Designing VM Schedulers for Embedded Real-Time Applications

Alejandro Masrur[1], Thomas Pfeuffer[1], Martin Geier[1], Sebastian Drössler[2] and Samarjit Chakraborty[1]
[1]Institute for Real-Time Computer Systems, TU Munich, Germany
[2]ReliaTec GmbH, Garching, Germany
{a.masrur, samarjit}@tum.de, s.droessler@reliatec.de

ABSTRACT

Virtual Machines (VMs) allow for platform-independent software development and their use in embedded systems is increasing. In particular, VMs are rewarding in the context of mixed-criticality applications to provide isolation between critical and non-critical tasks running on the same processor. In this paper, we study the design of a real-time system based on a VM monitor/hypervisor that supports multiple VMs/domains. Since each VM in the system runs several real-time tasks, scheduling the VMs leads to a hierarchical scheduling problem. So far, most published techniques for analyzing hierarchical scheduling deal with the schedulability problem, i.e., for a given hierarchical scheduler, testing whether a set of real-time tasks meet their deadlines. In this paper, we are rather concerned with the synthesis of hierarchical/VM schedulers; that is, how to design a scheduler such that all real-time tasks running on the different VMs meet their deadlines. We consider a setup where the tasks are scheduled on multiple VMs under fixed priorities according to the Deadline Monotonic (DM) policy. The VMs are scheduled under fixed priorities on a Rate Monotonic (RM) basis using one or more processors. A partitioned scheduling of VMs is considered, i.e., VMs are not allowed to migrate from one processor to the other. In this context, we propose a method for selecting optimum time slices and periods for each VM in the system. Our goal is to configure the VM scheduler such that not only all tasks are schedulable but also the minimum possible resources are used. Finally, to illustrate the proposed design technique, we present a case study based on automotive control applications.

Categories and Subject Descriptors: D.4.7 [Software]: Operating Systems—*Real-time systems and embedded systems*

General Terms: Theory, Design, Algorithms

Keywords: Real-Time Scheduling, Virtual Machines, Embedded Software

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

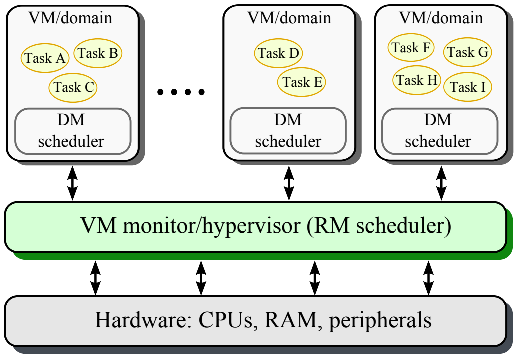

Figure 1: VM architecture under consideration

1. INTRODUCTION

In the automotive domain, for example, different functionalities or applications are traditionally implemented on different electronic control units (ECUs). This has led to a large number of ECUs in modern cars, which complicates wiring and increases cost. As a result, there is a strong focus on integrating multiple applications on a single ECU. When many applications run concurrently on the same hardware, they interact with each other as they share processing units, memory, I/O devices, etc. The automotive software is very complex and hard to test and validate ([19], [7]). Thus, if multiple automotive applications run on the same ECU, it is difficult to guarantee that an error in one application will not affect the others.

A way to prevent errors from propagating is to implement some sort of isolation between applications. Most modern processors offer features such as user/supervisor modes and MMU (Memory Management Unit). These already allow for some isolation, particularly, between operating system (OS) and user applications. However, an effective separation between single user-level applications is still required, especially, if these have different levels of criticality. In our previous example from the automotive domain, we would like to avoid that general-purpose applications such as navigation and multimedia interfere with safety-/time-critical applications such as airbag control and brake system.

A *virtualization* layer between hardware and software as illustrated in Fig. 1 can be used to provide isolation between user-level applications. Such a virtualization layer is normally referred to as *virtual machine monitor* or *hypervisor*. The OS and the user applications now run on *virtual machines* or *domains* (VMs) and not directly on the hardware. The VM monitor traps all requests directed to shared resources (like processors, I/O devices, etc.) and administers access to them by scheduling the VMs. This way, the propagation of an error is restricted to a VM and cannot affect the whole system.

Scope and contributions of this paper: The use of VMs in embedded real-time applications results in a two-level hierarchical scheduling problem. Here, the first-level or VM scheduler assigns CPU time to VMs, whereas the second-level or task scheduler administers time within a VM. In order to guarantee all timing constraints, a careful configuration of the VM scheduler is required.

Some techniques for analyzing hierarchical scheduling under different scheduling policies are already known in the literature. However, most of them focus on the *schedulability analysis* of a fixed first-level/VM scheduler and not on *synthesizing* such a scheduler. To the best of our knowledge, there exists no previous work on how to design VM schedulers for real-time applications. Moreover, existing schedulability conditions derived for hierarchical scheduling do not extend to this scenario in a straightforward manner.

In this paper, we study VMs that are scheduled under fixed priorities and in a partitioned manner on multiple processors, i.e., VMs are not allowed to migrate from a processor to another. (Since VM migration is associated with a non-negligible overhead, we believe this to be less practical in embedded real-time systems where resources are limited.) Further, we consider that multiple real-time tasks run on each VM also on a fixed-priority basis. Although the technique presented in this paper remains valid for any priority assignment, we assume that real-time tasks are assigned priorities according to Deadline Monotonic (DM) whereas VMs are scheduled under Rate Monotonic (RM) – see Fig. 1.

We propose a method for selecting efficient time partitions for VMs such that all timing constraints are met and the minimum possible resources are used. In principle, partitioning time between VMs consists of finding periods and time slices for each VM in the system. The period assigned to a VM determines its activation rate, whereas the time slice determines the amount of CPU time that the VM is allowed to utilize at each activation.

