ_i (with each element denoted by $\mathsf{tSet}_i[q]$) in which the request bound function of τ_i ($rbf_i(t)$) is computed [59], $p_{i,q}$ is a binary variable with $p_{i,q} = 1$ indicating that the schedulability constraint is satisfied for the q^{th} *checkpoint* (demanded time is smaller than the available time), $x_{i,k}$ is a binary variable indicating that task τ_i is assigned to the k^{th} vCPU, and L is again a large enough constant. Again, this is a MILP problem that can be solved by using standard tools.

Informally speaking, the first inequality in this optimization problem imposes that for each virtual CPU π_k, every task τ_i assigned to such vCPU (hence having $x_{i,k} = 1$) respects a schedulability condition ($C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \alpha_{k-1} \cdot \mathsf{tSet}_i[q]$) in at least a checkpoint $\mathsf{tSet}_i[q]$; the $L \cdot (2 - p_{i,q} - x_{i,k})$ term ensures that the inequality is true (and hence the schedulability condition does not need to be checked) if the task is not assigned to vCPU π_k ($x_{i,k} = 0$) or if $p_{i,q} = 0$. The second inequality states that each task is assigned to 1 and only 1 virtual CPU and the third inequality states that the schedulability condition of the first inequality is satisfied for at least a scheduling point $\mathsf{tSet}_i[q]$ per task. Again, note that the scheduling condition used in the first inequality is of the form $rbf_i(t) \leq \alpha \cdot t$, which uses the $\alpha - \Delta$ approximation of $sbf(t)$ with $\Delta = 0$.

Hence, this model will be called the "$\alpha - 0$ model" from now on.

Using the $\alpha - 0$ model, the partitioning algorithm aims at assigning tasks to virtual CPUs so that a valid (Q_k, P_k) reservation can be computed for each vCPU. This is because the first inequality basically imposes that on each vCPU π_k the schedulability test of fixed-priority scheduling is passed using a portion of the processor bandwidth $\alpha_{k-1} \leq 1$. As such, in the worst case, reserving the whole processor for a vCPU, i.e., with a limit-case reservation (Q_k, P_k) with $Q_k = P_k$, guarantees that all the tasks of the vCPU respect all their deadlines.

In general, if the total utilization of the tasks assigned to vCPU π_k is low enough, an optimal reservation design algorithm can compute a reservation (Q_k, P_k) with $Q_k < P_k$. Notice, however, that in this case the resulting reservation will typically have a bandwidth Q_k/P_k larger than α_k, as the first inequality of Equation (4) imposes $\Delta = 0$ (and hence does not account for any reservation-related service delay Δ_k). Hence, in this case α_{k-1} does not actually represent the real vCPU bandwidth (which is equal to the utilization of the CPU reservation serving the vCPU), but only a lower bound for it. This is an important difference between this $\alpha - 0$ model and the more traditional $\alpha - \Delta$ model.

Finally, it is worth noticing that the first inequality is similar to the inequality used for BDM (there are more constraints, but the main constraints only contain a single α_k and not a sum on α_k). The main difference between the optimization problems of Equation 3 and Equation 4 is that, in the first one (used for BDM), the interference W_i is computed in a very pessimistic way, while in the second one (used for partitioned scheduling) it is possible to use the exact workload $rbf_i(t) = \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{t}{T_j} \right\rceil (C_j \cdot x_{j,k})$.

4.3. Comparing global vs. partitioned guest scheduling

Before focusing on a specific scheduling approach, it makes sense to compare the performance of global guest schedulers with the performance of partitioned guest schedulers. A pragmatic metric that can be evaluated to perform such a comparison is the *allocation overhead*, defined as the difference between the sum of all the bandwidths of the vCPU reservations, i.e., $\sum_{k=0}^{m-1} \frac{Q_k}{P_k}$, and the utilization $\sum_i \frac{C_i}{T_i}$ of the taskset: $O = \sum_{k=0}^{m-1} \frac{Q_k}{P_k} - \sum_i \frac{C_i}{T_i}$.

To compare the performance of global and partitioned scheduling approaches, we generated a large number of task sets using the popular *Randfixedsum* algorithm as implemented by Emberson et al. [60] and designed the vCPU reservations (Q_k, P_k) for three scheduling approaches: MPR, BDM, and the partitioning algorithm from [14]. As a representative example of the obtained results, Figure 3 reports the allocation overhead for task sets composed of different numbers of tasks (from $N = 4$ to $N = 8$) and having different utilizations (ranging from $U = 1.0$ to $U = 1.8$) scheduled on VMs composed by 2 vCPUs ($m = 2$). In the figure, "MPR Original" indicates the utilization computed using the original MPR algorithm, which uses a global EDF scheduler in the guest and "MPR BCL" indicates the utilization computed using the CARTS tool [61], which implements a modified version of the analysis, using the interfering workload computation by Bertogna, Cirinei and Lipari (BCL) [58] and a global fixed priority scheduler in the guest[4]. BDM uses the same interfering workload computation used by "MPR BCL".

From the results presented in Figure 3 it is possible to immediately notice that the allocation overhead introduced by a partitioned guest scheduler is almost constant and quite small. On the other hand, global scheduling approaches tend to introduce much higher allocation overheads. Similar experiments with different numbers of tasks and vCPUs (and different tasksets' utilizations) confirmed these results.

Based on the results of these experiments, the rest of the paper will focus on partitioned guest schedulers, which in this kind of hierarchical scheduling always significantly outperform global approaches.

Notice that similar results have been noticed for non-hierarchical fixed-priority and EDF schedulers, and are due to the pessimism of the global schedulability analysis [62]. While for non-hierarchical systems this is merely an *analysis* issue and the only consequence is that schedulable task sets will not be accepted (many task sets that do not pass the admission test are actually schedulable), for scheduling hierarchies like the ones considered in this paper it results in a *design issue* with more important consequences. In fact, since the MPR or BDM algorithms are used to design the vCPU servers (finding appropriate (Q_k, P_k) assignments for all vCPUs), the pessimism of the analysis actually results in much larger CPU allocations and the consequent waste of system resources.

5. Partitioning Algorithms

As previously noticed, the $\alpha - 0$ model allows minimizing $\sum_k \alpha_k$, but this does not necessarily minimize the total fraction of CPU time $\sum_k Q_k/P_k$ allocated to the VM, which, in practice, is the important metric to be actually minimized. As a simple example, consider $\Gamma = \{\tau_1 = (C_1, T_1), \tau_2 = (C_2, T_2)\}$, with $C_1/P_1 + C_2/P_2 = 0.5$, scheduled on 2 vCPUs. The optimization problem based on Equation (4) will place τ_1 on the first vCPU and τ_2 on the second one, with $\alpha_1 = C_1/P_1$ and $\alpha_2 = C_2/P_2$, which leads to the minimum possible sum $\alpha_1 + \alpha_2 = 0.5$. When designing the actual reservation servers (i.e., with $\Delta_k > 0$), however, $Q_1/P_1 + Q_2/P_2$ will be larger than 0.5, and it might happen that placing both the tasks on the same vCPU can result in a lower total CPU bandwidth. As it will be shown in Section 6, an optimal algorithm minimizing $\sum_k Q_k/P_k$ often places as many tasks as possible on a single vCPU[5].

As a result, designing a VM for a real-time taskset $\Gamma = \{\tau_i\}$ based on the $\alpha - 0$ model requires a 2-steps algorithm: **first**

[4]Note that the original MPR algorithm uses a less pessimistic computation of the workload, but global EDF analysis is more pessimistic than global fixed priority analysis.

[5]Under global scheduling, it has been formally proved [63] that the "optimal" allocation, i.e., the one providing the minimum possible fraction of CPU time to the VM while guaranteeing its schedulability, is the one that uses the smallest possible number of vCPUs — this is the so-called *minimum parallelism form*.

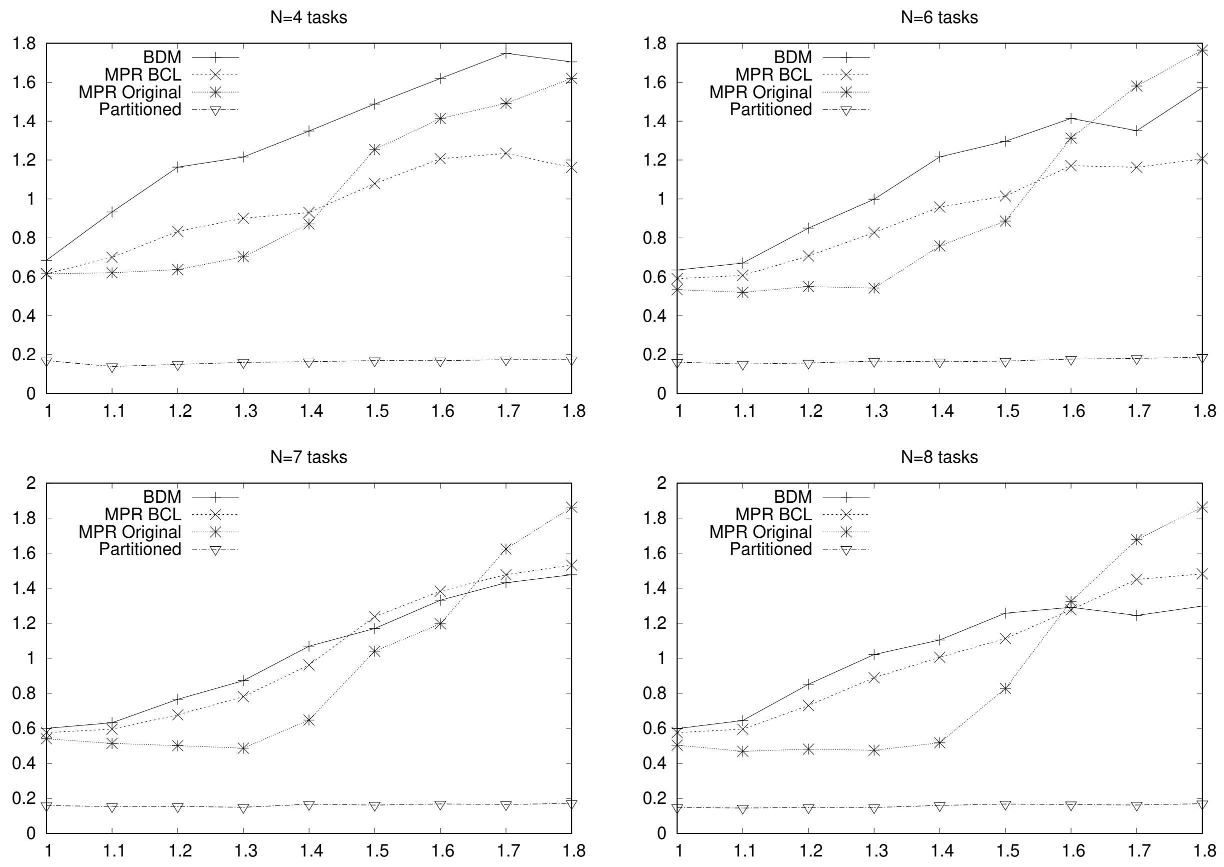


Figure 3: Figure comparing the allocation overhead (the lower the better) of global scheduling approaches (MPR and BDM) vs. partitioned scheduling under the $\alpha - 0$ model. The y-axis of the plots reports the allocation overhead while the x-axis reports the task set utilization.

partition the taskset, **then** design the vCPU reservations. As discussed above, this might lead to sub-optimal resource allocations.