Clearly, the period assigned to a VM is dominated by the smallest deadline among all tasks running on the VM – a longer period leads to deadline misses, a shorter period produces unnecessary context switches. However, selecting optimum time slices for VMs is not as straightforward and requires a non-trivial analysis.

Therefore, we first formulate the timing requirements for a VM scheduled under a fixed-priority policy. These are mainly dictated by the shortest deadline on the VM and by the priority of the VM in the system. We then derive schedulability conditions for the tasks running on that VM and compute an estimate of the necessary time slice. This estimate is used as initial value for the time slice, which can then be improved towards the optimum in subsequent steps. We show that the resulting method has pseudo-polynomial complexity. Hence, we will be able to find an optimum time slice in a limited number of iterations.

In addition, we present a case study in which we use simplified versions of automotive applications such as engine management and electronic stability control to demonstrate our design technique. Further we compare two possible design cases with respect to their timing behavior. The first case consists of scheduling each task of an application on a separate VM. The second case deals with scheduling an entire application (i.e., multiple tasks) on the same VM. Scheduling single tasks on separated VMs is less relevant from a practical point of view. However, this allows for a higher processor utilization and it is interesting for comparison purposes.

The remainder of the paper is organized as follows. First, we give an overview of related work in Section 2 and introduce models and notation used along the paper in Section 3. In Section 4, we analyze the minimum requirements that a VM has to fulfill to guarantee the schedulability of tasks running on it. Further, Section 5 presents our design technique for VM schedulers. We finally discuss a set of experiments on the basis of our case study in Section 6, whereas Section 7 summarizes the contributions presented in this paper.

2. RELATED WORK

The analysis of hierarchical scheduling has attracted a lot of attention in the literature. As a result, there are already a number of techniques for analyzing multilevel scheduling (i.e., with more than one scheduler/scheduling level) under different scheduling policies.

In [6], Deng and Liu presented an analysis of a two-level hierarchical scheduling using a first-level scheduler based on EDF (Earliest Deadline First). They considered tasks to be sporadic and scheduled under different algorithms. However, since they used utilization bounds, some pessimism is incurred in deriving schedulability conditions.

Kuo and Li [12] analyzed the hierarchical scheduling based on RM and EDF, for which they assumed periodic tasks with deadlines equal to periods. In [17], Mok et al. proposed a bounded-delay processing supply for hierarchical scheduling, for which they also assumed periodic tasks with deadlines equal to periods. In contrast to [12] and [17], in this paper, we consider the more general case of sporadic tasks with deadlines that may be less than the minimum inter-arrival time between two jobs.

Lipari et al. [15] presented a framework called PShED (Processor Sharing with Earliest Deadline First). Here, servers (i.e., VMs in our case) are allowed to update their urgency according to the deadline of the currently running task. Since the VM scheduler used in this paper is based on the current implementation of Xen, it is not possible to change the priority of a VM according to the task that is currently running on it. However, in order to achieve isolation between applications on the different VMs, it is indeed more meaningful to separate the priority of a VM from that of its currently running task.

Shin and Lee [21] presented the periodic processing supply model. Here, each VM receives a maximum fixed amount of CPU time on a periodic basis. The periodic supply model only guarantees that a VM executes once within its period and, in worst case, the VM may finish executing towards the end of its period. Both fixed- and dynamic-priority VM schedulers can be described by the periodic supply model. However, assuming that the VM may finish executing towards the end of its period introduces additional (undesired) pessimism, particularly, when VMs are scheduled under fixed priorities.

Our analysis technique has similarities to that of Davis and Burns [5]. That is, we also consider that both the tasks as well as the VMs are scheduled under fixed priorities and use the worst-case response time analysis. However, approaching this problem from the synthesis perspective, i.e., where suitable time slice and period need to be configured for each VM in the system, leads us to different issues (than those studied in [5]) as described later.

The hierarchical scheduling considered in this paper can also be described using the EDP (Explicit Deadline Periodic) processing supply model [8]. Nevertheless, in contrast to [8], we are concerned with the case where periods and time slices need to be selected for every VM in the system, which makes additional analysis be necessary.

More recently, Shin and Lee [22] presented a compositional method for analyzing EDF and RM hierarchical scheduling, for which they considered periodic tasks with deadlines equal to periods. In [20], Shin et al. proposed an analysis of multiprocessor scheduling based on hierarchical scheduling and processor clusters, which uses the periodic processing supply model. This work was continued later by Easwaran et al. [9].

The practical use of VMs has also been intensively studied in the literature. However, most related works in this area focus on analyzing the performance and fairness of VM scheduling policies [10, 3, 18]. A few other works study different scheduling techniques that take characteristics of VMs into account. For example, scheduling techniques for VMs with intensive I/O demand and throughput were studied in [11] and [23], respectively.

In a previous work [16], we studied the real-time behavior of the Xen hypervisor (an available open-source VM monitor [1]). There we proposed and implemented a fixed-priority variant of Xen's SEDF (Simple EDF) scheduler. The proposed scheduler in [16] distinguishes between real-time and non-real-time VMs/domains and is used in the context of the case study presented in this paper.

3. MODELS AND NOTATION

Here, we introduce both the models and the notation we use. For ease of exposition, we will define some parameters and variables later as it becomes necessary along the paper.

We denote by $\mathbf{T}$ a set of sporadic, independent, and fully preemptive real-time tasks. Each task T_i in $\mathbf{T}$ is characterized by its relative deadline d_i, its worst-case execution time e_i and its minimum inter-release time p_i, i.e., the minimum distance between two consecutive jobs of T_i. For all tasks, we assume that relative deadlines d_i are less than or equal to the respective minimum inter-release times p_i.

As stated previously, we consider the design of real-time systems based on a VM monitor/hypervisor that supports multiple VMs/domains. The tasks in $\mathbf{T}$ are allocated to several VMs, which then run on one or more processors/cores in a partitioned manner. This results from the use of an available VM monitor such as Xen, whose VM scheduler – the standard SEDF and the extended fixed-priority version used in this paper [16] – allocates VMs to fixed cores and does not allow them to migrate.

Further, $\mathbf{V}$ denotes the set of all VMs/domains V_l in the system. Every V_l is assigned a time slice s_l^x and a period p_l^x – all parameters that are related to VMs will be represented with the upper index x. The VM scheduler then allows every VM to run a maximum amount of time s_l^x every p_l^x time units (clearly, $s_l^x \leq p_l^x$ must hold for all VMs). Later, we use $\mathbf{T}^l \subset \mathbf{T}$ to denote the subset of real-time tasks running on a specific V_l.

VMs are scheduled under the RM and tasks under the DM policy resulting in a DM over RM hierarchical scheduling. Without loss of generality, we assume that tasks in $\mathbf{T}$ and $\mathbf{T}^l$ as well as VMs in $\mathbf{V}$ are sorted according to decreasing priorities. This way, a task T_i has higher priority (i.e., shorter deadline) than the task T_{i+1}. Similarly, V_l has higher priority (i.e., shorter period) than V_{l+1}.

In the next sections, we will analyze the worst-case response time of real-time tasks under this constellation. Further, we derive a set of equations that help configuring time slices and periods for each VM in the system.

4. MINIMUM REQUIREMENTS FOR A VM

The concept of *starvation length* has been used in the literature to designate the largest time interval without processing supply, i.e., the worst-case waiting time between two consecutive executions of a VM [22, 20, 9].

In order to simplify the analysis in this paper, we introduce the term *execution length* to denominate the largest time interval that it takes a VM to execute for s_l^x time units (i.e., to execute for an amount of time equal to its assigned time slice). The execution length clearly depends on the scheduling of VMs and has direct impact on the schedulability of real-time tasks running on them. Hence, determining the execution length is the first step towards a schedulability analysis.

As stated above, we consider that VMs are scheduled according to fixed priorities under RM. In addition, let us assume that scheduling all VMs is feasible. In other words, every V_l can finish executing its assigned time slice s_l^x within p_l^x time units from its release. The execution length of V_l is depicted in Fig. 2 and results from considering the following two conditions:

1. V_l can first finish executing s_l^x time units as early as possible.
2. All higher priority VMs are then released together with the next execution of V_l.

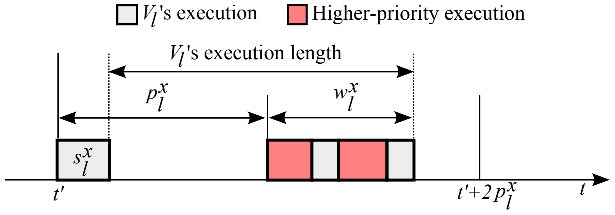

Figure 2: V_l's execution length

The first condition yields the largest time interval without processing supply between two releases of V_l (i.e., $p_l^x - s_l^x$). The second condition leads to the maximum higher-priority interference during the next execution of V_l. This maximum interference reflects in that the next instance of V_l takes the longest to execute s_l^x time units. The point in time at which V_l finishes its execution can be found using known worst-case response time analysis [13, 2]:

$$t^{(c+1)} = s_l^x + \sum_{j=1}^{l-1} \left\lceil \frac{t^{(c)}}{p_j^x} \right\rceil s_j^x. \qquad (1)$$

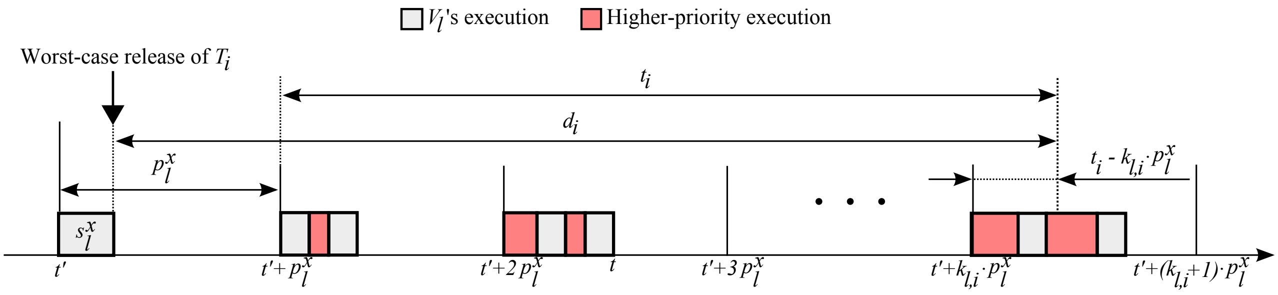

Figure 3: Schedulability condition for a task T_i running on V_l

Recall that VMs in $\mathbf{V}$ are sorted according to decreasing priority (i.e., according to increasing/non-decreasing periods under the RM policy). So, the second term of Eq. (1) corresponds to all higher-priority VMs in the system. This equation can be solved iteratively starting from $t^{(1)} = s_l^x$ and until $t^{(c+1)} = t^{(c)}$ is satisfied for some $c \geq 1$. This value of $t^{(c+1)}$ then is V_l's worst-case response time which we denote here by w_l^x – recall that all VM parameters are denoted with the upper index x. As a result, V_l's execution length L_l^x is given by:

$$L_l^x = p_l^x - s_l^x + w_l^x. \tag{2}$$

Further, we denote by $d_{l,min} = \min_{i=1}^{|\mathbf{T}^l|}(d_i)$ the smallest deadline in $\mathbf{T}^l$, where $|\mathbf{T}^l|$ represents the number of tasks in $\mathbf{T}^l$. The worst-case execution time of the task with $d_{l,min}$ is denoted by $e_{l,min}$ – note that this is not necessarily the minimum worst-case execution time in $\mathbf{T}^l$. The task with $d_{l,min}$ has the highest priority on V_l and executes without interruption as soon as V_l is activated. Let us now assume that s_l^x is at least equal to $e_{l,min}$:

$$e_{l,min} \leq s_l^x. \tag{3}$$

So, in order that V_l can always meet $d_{l,min}$, its execution length L_l^x must be less than or equal to $d_{l,min}$:

$$p_l^x - s_l^x + w_l^x \leq d_{l,min}. \tag{4}$$

From Eq. (4), it can be seen that choosing s_l^x to be smaller than $e_{l,min}$ forces us to reduce p_l^x. Otherwise, V_l cannot meet $d_{l,min}$ in the worst case. However, a shorter p_l^x can potentially increase the number of context switches due to V_l. For this reason, it is meaningful to select a value of s_l^x according to Eq. (3).