Moreover, in many practical situations it is important to impose a maximum U_k^{max} to the fraction of CPU time Q_k/P_k reserved to a vCPU. For example, this is important for online admission control of the VMs (as discussed in Section 5.4), for preventing the starvation of other tasks/VMs (it is important to leave some unused CPU time for background tasks), or for leaving some vCPU time for the guest OS. However, when using the $\alpha - 0$ model $\alpha_k \neq Q_k/P_k$, hence constraints such as "$\alpha_k \leq U_k^{max}$", would not guarantee that $Q_k/P_k \leq U_k^{max}$ after the design of the reservation in the second step. Again, accounting for the allocation delay Δ_k when partitioning the taskset could help to solve this problem.

5.1. Accounting for the Allocation Delay

To address some of the issues with the $\alpha - 0$ model, it is possible to use the $\alpha - \Delta$ model with a fixed $\Delta > 0$. While still leading to sub-optimal solutions, being the service delay not optimized, this approach has the important property that $\alpha_k = Q_k/P_k$ already at the stage of partitioning. As such, it allows to do partitioning and design in one single step.

Since short reservation periods might result in increased scheduling overhead (the OS kernel generally arms a timer firing at the end of the reservation period), when designing the VM scheduling parameters it is important to also impose a minimum value for the reservation period P_k. Hence, a constraint $P_k \geq P^{min}$ should be added to the optimization problem of Equation 4 for each vCPU π_k. However, this is difficult to express in terms of α_k and Δ_k without using non-linear constraints. In theory, the minimum reservation period could be enforced by using a constraint $\Delta_k \geq \Delta^{min}$ on the minimum allocation delay. However, the modified schedulability constraints would require a non-linear term of the form $\alpha_k(t - \Delta_k)$, which would make the optimization problem not anymore a MILP.

This issue can be addressed without renouncing to the MILP formulation by introducing a new variable $\beta_k = \alpha_k\Delta_k$ and modifying Equation 4 as follows:

$$\begin{array}{l}
\textbf{minimize} \sum_{k=0}^{m-1} \alpha_k \text{ (objective A) } \textbf{or} \max_k \alpha_k \text{ (objective B)} \\
\textbf{subject to} \\
\forall \tau_i \in \Gamma, \forall k = 1, \ldots, m, \forall 1 < q \leq |\mathsf{tSet}_i|, \\
\quad C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \\
\qquad \alpha_{k-1} \cdot \mathsf{tSet}_i[q] - \beta_{k-1} + L \cdot (2 - p_{i,q} - x_{i,k}) \\
\forall \tau_i \in \Gamma, \sum_{k=1}^{m} x_{i,k} = 1 \\
\forall \tau_i \in \Gamma, \sum_{q=1}^{|\mathsf{tSet}_i|} p_{i,q} \geq 1 \\
\forall 0 \leq k < m, \beta_k \geq \frac{P^{min}}{2}.
\end{array} \tag{5}$$

Lemma 1. *The optimization problem presented in Equation 5 partitions the tasks on m different vCPUs so that for all the vCPUs the tasks assigned to π_k are schedulable with parameters (Q_k, P_k) and $P_k > P^{min}$.*

Proof. The optimization problem is similar to the one presented in Equation 4, but uses the schedulability condition

$$C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \alpha_{k-1} \cdot (\mathsf{tSet}_i[q] - \Delta_{k-1})$$

instead of

$$C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \alpha_{k-1} \cdot \mathsf{tSet}_i[q]$$

By introducing $\beta_k = \alpha_k \Delta_k$, the constraints of the form $\alpha_k(t - \Delta_k) = \alpha_k t - \alpha_k \Delta_k$ can be rewritten as $\alpha_k t - \beta_k$, hence

$$C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \alpha_{k-1} \cdot \mathsf{tSet}_i[q] + L \cdot (2 - p_{i,q} - x_{i,k})$$

becomes

$$C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \leq \alpha_{k-1} \cdot \mathsf{tSet}_i[q] - \beta_{k-1} + L \cdot (2 - p_{i,q} - x_{i,k})$$

which is still a linear constraint[6].

The relationship between β_k and P_k can be found by remembering that $\alpha_k = Q_k / P_k$ and $\Delta_k = 2(P_k - Q_k)$, so

$$\beta_k = \alpha_k \Delta_k = \alpha_k 2(P_k - Q_k) \Rightarrow \beta_k = 2 P_k \alpha_k (1 - \alpha_k)$$

Hence, a minimum period P^{min} can be enforced by imposing $\beta_k \geq 2 P^{min} \alpha_k (1 - \alpha_k)$. Since $\alpha_k(1 - \alpha_k)$ has a maximum for $\alpha_k = 1/2$, the constraint can be written as

$$\beta_k \geq 2 P^{min} \frac{1}{2}(1 - \frac{1}{2}) \Rightarrow \beta_k \geq \frac{P^{min}}{2}$$

independently of the server bandwidth α_k. Combining these constraints, the MILP shown in Equation (5) is obtained. □

[6]As for Equation 4, the $L \cdot (2 - p_{i,q} - x_{i,k})$ term has been introduced to make the inequality true when $p_{i,q} = 0$ or $x_{i,k} = 0$, regardless of the other variables.

This will be called the $\alpha - \beta$ model in the rest of the paper. Although imposing $\beta_k \geq P^{min}/2$ guarantees that $P_k \geq P^{min}$ (as shown above) and has the advantage of not breaking the linearity of the optimization problem (which is still a MILP, easily solvable with many tools), this constraint on β_k is much stricter than the actual minimum period constraint, i.e., there may exist many valid solutions with $P_k \geq P^{min}$ that do not respect $\beta_k \geq P^{min}/2$). As a result, the optimization problem of Equation (5) risks resulting in a resource over-allocation.

The only way to avoid the over-allocation issues presented by the $\alpha - \beta$ optimization problem is to impose the actual minimum period constraint and avoid fixing β. However, this implies renouncing to linear problem formulations. For instance, it would be tempting to encode the joint partitioning and server design problem with convex optimization techniques such as Geometric Programming (GP) [64]. Indeed, it is possible to reshuffle the equation of the linear $\alpha - \Delta$ supply function to encode the corresponding schedulability test as a set of GP constraints. Nevertheless, when attempting this solution it is not possible to correctly encode the required decision constraints (task-to-vCPU allocation and selection of valid check-points for the schedulability test) in a GP. This is due to the fact that the problem formulation requires boolean decision variables (i.e., in $\{0, 1\}$), while GP, even in its mixed-integer form, only allows working with strictly positive variables. This limitation has been assessed by testing both commercial (MOSEK) and open-source (CVX) solvers for convex programming [65] that support GP.

5.2. Optimal partitioning and server design

As an alternative way to find an optimal solution to the design problem, a Constrained Programming (CP) problem has been formulated to compute the optimal partitioning and server parameters (budgets and periods for each vCPU). CP also allows directly encoding important constraints such as enforcing a minimum server period $P^{\min}$. Although CP formulations are notably hard and slow to solve, this approach represents a baseline for comparison in the experimental evaluation to assess the distance of other solutions from the true optimal one.

The CP formulation is reported next:

$$\begin{array}{l}
\textbf{minimize} \sum_{k=0}^{m-1} Q_k / P_k \\
\textbf{subject to} \\
\forall \tau_i \in \Gamma, \forall k = 1, \ldots, m, \forall 1 < q \leq |\mathsf{tSet}_i|, \\
\quad \left(C_i + \sum_{\tau_j \in \mathsf{hp}(i)} \left\lceil \frac{\mathsf{tSet}_i[q]}{T_j} \right\rceil (C_j \cdot x_{j,k}) \right) \cdot \\
\quad \cdot x_{i,k} \cdot p_{i,q} \leq \mathsf{sbf}(\mathsf{tSet}_i[q], Q_{k-1}, P_{k-1}) \\
\forall \tau_i \in \Gamma, \sum_{k=1}^{m} x_{i,k} = 1 \\
\forall \tau_i \in \Gamma, \sum_{q=1}^{|\mathsf{tSet}_i|} p_{i,q} \geq 1 \\
\forall 0 \leq k < m, P_k \geq P^{\min}
\end{array} \tag{6}$$

where tSet_i is the set of check-point for verifying the schedulability of task τ_i, $p_{i,q}$ is a binary variable with $p_{i,q} = 1$ indicating that the schedulability test for task τ_i is satisfied in the check-point $\mathsf{tSet}_i[q]$, and $x_{i,k}$ is a binary variable indicating that task τ_i is assigned to the k^{th} vCPU.

Lemma 2. *A solution of the CP formulation in* (6) *is an optimal configuration of the reservation servers with minimal* $\sum_{k=0}^{m-1} Q_k/P_k$ *that ensures the task set schedulability.*

Proof. For each processor with index k, the first constraint in (6) encodes the (exact) schedulability test for fixed-priority scheduling considering the tasks allocated to the k-th processor only. Indeed, note that (i) whenever task τ_i is not allocated to the k-th processor ($x_{i,k} = 0$) the constraint has no effect, (ii) the schedulability constraint does not have to be verified for all the check-points, but only for the ones having $p_{i,q} = 1$, and (iii) whenever an interfering task τ_j is not allocated to the k-th processor ($x_{j,k} = 0$) its workload contribution to the first constraint is zero.

The second constraint enforces that each task is allocated on exactly one processor, and the third constraint enforces that for each task the schedulability test is verified in at least one of its check-points.

The fourth constraint simply enforces a minimum period for the servers.

Hence the correctness of the CP formulation. □

Equation (6) can be seamlessly extended to perform the selection of the server parameters with a certain granularity by replacing variables Q_k and P_k with $Q_k \cdot Q^{\text{grain}}$ and $P_k \cdot P^{\text{grain}}$, where Q^{grain} and P^{grain} are the granularity of the server budget and period, respectively.

5.3. Heuristic algorithms

All the partitioning and design algorithms described above require a formulation of the problem as a MILP or CP, which can be solved by using tools like GLPK or CPLEX. These tools generally require from a few milliseconds (for small tasksets) to some tents of seconds (for larger tasksets scheduled on more vCPUs) to generate a solution and are hence mostly suitable for off-line design.

Conversely, appropriate heuristics algorithms can be used to support online partitioning and design. For example, the well-known FF, BF and WF heuristics can be modified to support scheduling hierarchies with the $\alpha - 0$ and $\alpha - \Delta$ models, or even using exact schedulability tests. The basic idea is that the usual *F algorithms can be applied by using a *pseudo-utilization pu* instead of the taskset utilization $\sum C/T$, as shown in Algorithm 1. If such a pseudo-utilization is computed as in Algorithm 2, then the resulting algorithm can find approximate solutions based on the $\alpha - 0$ model. Notice that this algorithm tries to ensure that $rbf_k(t)/t \leq U^{max}$ for at least a *scheduling check-point* of each task τ_k (for each task τ_k, the minimum of the ratio $rbf_k(t)/t$ on all the scheduling check-points is used, so that the condition is verified for *at least* a scheduling check-point). Moreover, the maximum on all the tasks of the taskset is used to ensure that the condition is respected *for all tasks*. As it will be shown in Section 6, these heuristics can successfully partition a task set over multiple vCPUs in a few milliseconds.

If we want to use an exact model of the $sbf()$, instead, then Algorithm 3 can be used. Notice that this algorithm computes the pseudo-utilization by designing the optimal server for the vCPU (performed by the `design_server()` function). Such a function tries all the possible periods $P_k \geq P^{min}$ and budgets $P_k U(\Gamma_k) \leq Q_k \leq P_k$, checking if the task set Γ_k is schedulable with them; then, the (Q_k, P_k) pair that can correctly schedule Γ_k and minimizes Q_k/P_k is selected. This algorithm can be optimized by reducing the (Q_k, P_k) search space [66].

Finally, it is important to notice that the FF, BF and WF heuristics, which are traditionally used for bin-packing, might be sub-optimal when partitioning tasks among vCPUs scheduled through CPU reservations. For example, using BF on the pseudo-utilizations will end up placing a task on the vCPU where it increases the pseudo-utilization more (consuming more bandwidth, and hence causing a higher allocation overhead). To reduce the allocation overhead, it would be more interesting to allocate a task on the vCPU where it would cause the minimum increase in the overhead (computed as the difference between Q_k/P_k and the utilization of the tasks allocated to the CPU — obviously subject to the constraint that the vCPU's reservation (Q_k, P_k) is schedulable). This will be referred to as the "Ovh" heuristic in the rest of the paper.

```
Input: The set of vCPUs (with the tasks allocated to
       them)
Input: The new task $\tau_i$ to be allocated
Output: The vCPU on which the task should be
        allocated
foreach vCPU $\pi_j$ do
    Add $\tau_i$ to $\pi_j$;
    pseudo_u ← Compute_pseudo_u($\pi_j$);
    if pseudo_u < 1 then
        /* Apply the standard FF / WF / BF
           heuristics                          */
        if mode == FF then
            return $\pi_j$
        end
        if pseudo_u > pseudo_u_max then
            pseudo_u_max ← pseudo_u;
            Vmax ← $\pi_j$
        end
        if pseudo_u < pseudo_u_min then
            pseudo_u_min ← pseudo_u;
            Vmin ← $\pi_j$
        end
    end
end
if mode == WF then
    return Vmin
end
if mode == BF then
    return Vmax
end
return Error!
```
Algorithm 1: The heuristic partitioning algorithm.

5.4. Discussion

When comparing the various partitioning algorithms, there are different aspects to be considered. For example, as men-

```
Function Compute_pseudo_u(V):
    pseudo_u ← 0 ; foreach task $\tau_k \in V$ do
        compute scheduling points (checkpoints);
        pu ← 1.1 ;
        foreach scheduling point t do
            compute $rbf_k(t) = \sum_{\tau_j \in hp(k)} \lceil (t/T_j) \rceil C_j$;
            if $rbf_k(t)/t < pu$ then
                pu ← $rbf_k(t)/t$
            end
        end
        if $pu >$ pseudo_u then
            pseudo_u ← pu
        end
    end
    return pseudo_u
```

Algorithm 2: PseudoU computation: Approximate $\alpha - 0$ model.

```
Function Compute_pseudo_u(V):
    $(Q, P)$ ← design_server($V$);
    return $Q/P$
```

Algorithm 3: PseudoU computation: Exact supply bound function model.

tioned in Section 3, the resource control component of some real-time virtual environments (such as real-time cloud/edge/fog systems) must be able to dynamically accept new real-time applications at runtime. Assuming a known guest scheduling algorithm (fixed-priority scheduling with static partitioning of the tasks), when a new application arrives, the system must decide how many vCPUs to use, partition the application's task set among vCPUs, and design the (Q_k, P_k) reservation parameters for each vCPU. The resulting (Q_k, P_k) reservations must be scheduled on the physical CPUs, hence they must pass a schedulability test that depends on the host scheduling algorithm. If the system is not able to compute appropriate reservation parameters or if the CPU reservations do not pass the admission tests, then the application cannot be accepted in the system. This means that **the partitioning and design algorithms must be fast enough to be executed online during the system operation**. As it will be shown in Section 6, the heuristic algorithms are always fast enough to support online admission tests, while the partitioning algorithms based on optimization problems (MILP or CP) can be used only for small task sets scheduled on a limited number of vCPUs.

Moreover, when a new real-time application arrives, some of the physical CPUs might already host some real-time load with utilization U^k, meaning that they cannot accept a reservation with $Q_k/P_k > 1 - U^k$. Hence, a constraint $Q_k/P_k < U_k^{max}$, with $U_k^{max} = 1 - U^k$ should be added to the design problem. As already noticed, this could be problematic when using the $\alpha - 0$ model with the problem formulation of Equation 4, because in this model $\alpha_k \neq Q_k/P_k$ (hence, even imposing constraints $\alpha_k \leq U_k^{max}$ the solution of the problem can result in reservations (Q_k, P_k) with $1 - U^k < Q_k/P_k \leq 1$, which cannot be scheduled). In this case, the algorithms that can take the allocation delay into account (such as the ones based on the $\alpha - \beta$ model, or the heuristic algorithms with the pseudo-utilization computed as in Algorithm 3) are more useful and can be directly used (by adding $\alpha_k \leq U_k^{max}$ in Equation 5 or by changing "`if pseudo_u < 1`" into "`if pseudo_u <`U_k^{max} in Algorithm 1).

The third important thing to be considered is that the MILP (for $\alpha - 0$ and $\alpha - \beta$ models) and CP partitioning algorithms can be used only if all the real-time tasks composing an application are known when the application starts (so, real-time tasks cannot be dynamically created). This constraint is due to the fact that the MILP or CP problem used to partition the tasks requires a complete knowledge of the whole task set. The heuristic algorithms, instead, can be used also if real-time tasks are dynamically created (each task is assigned to a vCPU when it is created). If all the real-time tasks of the application are created at the same time, when the application starts, the performance of the heuristic algorithms can be improved by pre-ordering (for example, by decreasing utilization) the tasks before applying the FF, WF, or BF heuristics. When tasks are ordered by decreasing utilization before assigning them to vCPUs (so that tasks with higher utilization, which are more difficult to place, are served first) the partitioning heuristics will be named U-FF, U-WF, U-BF and so on.

6. Experimental Evaluation

The various partitioning and design algorithms presented and discussed in Section 5 have been compared through a set of experiments, based on synthetic task sets with various characteristics. All the task sets have been generated by using the Randfixedsum algorithm, which is a standard way to generate sets of real-time tasks. In each experiment, multiple task sets composed of different numbers N of tasks (ranging from $N = 4$ to $N = 16$) and having different utilizations U (ranging from 1.0 to values immediately smaller than N) have been generated, partitioned over m vCPUs (ranging from $m = 2$ to $m = 16$), and used to design the vCPU reservations (Q_k, P_k) (with $0 \leq k < m$) based on the different algorithms presented above. This section reports and comments some selected results that allow to understand and evaluate the various partitioning algorithms. Whenever differently specified, 100 task sets per point have been generated to obtain the plots presented in the following.

Notice that if a partitioning and design algorithm is able to partition a taskset among m vCPUs and to design the vCPUs scheduling parameters, then all the tasks of the taskset are guaranteed to respect all of their deadlines. Hence, measuring values such as the number of missed deadlines, the deadline miss ratio or similar is not a good way to compare different algorithms. The performance of the algorithms can be instead evaluated by measuring how much CPU time the algorithms require to reserve for the VM, or, better, the *allocation overhead*, as defined in Section 4. However, evaluating only the allocation overhead could be misleading, because some algorithms might be able to correctly partition tasksets that other algorithms mark as unschedulable. Hence, it can be interesting to measure also another metric: the *fraction of schedulable tasksets*. This metric is measured by testing each algorithm on a large number of