4.1 Schedulability on a VM

Most embedded applications consist of a set of several real-time tasks. As a consequence, it is reasonable to allocate all tasks composing such an application to a single VM. To isolate the different applications from each other, these should run on different VMs. From now on, we assume that all tasks T_i in $\mathbf{T}^l$ (i.e., all tasks running on V_l) belong to the same application.

Further, we assume that all tasks belonging to an application are schedulable on one processor. This is true in many domains such as automotive where control applications typically run on separated single-core ECUs. As a consequence, if a V_l runs alone on one processor and $s_l^x = p_l^x$ holds, the whole task set $\mathbf{T}^l$ is schedulable. That is, the worst-case response time of every $T_i \in \mathbf{T}^l$ is less than or equal to its deadline d_i.

Let us denote by $\hat{w}_i$ the worst-case execution demand of a task $T_i \in \mathbf{T}^l$ within d_i time units:

$$\hat{w}_i = e_i + \sum_{j=1}^{i-1} \left\lceil \frac{d_i}{p_j} \right\rceil e_j. \tag{5}$$

Recall that tasks in $\mathbf{T}^l$ are sorted according to decreasing priority (i.e., according to increasing/non-decreasing deadlines under the DM policy). So, the second term of Eq. (5) corresponds to all higher-priority tasks on V_l.

In what follows, we assume that the worst-case execution demand within d_i time units is less than or equal to d_i for every $T_i \in \mathbf{T}^l$, i.e., $\hat{w}_i \leq d_i$ holds. Otherwise, it will not be possible to schedule the task on a VM.

Now, for a task T_i to meet its deadline on V_l, the following inequality must hold – see Fig. 3:

$$k_{l,i} \cdot s_l^x + \min\bigl(s_l^x, \alpha_l^x(t_i - k_{l,i} \cdot p_l^x)\bigr) \geq \hat{w}_i, \tag{6}$$

where t_i is equal to $d_i - (p_l^x - s_l^x)$ and $k_{l,i}$ is computed by $\lfloor \frac{t_i}{p_l^x} \rfloor$. The function $\alpha_l^x(t)$ returns the amount of time that V_l is able to run in a time interval of length t. This function assumes the critical instant for V_l, i.e., V_l is released together with all higher-priority VMs at the beginning of the interval of length t.

In order to understand Eq. (6), let us first suppose that $\min\bigl(s_l^x, \alpha_l^x(t_i - k_{l,i} \cdot p_l^x)\bigr) = 0$ such that we have:

$$k_{l,i} \cdot s_l^x \geq \hat{w}_i.$$

In worst case, the event triggering task T_i arrives together with all higher-priority events (triggering all higher-priority tasks on V_l) exactly after V_l runs out of time. In addition, if V_l could execute its whole time slice immediately after it was released, the waiting time until the next time slice is the largest possible, i.e., $p_l^x - s_l^x$. Because T_i is released together with all higher priority tasks on V_l, the execution demand within its deadline is going to be $\hat{w}_i$, i.e., the highest execution demand possible. As mentioned before, we assume that $\hat{w}_i \leq d_i$ holds for all T_i in the system. As a consequence, T_i can meet its deadline if V_l can execute $\hat{w}_i$ time units before d_i expires – see Fig. 3. So, for $t_i = d_i - (p_l^x - s_l^x)$, V_l executes $k_{l,i} = \lfloor \frac{t_i}{p_l^x} \rfloor$ times before d_i. If $k_{l,i} \cdot s_l^x \geq \hat{w}_i$ holds, T_i is feasible on V_l.

V_l is executed $k_{l,i}$ times within t_i. Therefore, the time interval $t_i - k_{l,i} \cdot p_l^x$ is the remainder of t_i after $k_{l,i}$ executions of V_l. The term $\min\big(s_l^x, \alpha_l^x(t_i - k_{l,i} \cdot p_l^x)\big)$ in Eq. (6) represents V_l's additional execution time in $t_i - k_{l,i} \cdot p_l^x$ assuming the worst-case situation (i.e., V_l is released together with all higher-priority VMs). Clearly, V_l cannot execute for longer than s_l^x time units in a period p_l^x. So, since $t_i - k_{l,i} \cdot p_l^x < p_l^x$ holds, V_l's additional execution time in this interval is at maximum equal to s_l^x depending on its worst-case response time. Now, $\alpha_l^x(t)$ returns the maximum value of a variable e_l^x for which the following holds:

$$t^{(c+1)} = e_l^x + \sum_{j=1}^{l-1} \left\lceil \frac{t^{(c)}}{p_j^x} \right\rceil s_j^x. \tag{7}$$

Recall that VMs are sorted according to decreasing priority. This equation can be solved iteratively starting from $t^{(1)} = e_l^x$ and until $t^{(c+1)} = t^{(c)}$ is satisfied for some $c \geq 1$. This value of $t^{(c+1)}$ is then denoted by t_l^x for which the second condition must hold:

$$t_l^x \leq t_i - k_{l,i} \cdot p_l^x. \tag{8}$$

In other words, e_l^x is V_l's largest amount of execution time leading to a worst-case response time (i.e., considering maximum interference by higher-priority VMs) that is less than or equal to $t_i - k_{l,i} \cdot p_l^x$ (the remaining time after $k_{l,i}$ complete executions of V_l). Since $t_i - k_{l,i} \cdot p_l^x < p_l^x$ holds and the V_l can only execute a maximum of s_l^x time units (i.e., $e_l^x \leq s_l^x$), we will always able to find a value of e_l^x in a finite number of iterations.