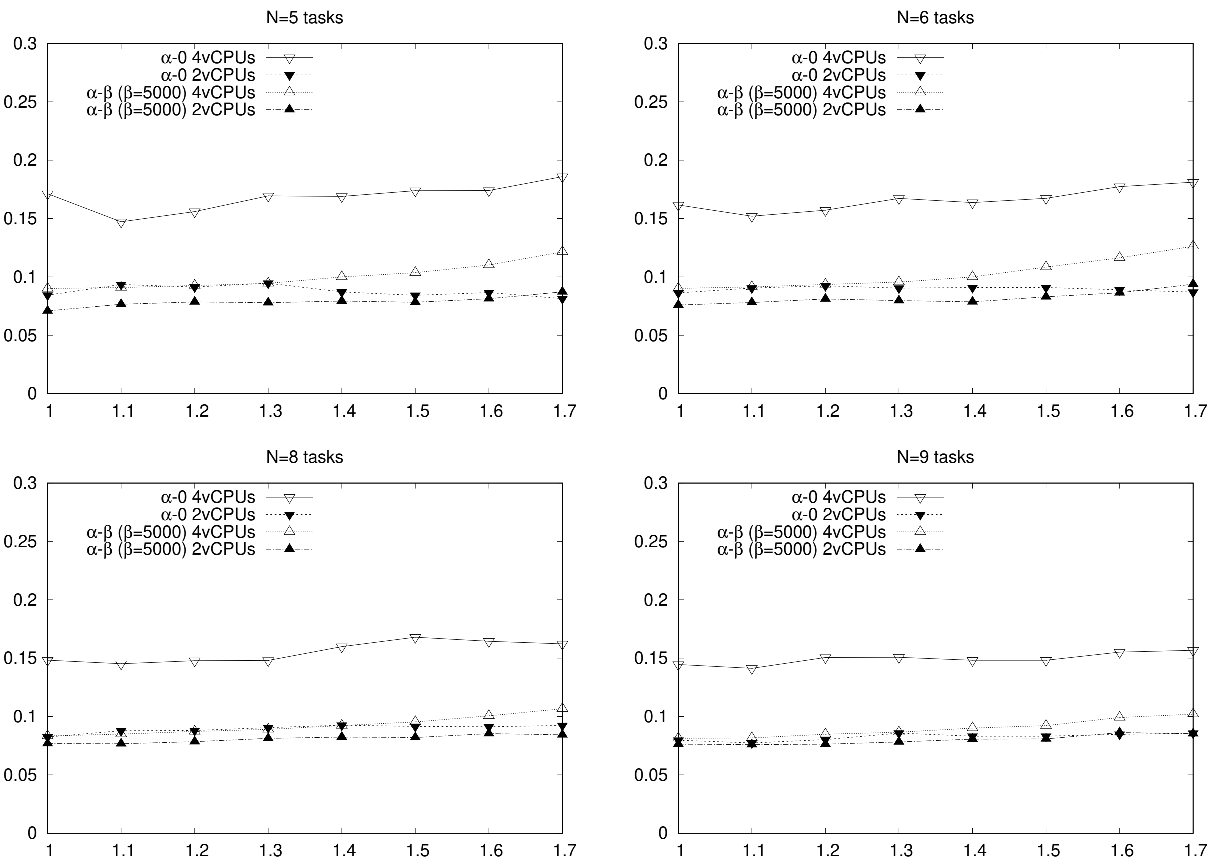


Figure 4: Figure comparing the allocation overhead of the $\alpha - 0$ and $\alpha - \beta$ models when different numbers of vCPUs are used. The x-axis of the plots reports the task set utilization while the y-axis reports the allocation overhead.

tasksets, and counting how many of these tasksets the algorithm is able to correctly partition and mark as schedulable.

This section is organized as follows: first of all, the $\alpha - 0$ model is compared with the new $\alpha - \beta$ model (in terms of allocation overhead), showing an issue with the old model and how the new model addresses such an issue. Then, the various heuristic algorithms from Section 5 are compared, highlighting the three algorithms that perform better. Finally, such algorithms are compared (in terms of allocation overhead, fraction of schedulable tasksets, and running time) with the partitioning algorithms based on the $\alpha - 0$ and $\alpha - \beta$ models and with the optimal solution based on Constrained Programming presented in Section 5.2 (notice that due to the large amount of time needed to solve the Constrained Programming problem, the optimal solution has been computed only on the first 30 tasksets).

6.1. *Issues with the $\alpha - 0$ Model*

First of all, the differences between the allocations obtained using the $\alpha - 0$ and the $\alpha - \beta$ model have been investigated. Figure 4 shows the allocation overhead measured when task sets composed of $N = \{5, 6, 8, 9\}$ tasks and utilization U ranging from 1 to 1.7 are scheduled over $m = 2$ and $m = 4$ vCPUs. Tasks are partitioned using the $\alpha - 0$ and $\alpha - \beta$ models; since $P^{min} = 10ms$ is considered, β has been set equal to $5ms = 5000\mu s$. From the figure, it is immediately possible to notice that the $\alpha - 0$ model results in tasks partitionings that depend on the number of vCPUs: when more vCPUs are used, the tasks are distributed over all the vCPUs (because the model does not consider the allocation delay Δ) and the allocation overhead increases. In particular, the figure shows that the $\alpha - \beta$ model (with $\beta = 5000$) results in almost the same allocation overhead when $m = 2$ or $m = 4$ vCPUs are used, while the $\alpha - 0$ model gives comparable results when $m = 2$ vCPUs are used, but results in a much larger overhead for $m = 4$ vCPUs.

These results show that, when the allocation delay is not considered, the VM design must cope with the selection of the "right" number of vCPUs to be used (see the discussion at the end of Section 3.1), while, when the allocation delay is accounted for, the VM can be designed by using the largest available number of vCPUs (and the problem's solution will result in $Q_k = 0$ for the vCPUs that are not needed — not allocating any task to them).

Experiments with different values of N, U, and m, not reported here due to lack of space, confirmed this result.

6.2. *The Heuristic Algorithms*

After comparing the $\alpha - 0$ and $\alpha - \beta$ models, the performance of the heuristic algorithms has been evaluated. As discussed in

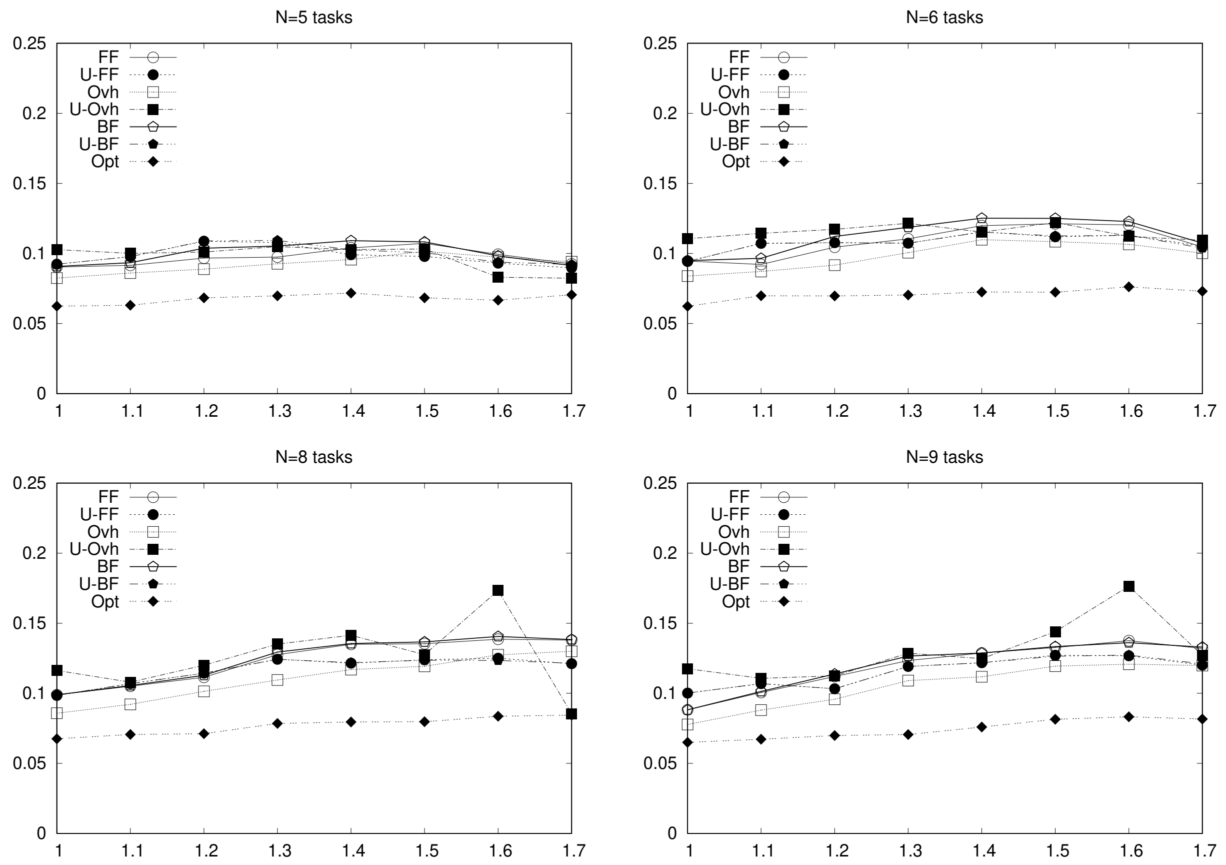


Figure 5: Allocation overhead of the heuristic algorithms on 2 vCPUs.

Section 5, different heuristics are possible, using FF, BF, WF, or Ovh and evaluating the pseudo-utilization in different ways. Moreover, if real-time tasks are not dynamically created, then the task set can be ordered by decreasing utilization before applying the partitioning algorithm. When task sets are ordered by the decreasing utilization, the corresponding heuristic algorithm is named with the 'U-' prefix.

The experiments showed that the WF-based algorithms always cause a large allocation overhead (this is consistent with some previous results on partitioning of real-time tasks on non-hierarchical schedulers [35]), hence the results of these algorithms have been removed from the figures presented in this section. Figure 5 reports the results obtained scheduling a limited number of tasks (ranging from $N = 4$ to $N = 10$) on only $m = 2$ vCPUs and shows that in this case all the heuristics (except for U-Ovh in some cases) behave similarly, but the Ovh heuristic generally causes a smaller allocation overhead. Moreover, the allocation overhead of the heuristics is similar to the minimum possible (the optimal CP-based partitioning, marked as "Opt" in the figures).

Figure 6 plots the allocation overhead of various heuristic algorithms for larger tasksets, scheduled on 8 vCPUs (in this case, the utilization ranges from $U = 1$ to $U = 7$ and the number of tasks ranges from $N = 4$ to $N = 16$). The results displayed in this figure are more interesting (exhibiting more differences in the performance of the various heuristics), but still show that the "Ovh" heuristic performs quite well in all the cases, resulting only in a small increase in allocation overhead (less than 5%) respect to the optimal partitioning. Then, it is interesting to notice how the BF algorithm generally has performance comparable with FF (again, this is in line with previous results [35]) and is worse than FF in some cases. It is also worth noticing that FF performs well for small values of N (see $N = 6$ or even $N = 8$), but has some issues when the number of tasks increases (see $N = 10$ and $N = 12$). In this case, pre-ordering the taskset according to decreasing utilization (see the "U-FF" plot) reduces the allocation overhead.

According to these results, the heuristic algorithms considered in the next experiments are FF, U-FF and Ovh.

6.3. Comparing all the Algorithms

The next results compare the performance of the practical heuristic algorithms presented in Section 5 with the partitioning algorithms based on the $\alpha - 0$ and $\alpha - \beta$ models. Figure 7 shows the allocation overhead obtained when considering small task sets scheduled on $m = 4$ vCPUs and permits to notice some important things. First of all, the $\alpha - 0$ model shows the highest overhead: this happens because the design is performed over $m = 4$ vCPUs, while $m = 2$ vCPUs would generally suffice to properly schedule the tasks. Not considering the allocation

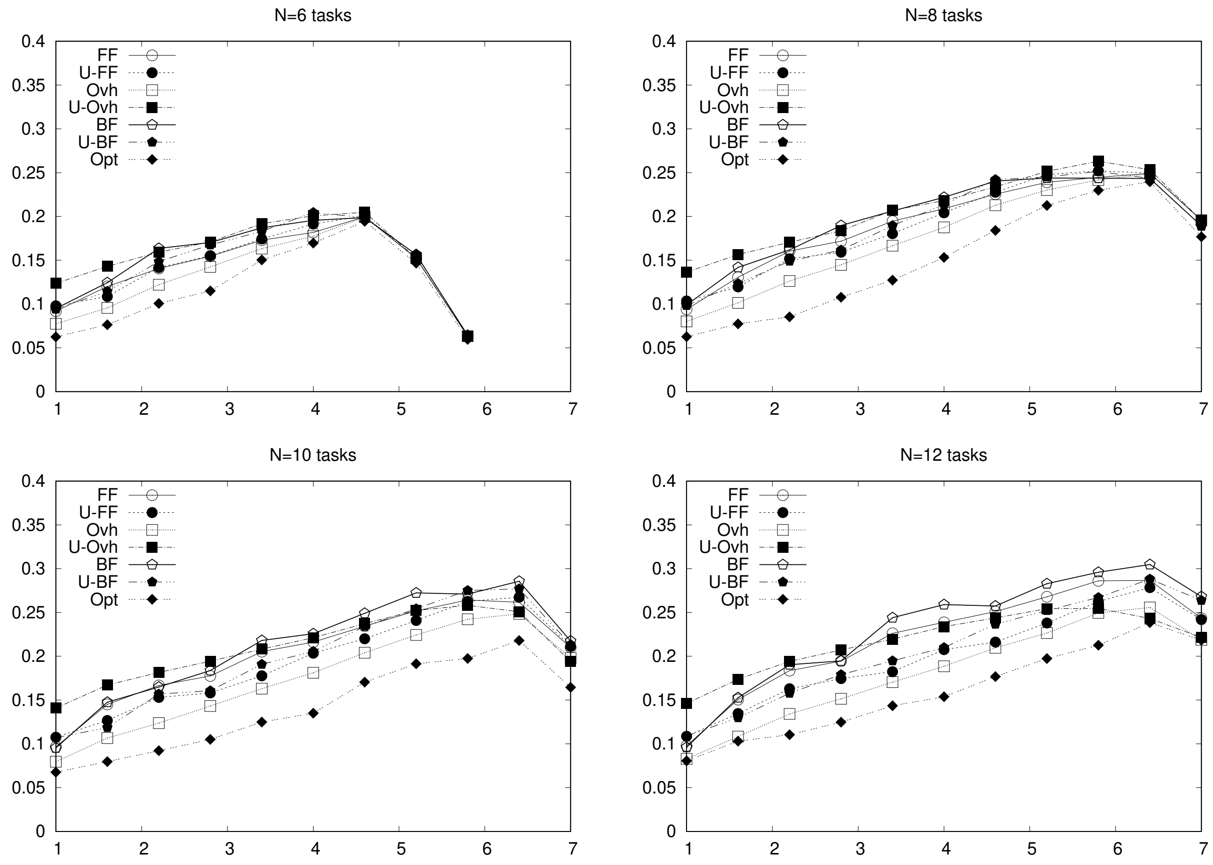


Figure 6: Allocation overhead of the heuristic algorithms on 8 vCPUs.

delay Δ, the $\alpha - 0$ algorithm tends to spread the tasks over all of the vCPUs, resulting in sub-optimal allocations (see previous experiments). All the other algorithms do not suffer this issue. As a second interesting thing, it is possible to notice that the heuristic algorithms perform quite well, resulting in an allocation overhead that is similar to the one produced by the $\alpha - \beta$ algorithm. The heuristics' performance, however, tends to become worse when the number of tasks increases. The third thing to notice is that the $\alpha - \beta$ algorithm appears 2 times in the graphs (the second time with the "with Redesign" suffix, meaning that the optimal server design algorithm has been applied to the tasks partitioning generated by solving the $\alpha - \beta$ problem). Since the $\alpha - \beta$ model does not allow imposing the real "$P \geq P^{min}$" constraint but imposes "$\beta \geq P^{min}/2$", which is more stringent than the original constraint, the MILP results can be pessimistic. The redesign phase addresses this issue by performing the server design with the correct "$P \geq P^{min}$" constraint after the $\alpha - \beta$ model has been used for partitioning the tasks. As a result, the "$\alpha - \beta$ with Redesign" algorithm provides the smallest allocation overhead. It is also possible to notice that the allocation overhead of the "$\alpha - \beta$ with Redesign" algorithm is very close to the allocation overhead of the optimal possible assignment (computed by using the "Constrained Programming" formulation), also shown in the figure. The last thing to notice is that, in this figure, the difference between the various heuristic algorithms (FF, U-FF, and Ovh) is not too relevant (FF tends to perform better with a small number of tasks, and Ovh is always better than the other two heuristics, but the allocation overheads of the three algorithms are similar — and not too large with respect to $\alpha - \beta$).

The results obtained when scheduling larger tasksets on a larger number of vCPUs look more interesting. For example, Figure 8 shows the results obtained for $m = 8$ vCPUs and looking at the figure it is possible to notice how **the curves related to $\alpha - \beta$ algorithms tend to become unstable for high utilizations**. This happens because the constraint on β is too strict and ends up considering most of the task sets unschedulable. By better investigating this issue, it has been found that the $\alpha - \beta$ model is characterized by very low schedulability performance with respect to the other algorithms. Indeed, when U increases **only a few of the 100 task sets per point actually result in a correct server assignment**. Figure 9 plots the *fraction of schedulable tasksets* (fraction of task sets that can be correctly partitioned — and are hence considered schedulable) for the various algorithms, showing this effect. From the figure it is possible to notice that all the algorithms but $\alpha - \beta$ are able to schedule almost all of the tasksets (the fraction of schedulable tasksets is about 1), while for $\alpha - \beta$ the fraction of schedulable tasksets rapidly decreases to very low values. Another interesting thing to notice is that the heuristic algorithms perform well,

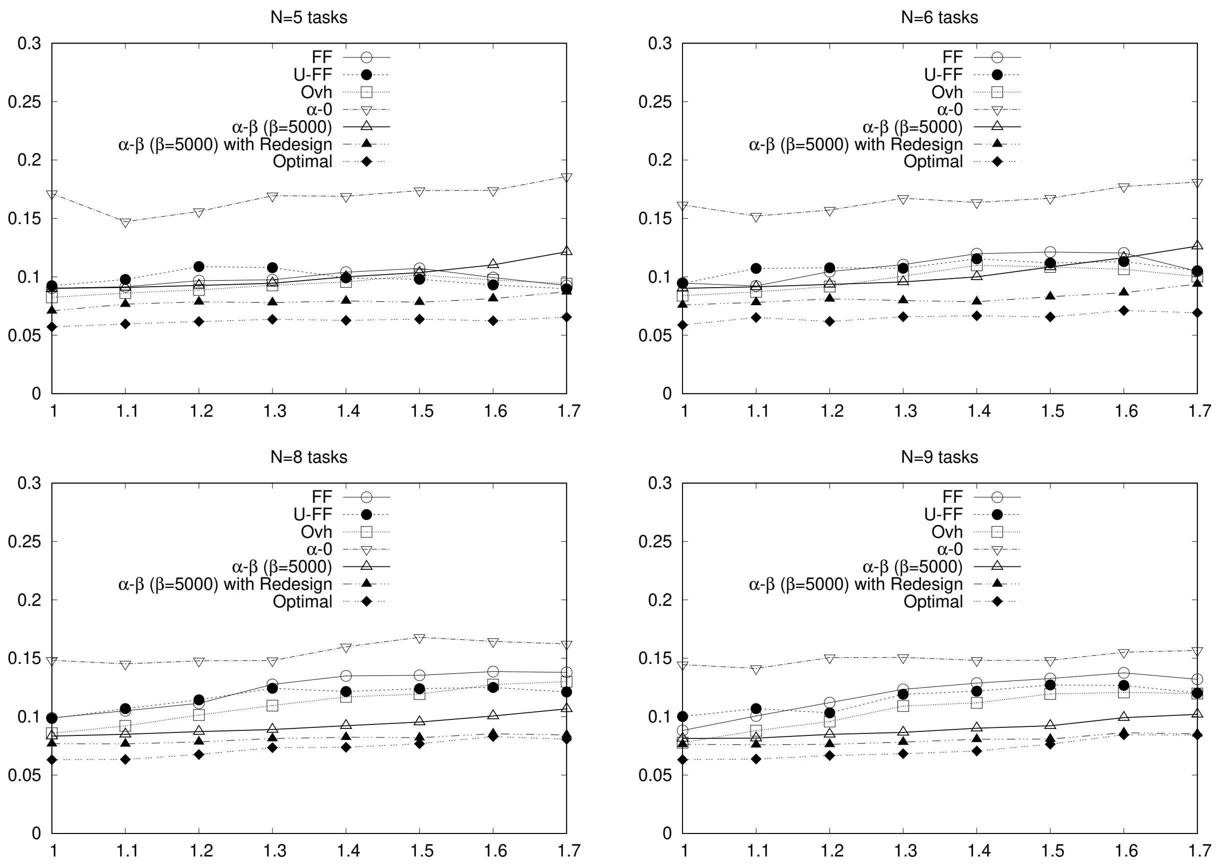


Figure 7: Figure comparing the overhead of the $\alpha - 0$ and $\alpha - \beta$ algorithms with the heuristic.

and Ovh is often better than $\alpha - \beta$ (both in terms of allocation overhead and number of schedulable task sets).