5. DESIGNING THE VM SCHEDULER

So far, we have analyzed the minimal requirements for a VM and the schedulability of a real-time task running on it. In this section, we focus on selecting time slices and periods for each VM such that all deadlines can be guaranteed.

As VMs are scheduled under RM, p_l^x determines the priority of a V_l. Further, we know that p_l^x must satisfy Eq. (4) for V_l to schedule the task with the minimum deadline $d_{l,min}$. Hence, the VM executing the task with the minimum $d_{l,min}$ in the system is the one with the highest priority.

The design procedure is illustrated in Fig. 4. We start selecting the period and the time slice for the VM with the highest priority. Then we continue with the lower priority VMs in order of decreasing priorities. This is simply because we need to know the parameters of higher-priority VMs to be able to compute the worst-case response time of a lower-priority VM.

5.1 The highest-priority VM

VMs are sorted according to decreasing priority, so the highest-priority VM is V_1 and the minimum $d_{l,min}$ in the system is $d_{1,min}$. The worst-case response time of V_1 is equal to its time slice s_1^x. Thus, its execution length becomes $L_1^x = p_1^x$ and Eq. (4) reduces to $p_1^x \leq d_{1,min}$. If we now select $s_1^x = e_{1,min}$ according to Eq. (3), we can set p_1^x to be equal to $d_{1,min}$.

The selected value of s_1^x allows meeting $d_{1,min}$ on V_1. However, we need to configure s_1^x for all tasks on V_1 to meet their deadlines. For this purpose, we make use of Eq. (6). Clearly, in the case of V_1, the function $\alpha_1^x(t_i - k_{1,i} \cdot p_1^x)$ reduces to $t_i - k_{1,i} \cdot p_1^x$ and Eq. (6) changes to:

$$k_{1,i} \cdot s_1^x + \min(s_1^x, t_i - k_{1,i} \cdot p_1^x) \geq \hat{w}_i. \tag{9}$$

To find a proper value of s_1^x that satisfies a deadline $d_i \geq d_{1,min}$, we assume that the second term on the left-hand side of this inequality is zero. Replacing $k_{1,i}$ by $\lfloor \frac{t_i}{p_1^x} \rfloor$ where $t_i = d_i - (p_1^x - s_1^x)$, we have:

$$\left\lfloor \frac{d_i - (p_1^x - s_1^x)}{p_1^x} \right\rfloor \cdot s_1^x \geq \hat{w}_i.$$

Now, removing the floor function, reshaping and equalizing to zero, we obtain a quadratic equation on s_1^x:

$$(s_1^x)^2 + (d_i - p_1^x) \cdot s_1^x - \hat{w}_i \cdot p_1^x = 0. \tag{10}$$

For T_i to be schedulable on V_1, at least one root of this equation must be a real positive number. This root gives us an approximated value of s_1^x for which T_i can be scheduled. We can then verify whether this value of s_1^x fulfills Eq. (9). If $k_{1,i} \cdot s_1^x + \min(s_1^x, t_i - k_{1,i} \cdot p_1^x)$ is exactly equal to $\hat{w}_i$, the obtained s_1^x is the minimum possible guaranteeing the schedulability of T_i. If it is greater than $\hat{w}_i$, then the obtained s_1^x guarantees the schedulability of T_i but a smaller s_1^x is also possible. In case that the obtained s_1^x does not fulfill Eq. (9) (which is possible since we have ignored the term $\min(s_1^x, t_i - k_{1,i} \cdot p_1^x)$ and removed the floor function to find s_1^x), we will need to increase s_1^x. T_i can be scheduled only if we can find a value of s_1^x that is less than or equal to p_1^x. In the latter two cases, we proceed as follows. We first compute Δs_1^x as given below:

$$\Delta s_1^x = \hat{w}_i - k_{1,i} \cdot s_1^x - \min(s_1^x, t_i - k_{1,i} \cdot p_1^x).$$

If Δs_1^x is positive, we have to increase the value of s_1^x. If Δs_1^x is negative, we can decrease s_1^x. Then, we compute:

$$\eta_1^x = k_{1,i} + \left\lceil \frac{\min(s_1^x, t_i - k_{1,i} \cdot p_1^x)}{s_1^x} \right\rceil.$$

This η_1^x is the number of times that V_1 executes before d_i either entirely or partially. The idea is to distribute Δs_1^x between all these V_1 instances/executions. Hence, we recompute s_1^x as the sum of the current value of s_1^x plus $\frac{\Delta s_1^x}{\eta_1^x}$. Clearly, s_1^x is going to increase if Δs_1^x is positive and decrease for a negative Δs_1^x. Since recalculating s_1^x might change the number of times V_1 executes before d_i, we need to verify again that the new s_1^x satisfies Eq. (9).

Complexity of the computation: In practice, VM monitors do not allow configuring any arbitrary value for time slices and periods of the VMs. These are normally discretized. That is, the value of time slices and periods are set to multiples of a system-dependent constant κ, i.e., $s_1^x = q_s \cdot \kappa$ and $p_1^x = q_p \cdot \kappa$, where q_s and q_p are any possible integer numbers greater than zero. The next possible greater value of s_1^x is thus $(q_s + 1) \cdot \kappa$ and so on. As a result, since s_1^x can be at most equal to p_1^x, the number of iterations required for finding the optimum value of s_1^x is limited. In other words, the described algorithm has pseudo-polynomial complexity.