To see how the algorithms' performance is affected by the task sets' sizes, Figures 10 and 11 show the allocation overhead and the fraction of schedulable task sets, respectively, as a function of the number of tasks N (at constant utilization U). The figures confirm that the FF heuristic algorithm performs better for small task sets (and in some cases even performs better than U-FF), but it introduces an allocation overhead that increases with the task set size (and might become worse than most of the other algorithms). The allocation overhead of the $\alpha - 0$ and $\alpha - \beta$ algorithms also increases with the task set size, while the allocation overhead introduced by the Ovh heuristic is generally more constant. Figure 11 confirms that the small allocation overhead shown for $\alpha - \beta$ in Figure 10 might be misleading, as this algorithm is able to partition a much smaller fraction of task sets with respect to all the other algorithms.

After measuring the allocation overhead and the fraction of schedulable tasksets, all the algorithms have been evaluated also considering the partitioning time. These experiments have been performed on two different server machines: an arm64 server based on an 80-core Cavium ThunderX SOC, and an Intel server based on a 40-core Xeon E5-2640 CPU.

Figure 12 reports the maximum partitioning times measured on the arm64 server: the left of the figure displays the maximum between the partitioning times for the tasksets with the same utilization (as a function of the utilization), while the right of the figure displays the maximum between the partitioning times of the tasksets having the same number of tasks (as a function of N). It can be seen that the maximum partitioning times range from a few milliseconds to about $70ms$ for the FF or U-FF heuristic and from about $10ms$ to about $300ms$ for the Ovh and U-Ovh heuristic. The maximum partitioning times for the $\alpha - 0$ partitioning algorithm implemented with GLPK are not visible in the figures, but range from about $100ms$ to more than 2 minutes, while the maximum partitioning times for the $\alpha - \beta$ partitioning with GLPK are all very high and arrive at more than 10 minutes. From these numbers, it is immediately possible to notice how the additional constraints contained in the $\alpha - \beta$ model definitively slow down the MILP solver by a huge amount. The two partitioning algorithms based on GLPK are not usable online in practical scenarios.

The maximum partitioning times measured on the Xeon server, instead, are displayed in Figure 13. This figure confirms the results already observed on the arm64 server (even if the $\alpha - 0$ partitioning algorithm implemented with GLPK is usable in more situations). On this server, it was also possible to solve the $\alpha - 0$ and $\alpha - \beta$ MILP problems using the commercial CPLEX tool, which is highly optimized and can use multiple CPU cores. Using CPLEX allowed to reduce the par-

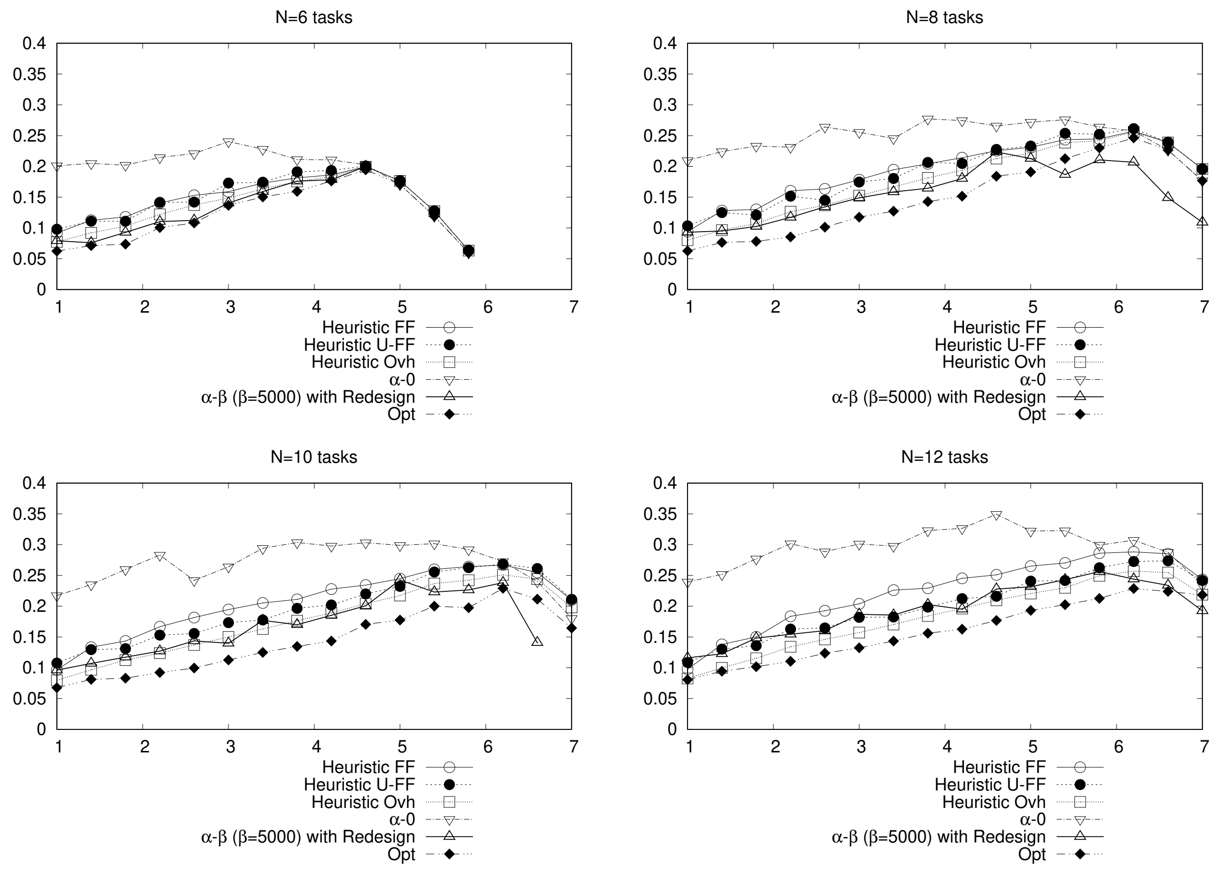


Figure 8: Overhead of various partitioning algorithms with higher utilizations.

titioning times to less than $400ms$. As a result, solving the $\alpha-0$ problem became feasible online for many tasksets. In any case, the heuristic algorithms always performed better than $\alpha - 0$, resulting in a lower allocation overhead, a similar fraction of schedulable task sets, and a lower partitioning time.

6.4. Take-Away Messages

Looking at the presented results, it is possible to draw some important conclusions.

First of all, the experiments showed that the heuristic algorithms should be used for online admission of new real-time applications (and online design of the VM reservations).

Second, if the applications' startup time can tolerate a small additional delay (less than $200ms$), then the Ovh algorithm should be used, otherwise the FF algorithm should be preferred.

Finally, if FF is used and real-time tasks are not dynamically created, then the tasks should be ordered by decreasing utilization before partitioning them.

In all the cases, using MILP-based solutions (such as the ones based on the $\alpha - 0$ and $\alpha - \beta$ models) does not seem to provide relevant advantages. If partitioning and design are performed off-line, then an optimal solution based on Constrained Programming (which provides some small advantages in terms of allocation overhead with respect to the Ovh heuristic, at the cost of a huge increase in the partitioning and design time) can be used.

7. Conclusions and Future Work

This paper analyzed the problem of scheduling real-time tasks in a VM or container. Starting from the observation that in this context partitioned scheduling is more suitable than global scheduling, various partitioning algorithms have been presented (some algorithms based on a MILP formulation, some heuristics, and an optimal algorithm to be used as a reference). The algorithms have been compared considering their allocation overhead, the percentage of real-time tasksets that they are able to accept, and their running time. The results indicate that the proposed heuristic algorithms should be used for online admission control in a large set of different virtual environments, such as real-time clouds or similar; in particular, a heuristic based on First Fit is to be preferred if the running time has to be reduced to the minimum, while the new "Opt" heuristic can reduce the allocation overhead (and accept a larger fraction of tasksets) if the system can tolerate a small delay (less than $200ms$) in admitting and starting a real-time application.

As a future work, these heuristics will be integrated into a cloud management software such as Kubernetes. Moreover, the analysis will be extended to support non-identical multicore

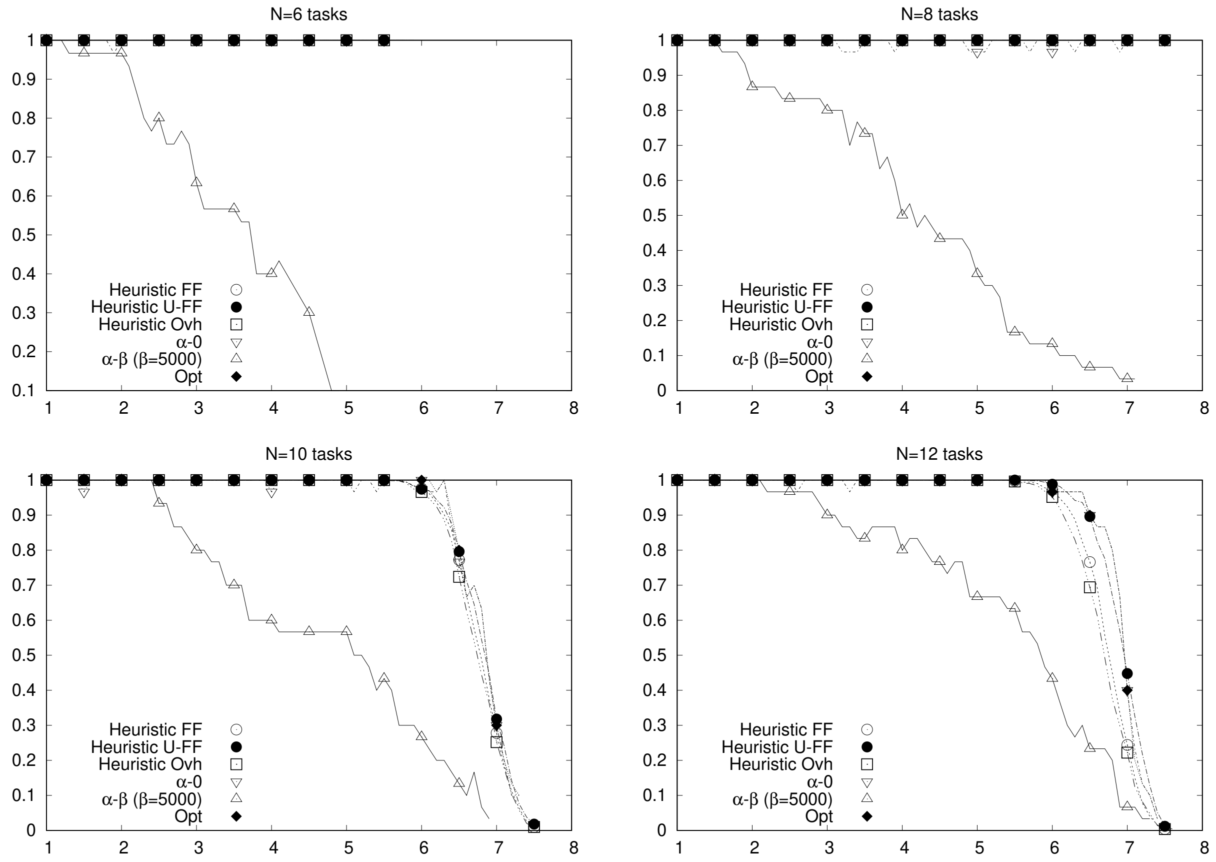


Figure 9: Fraction of schedulable tasksets for the various partitioning algorithms.

architectures (such as, for example, the ARM big.LITTLE architecture) and parallel tasks.

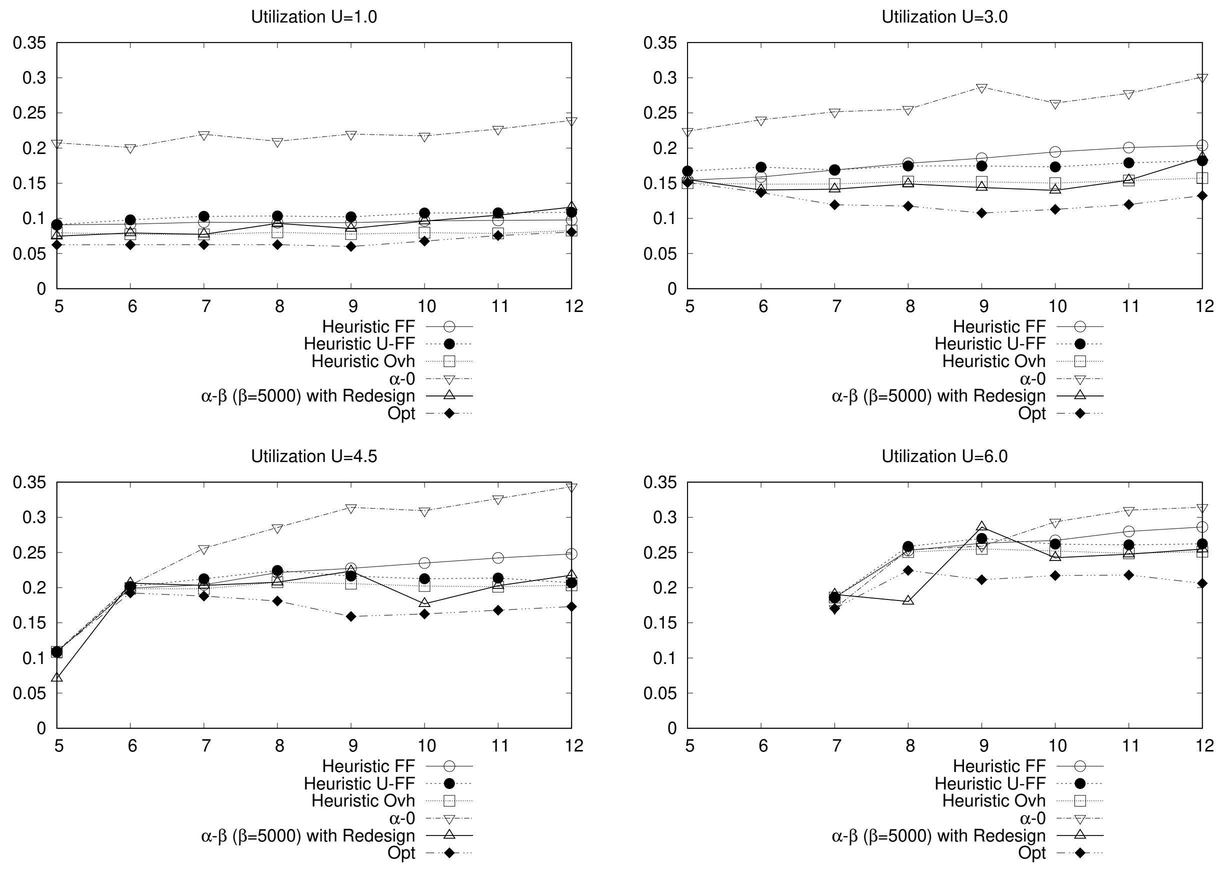


Figure 10: Overhead of various partitioning algorithms as a function of the number of tasks.

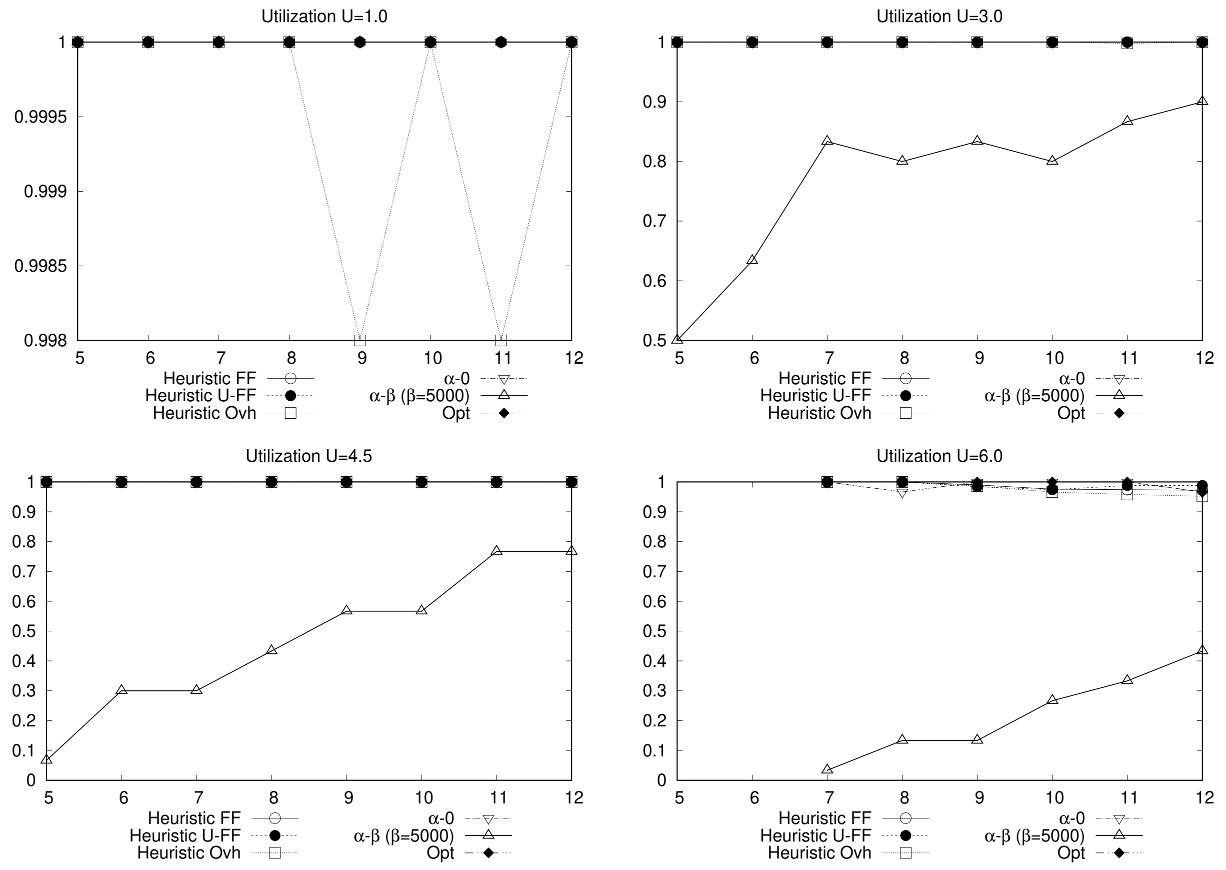


Figure 11: Fraction of schedulable tasksets for various partitioning algorithms as a function of the number of tasks.

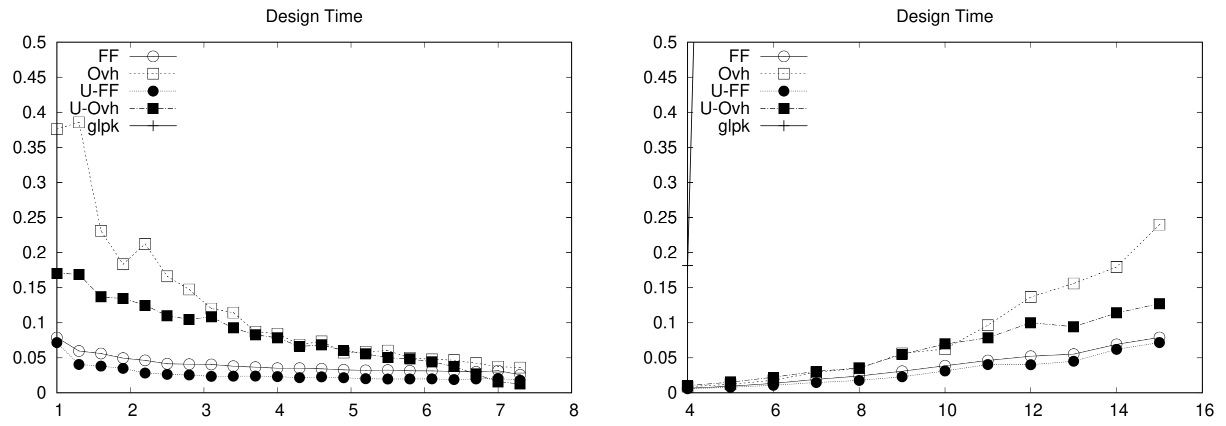
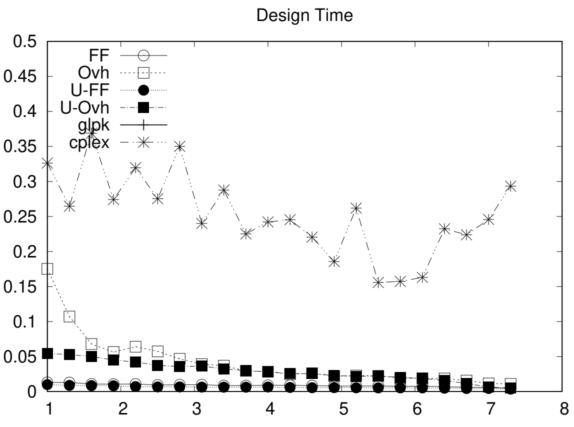


Figure 12: Maximum partitioning times measured on the ARM server, as a function of the taskset utilization (on the left) and of the number of tasks (on the right).