Proceeding as discussed for every T_i on V_1, we can find the minimum possible s_1^x such that all tasks running on V_1 can meet their deadlines. Next, we analyze how to configure time slices and periods for the lower-priority VMs as well.

5.2 VMs with lower priority

A given lower-priority V_l schedules a set of tasks. As already discussed, among the tasks running on V_l, the one

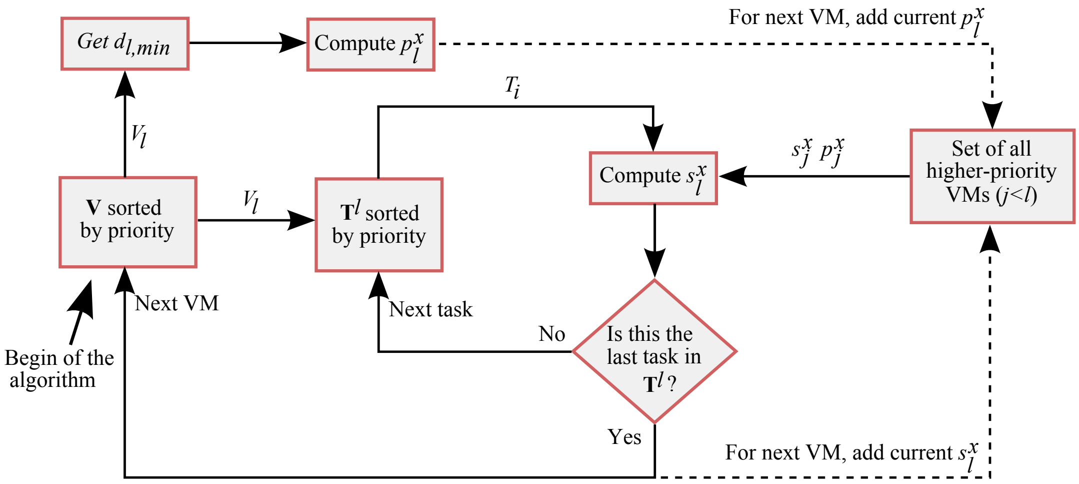
Get $d_{l,min}$
Compute p_l^x
For next VM, add current p_l^x
T_i
V_l
Compute s_l^x
s_j^x p_j^x
Set of all higher-priority VMs (j<l)
V sorted by priority
V_l
$\mathbf{T}^l$ sorted by priority
Next VM
Next task
No
Is this the last task in $\mathbf{T}^l$?
Begin of the algorithm
Yes
For next VM, add current s_l^x

Figure 4: Design procedure for VM schedulers

with the minimum deadline $d_{l,min}$ determines V_l's priority. In this case, the worst-case response time w_l^x is longer than s_l^x and its execution length is given by Eq. (2). The problem here is that w_l^x also depends on s_l^x and we cannot directly use Eq. (4) to find p_l^x. We know, however, that p_l^x must guarantee the schedulability of $d_{l,min}$. Since the task with $d_{l,min}$ has the highest priority on V_l, it is going to execute without delay as soon as V_l gains CPU access. So, if we set $s_l^x = e_{l,min}$, we can at least guarantee the schedulability of $d_{l,min}$. We can use this value of s_l^x to compute w_l^x applying Eq. (1). With s_l^x and w_l^x, we finally obtain $p_l^x = d_{l,min} + s_l^x - w_l^x$.

These values of s_l^x and p_l^x only guarantee that the task with $d_{l,min}$ can be scheduled on V_l. So, an s_l^x needs to be found such that all tasks on V_l can meet their deadlines. A new value of s_l^x changes also w_l^x. However, we do not need to recompute p_l^x since V_l can execute (with the value of p_l^x obtained above) $e_{l,min}$ time units before $d_{l,min}$ in worst case – recall that the task with $d_{l,min}$ has the highest priority on V_l and executes always first.

In order to find an s_l^x that also allows scheduling another task with $d_i \geq d_{l,min}$, we make use of Eq. (6). First, we assume that $\min\left(s_l^x, \alpha_l^x(t_i - k_{l,i} \cdot p_l^x)\right) = 0$ on the left-hand side of this inequality. Replacing $k_{l,i}$ by $\lfloor \frac{t_i}{p_l^x} \rfloor$ where $t_i = d_i - (p_l^x - s_l^x)$, we have just as before:

$$\left\lfloor \frac{d_i - (p_l^x - s_l^x)}{p_l^x} \right\rfloor \cdot s_l^x \geq \hat{w}_i.$$

Now, removing the floor function, reshaping and equalizing to zero, we obtain a quadratic equation on s_l^x:

$$(s_l^x)^2 + (d_i - p_l^x) \cdot s_l^x - \hat{w}_i \cdot p_l^x = 0.$$

As for the previous case of V_1, a task T_i is schedulable on V_l if at least one root of this equation is a real positive number. Again, this root give us an approximated value of s_l^x for T_i. We need to check whether this value of s_l^x fulfills Eq. (6). If $k_{l,i} \cdot s_l^x + \min\left(s_l^x, \alpha_l^x(t_i - k_{l,i} \cdot p_l^x)\right)$ is greater than $\hat{w}_i$, then the obtained s_l^x can be reduced without compromising T_i's schedulability. Otherwise, if the obtained s_l^x does not fulfill Eq. (6) (recall that $\min\left(s_l^x, \alpha_l^x(t_i - k_{l,i} \cdot p_l^x)\right)$ was assumed to be zero and we removed the floor function), we will need to increase s_l^x. In this latter case, T_i can only be schedulable, if there exists an s_l^x that is less than or equal to the period p_l^x assigned to the VM.

To find the minimum possible s_l^x for T_i, we will have to compute Eq. (6) in an iterative manner because $\alpha_l^x(t)$ is a non-linear function. Since a valid s_l^x is at most equal to p_l^x, we will be able to find a solution in a limited number of iterations. However, a linear approximation of $\alpha_l^x(t)$ allows us to reach a safe value of s_l^x in less steps. Recall that $\alpha_l^x(t)$ returns the largest e_l^x for which Eq. (7) and (8) hold. Now, Eq. (7) holds true if we replace $t^{(c+1)}$ and $t^{(c)}$ by the convergence value t_l^x. Removing the ceiling function and reshaping, we can achieve an upper bound on t_l^x:

$$t_l^x \leq \frac{e_l^x + S_{(l-1)}^x}{1 - U_{(l-1)}^x}, \tag{11}$$

where $S_{(l-1)}^x = \sum_{j=1}^{l-1} s_j^x$, and $U_{(l-1)}^x = \sum_{j=1}^{l-1} \frac{s_j^x}{p_j^x}$. Further, to obtain a linear approximation of $\alpha_l^x(t)$, we simply replace t_l^x in Eq. (8) and solve for e_l^x:

$$e_l^x \leq r_{l,i} \cdot \left(1 - U_{(l-1)}^x\right) - S_{(l-1)}^x, \tag{12}$$

where $r_{l,i} = t_i - k_{l,i} \cdot p_l^x$. Using this approximation, we can rewrite Eq. (6) in the following manner:

$$k_{l,i} \cdot s_l^x + \min\left(s_l^x, r_{l,i} \cdot \left(1 - U_{(l-1)}^x\right) - S_{(l-1)}^x\right) \geq \hat{w}_i. \tag{13}$$

We can proceed as previously calculating Δs_l^x:

$$\Delta s_l^x = \hat{w}_i - k_{l,i} \cdot s_l^x - \min\left(s_l^x, r_{l,i} \cdot \left(1 - U_{(l-1)}^x\right) - S_{(l-1)}^x\right).$$

Further, we can obtain the number of times that V_l is activated before d_i in the following way:

$$\eta_l^x = k_{l,i} + \left\lceil \frac{\min\left(s_l^x, r_{l,i} \cdot \left(1 - U_{(l-1)}^x\right) - S_{(l-1)}^x\right)}{s_l^x} \right\rceil.$$

Similar to the case of V_1, the idea is to distribute Δs_l^x between these η_l^x executions of V_l. So, the approximated minimum possible s_l^x will be given by the current s_l^x plus

$\frac{\Delta s_l^x}{\eta_l^x}$. The value of s_l^x increases if Δs_l^x is positive or decreases if Δs_l^x is negative. We again need to test whether the new s_l^x complies with Eq. (13), because V_l may execute a different number of times with the new s_l^x. As in the case of V_1, a second iteration may be necessary to achieve the best possible s_l^x according to Eq. (13).

As stated before, we can also use the exact Eq. (6) instead of (13) to find an optimal s_l^x – the procedure described above does not change. However, this requires a greater number of iterations since $\alpha_l^x(t)$ in (6) is a non-linear function and needs to be solved iteratively.

Complexity of the computation: Similar to the case of the highest priority VM, the complexity of the above described algorithm is pseudo-polynomial. Even if the exact expression of Eq. (6) is used for computing s_l^x, the complexity remains pseudo-polynomial. Eq. (6) requires indeed more iterations, however, these are also pseudo-polynomially bounded as explained in Section 4.1.

Proceeding as discussed for every T_i on V_l, we can find the approximated (or optimum if we use (6)) minimum possible s_l^x guaranteeing the schedulability of all real-time tasks running on V_l.

6. CASE STUDY

In this section, we consider two automotive control applications: Electronic Stability Control (ESC) and Engine Management (EM). In principle, ESC improves the steering capability of a vehicle by minimizing blocking and skidding on the wheels. On the other hand, EM calculates the optimum ignition point after every revolution of the car's engine. Both these applications have been simplified in this paper and adapted to illustrate our design technique for the VM scheduler.

Now, the ESC system considered here consists of two real-time tasks: T_1 and T_2. These tasks perform calculations and take decisions based on data collected from the wheel sensors. The EM application is composed of three real-time tasks: T_3, T_4 and T_5. The following table shows the different task parameters.

Table 1: Real-time tasks

T_i	p_i	d_i	e_i
T_1	$5ms$	$2.5ms$	$1ms$
T_2	$5ms$	$5ms$	$2ms$
T_3	$20ms$	$7ms$	$1ms$
T_4	$20ms$	$10ms$	$3ms$
T_5	$40ms$	$40ms$	$4ms$

The inter-arrival times of the EM tasks (T_3 to T_5) depend on the rotational speed of the engine, since sensors are located on the crankshaft. Clearly, the minimum inter-arrival times result from the highest possible rotational speed.

To schedule these two applications in our setup, we use the Xen hypervisor (version 3.4) featuring the fixed-priority VM scheduler presented in [16]. The system runs on an Intel Core 2 Duo platform operating at 2.16 GHz. Although Xen was originally developed for x86, it is currently being ported to architectures such as ARM and PowerPC that are typically encountered in automotive electronics. In addition, there exist lately a great interest in using the general purpose processing infrastructure in a modern car (typically used in navigation and entertainment) to schedule real-time applications. In this context, Xen seems to be is an interesting solution for providing isolation between VMs/domains running real-time and those running less critical tasks.

All VMs in Xen can only access hardware devices through a so-called privileged domain or domain zero (dom0). The OS running in dom0 must provide the device drivers for accessing the hardware [4]. However, it is possible to assign single hardware devices to an *unprivileged* domain (domU), which then has to provide the necessary device drivers.

In this case study, we assign the NIC (Network Interface Controller) to an unprivileged domain that we call network domain (domN). This domN is the interface between all VMs in the system and the communication network. In addition, we denominate by domRT a VM running one or more real-time tasks.