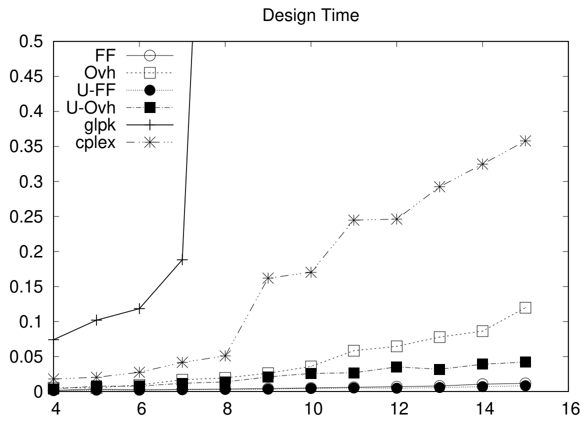


Figure 13: Maximum partitioning times measured on the Xeon server, as a function of the taskset utilization (on the left) and of the number of tasks (on the right).

References

[1] S.-P. Chuah, C. Yuen, N.-M. Cheung, Cloud gaming: a green solution to massive multiplayer online games, IEEE Wireless Communications 21 (4) (2014) 78–87.

[2] D. Meiländer, S. Gorlatch, Modeling the scalability of real-time online interactive applications on clouds, Future Generation Computer Systems 86 (2018) 1019–1031.

[3] M. Ghobaei-Arani, R. Khorsand, M. Ramezanpour, An autonomous resource provisioning framework for massively multiplayer online games in cloud environment, Journal of Network and Computer Applications 142 (2019) 76–97.

[4] M. Mao, M. Humphrey, Auto-scaling to minimize cost and meet application deadlines in cloud workflows, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.

[5] S. M. Salman, V. Struhar, A. V. Papadopoulos, M. Behnam, T. Nolte, Fogification of industrial robotic systems: Research challenges, in: Proceedings of the Workshop on Fog Computing and the IoT, IoT-Fog '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 41–45.

[6] V.-D. Balteanu, A. Neculai, C. Negru, F. Pop, A. Stoica, Near real-time scheduling in cloud-edge platforms, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, SAC '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1264–1271.

[7] N. Auluck, A. Azim, K. Fizza, Improving the schedulability of real-time tasks using fog computing, IEEE Transactions on Services Computing (2019) 1–1.

[8] M. S. Shaik, V. Struhár, Z. Bakhshi, V.-L. Dao, N. Desai, A. V. Papadopoulos, T. Nolte, V. Karagiannis, S. Schulte, A. Venito, G. Fohler, Enabling fog-based industrial robotics systems, in: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Vol. 1, 2020, pp. 61–68.

[9] L. Abeni, D. Faggioli, An experimental analysis of the xen and kvm latencies, in: Proceedings of the IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC), 2019, pp. 18–26.

[10] L. Abeni, D. Faggioli, Using xen and KVM as real-time hypervisors, Journal of Systems Architecture 106 (2020) 101709.

[11] H. Li, X. Xu, J. Ren, Y. Dong, Acrn: A big little hypervisor for iot development, in: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, Association for Computing Machinery, New York, NY, USA, 2019, p. 31–44.

[12] 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.

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

[14] L. Abeni, A. Biondi, E. Bini, Hierarchical scheduling of real-time tasks over linux-based virtual machines, Journal of Systems and Software 149 (2019) 234 – 249.

[15] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. Phan, I. Lee, O. Sokolsky, Rt-openstack: Cpu resource management for real-time cloud computing, in: Proceedings of the 8th International Conference on Cloud Computing, IEEE, 2015, pp. 179–186.

[16] V. Struhár, M. Behnam, M. Ashjaei, A. V. Papadopoulos, Real-Time Containers: A Survey, in: A. Cervin, Y. Yang (Eds.), 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020), Vol. 80 of OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2020, pp. 7:1–7:9.

[17] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, A. V. Papadopoulos, React: Enabling real-time container orchestration, in: 2021 ETFA–IEEE 26th International Conference on Emerging Technologies and Factory Automation, 2021, pp. 1–8.

[18] T. P. Baker, M. Cirinei, Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks, in: Proceedings of the International Conference On Principles Of Distributed Systems (OPODIS), 2007, pp. 62–75.

[19] N. Guan, Z. Gu, Q. Deng, S. Gao, G. Yu, Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking, in: Proceeding of the IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems, 2007, pp. 263–272.

[20] V. Bonifaci, A. Marchetti-Spaccamela, Feasibility analysis of sporadic real-time multiprocessor task systems, Algorithmica 63 (4) (2012) 763–780.

[21] A. Burmyakov, E. Bini, C.-G. Lee, Towards a tractable exact test for global multiprocessor fixed priority scheduling, IEEE Transactions on Computers (2022) 1–1.

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

[23] T. P. Baker, An analysis of edf schedulability on a multiprocessor, IEEE transactions on parallel and distributed systems 16 (8) (2005) 760–768.

[24] T. P. Baker, Multiprocessor edf and deadline monotonic schedulability analysis, in: Proceeding of the 24th IEEE Real-Time Systems Symposium, 2003, pp. 120–129.

[25] S. K. Baruah, J. Goossens, Rate-monotonic scheduling on uniform multiprocessors, IEEE transactions on computers 52 (7) (2003) 966–970.

[26] G. Nelissen, V. Berten, V. Nélis, J. Goossens, D. Milojevic, U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks, in: 2012 24th Euromicro Conference on Real-Time Systems, 2012, pp. 13–23.

[27] P. Regnier, G. Lima, E. Massa, G. Levin, S. Brandt, Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor, in: 2011 IEEE 32nd Real-Time Systems Symposium, 2011, pp. 104–115.

[28] G. Levin, S. Funk, C. Sadowski, I. Pye, S. Brandt, Dp-fair: A simple model for understanding optimal multiprocessor scheduling, in: 2010 22nd Euromicro Conference on Real-Time Systems, 2010, pp. 3–13.

[29] B. Andersson, E. Tovar, Multiprocessor scheduling with few preemptions, in: 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06), 2006, pp. 322–334.

[30] H. Cho, B. Ravindran, E. D. Jensen, An optimal real-time scheduling algorithm for multiprocessors, in: 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06), 2006, pp. 101–110.

[31] S. K. Baruah, N. K. Cohen, C. G. Plaxton, D. A. Varvel, Proportionate progress: A notion of fairness in resource allocation, Algorithmica 15 (6) (1996) 600–625.

[32] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, R. L. Graham, Worst-case performance bounds for simple one-dimensional packing algorithms, SIAM Journal on Computing 3 (4) (1974) 299–325.

[33] D.-I. Oh, T. P. Baker, Utilization bounds for n-processor rate monotone scheduling with static processor assignment, Real-Time Systems 15 (2) (1998) 183–192.

[34] 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.

[35] J. M. Lopez, M. Garcia, J. L. Diaz, D. F. Garcia, Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems, in: 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden, 2000, pp. 25–33.

[36] J. Lin, A. Srivatsa, A. Gerstlauer, B. L. Evans, Heterogeneous multiprocessor mapping for real-time streaming systems, in: 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2011, pp. 1605–1608.

[37] S. Baruah, An ILP representation of a DAG scheduling problem, Real-Time Systems 58 (2022) 85—102.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] I. Shin, A. Easwaran, I. Lee, Hierarchical scheduling framework for virtual clustering of multiprocessors, in: 2008 Euromicro Conference on Real-Time Systems, 2008, pp. 181–190.

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

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] L. Abeni, A. Balsini, T. Cucinotta, Container-based real-time scheduling in the linux kernel, SIGBED Review 16 (3) (2019) 33–38.

[51] S. Funk, J. Goossens, S. Baruah, On-line scheduling on uniform multiprocessors, in: Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420), 2001, pp. 183–192.

[52] G. Lipari, S. Baruah, Greedy reclamation of unused bandwidth in constant-bandwidth servers, in: Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000, 2000, pp. 193–200.

[53] L. Abeni, G. Lipari, A. Parri, Y. Sun, Multicore cpu reclaiming: Parallel or sequential?, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 1877–1884.

[54] J. L. Lorente, G. Lipari, E. Bini, A hierarchical scheduling model for component-based real-time systems, in: Proceedings 20th IEEE International Parallel Distributed Processing Symposium, Rhodes Island, Greece, 2006, pp. 8 pp.–.

[55] J. C. Palencia, M. G. Harbour, J. J. Gutiérrez, J. M. Rivas, Response-time analysis in hierarchically-scheduled time-partitioned distributed systems, IEEE Transactions on Parallel and Distributed Systems 28 (7) (2017) 2017–2030.

[56] A. Amurrio, E. Azketa, J. J. Gutierrez, M. Aldea, M. G. Harbour, Response-time analysis of multipath flows in hierarchically-scheduled time-partitioned distributed real-time systems, IEEE Access 8 (2020) 196700–196711.

[57] 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.

[58] 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.

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

[60] 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.

[61] 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.

[62] Y. Sun, M. Di Natale, Pessimism in multicore global schedulability analysis, Journal of Systems Architecture 97 (2019) 142–152.

[63] 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.

[64] S. Boyd, S.-J. Kim, L. Vandenberghe, A. Hassibi, A tutorial on geometric programming, Optimization and engineering 8 (1) (2007) 67–127.

[65] M. Grant, S. Boyd, Y. Ye, Disciplined Convex Programming, Springer US, Boston, MA, 2006, pp. 155–210.

[66] A. Biondi, A. Melani, M. Bertogna, G. Buttazzo, Optimal design for reservation servers under shared resources, in: 2014 26th Euromicro Conference on Real-Time Systems, 2014, pp. 153–164.