Now, we assume that sensors are connected to a bus and this again is connected to an Ethernet network via a gateway. Our Xen system then receives packets arriving through the NIC over this network. The number of multimedia or entertainment applications is increasing rapidly in today's cars. Hence, it is not unusual to encounter technologies in the automotive domain that are otherwise typical from the desktop domain.

The deadlines in Table 1 express the maximum acceptable reaction time to incoming packets (measured from the time instant at which a packet arrives to the time instant at which the response packet leaves the system). We analyze two possible design cases:

A) Each real-time task runs on a separated domain.

B) All application tasks run together on the same domain.

The case A is more inefficient from the perspective of RAM memory usage, because a domain requires around 20MBytes (when instantiating the Debian Lenny OS with real-time patches), while a task only needs approximately 0.5MBytes. (If mini-os is used – this a small operating system provided by Xen – only 4MBytes will be required for a domain.) On the other hand, using one task per domain allows for a higher utilization. This is because we can choose the time slice and the period of a each VM to fit the specific requirements of the only task running on it.

In what follows we compare these two design cases. Nevertheless, the first step towards whichever design case is to configure domN in a proper manner, which is the entry point of external events to the system.

6.1 The network domain

For the reason that domN is the interface between any domRT and the network, it is as critical as the most critical domain. All real-time tasks from the highest- to the lowest-priority one are released by packets arriving via domN from the network. As a consequence, to allow for preemptive scheduling, domN needs to be assigned the highest priority among all domRTs running on the same core.

The time slice and the period of domN will be denoted by s_N^x and p_N^x respectively. To find a suitable value for s_N^x, we need to consider that neither the NIC nor its drivers in domN can prioritize packets. However, we can enforce that only a maximum number of n packets need to be processed at any point in time. This way, it is possible to bound the blocking time due to lower-priority packets. This can be achieved by allocating at most n tasks to the system that receive/send packets over the network. Recall that the case $d_i \leq p_i$ is considered, so if the system does not miss any deadline, there will be always at most n packets to process. (The system reacts to a packet coming from a sensor before the next packet from the same sensor arrives.) In this case, s_N^x can be chosen as follows: $s_N^x = n \cdot e_N$, where e_N stands for the worst-case processing time of a packet in whichever direction (i.e., incoming/outgoing). Considering the two applications mentioned above, we have five tasks that have access to the network, i.e., $n = 5$. If now $e_N = 0.06ms$, s_N^x must be at least equal to $0.3ms$ to be able to process five packets per period.

Now, if $d_{1,min}$ is the minimum deadline among all d_i in the whole system, p_N^x has to be configured such that this most critical deadline can be met. The worst-case response time to $d_{1,min}$ is illustrated in Fig. 5. Here, V_1 (i.e., the domain/VM reacting to $d_{1,min}$) is released after domN finishes executing. In addition, the worst-case p_N^x results from considering that V_1 finishes before the next activation of domN – see Fig. 5. Recall that s_N^x was selected such that a packet of every task (either incoming or outgoing) can be processed within p_N^x. V_1's outgoing packet has to wait up to the next period of domN to be sent. This is because domN might have already used its whole slice in the current period.

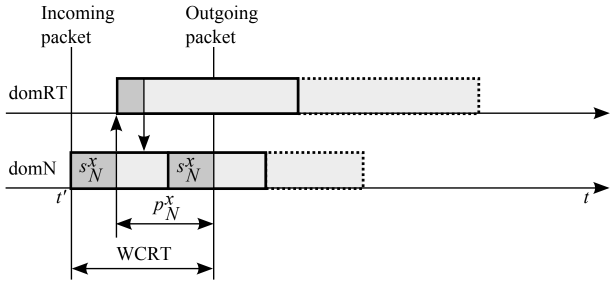

Figure 5: Worst-case response time (WCRT) to $d_{1,min}$

From Fig. 5, the following inequality must hold for the system to meet $d_{1,min}$: $s_N^x + p_N^x \leq d_{1,min}$, and we can now obtain $p_N^x = d_{1,min} - s_N^x$. Thus, as $d_{1,min} = 2.5$, $p_N^x = 2.2ms$ allows reacting to $d_{1,min}$ in time.

Design case A: From the point of view of the design, it is easy to tune the VM scheduler in this case. The fact that we have one real-time task per domain leads to five different domains (a domain for each of the tasks in Table 1).

Clearly, we assign priorities according to the DM policy, since this results in the optimal priority assignment [14]. So, the domain running T_1 has the highest priority, whereas the domain running T_5 the lowest. The periods and time slices can be simply set according to the task parameter. That is, the domain running T_1 is assigned $p_1^x = p_1 = 5ms$ and $s_1^x = e_1 = 1ms$, while $p_2^x = p_2 = 5ms$ and $s_2^x = e_2 = 2ms$ are the parameters of the domain running T_2, and so on. The whole system is then feasible, if the worst-case response time of each domain/VM is less than or equal to the deadline of the task running on it.

Design case B: The second design case is the focus of this paper. Here we schedule each single application on one domain/VM. For the considered case, that is, the ESC tasks T_1 and T_2 are scheduled together on one domain, while T_3 to T_5 are scheduled on a separated domain. Recall from Section 4.1 that higher-priority VMs/domains have influence on the design of lower-priority ones. Since domN has to have the highest priority on the core it runs, it is going to affect the configuration of the other domains running on the same core. Hence, we need first to determine on which core domN is going to run. It seems reasonable to allocate domN together with the EM domain to the same core and to let the ESC domain run on the other core. This is because the EM tasks (T_3 to T_5) have a total utilization of 0.3, while the ESC tasks show a utilization of 0.6.

Let us illustrate the design of the ESC domain. Proceeding as in Section 5.1, we select $p_{ESC}^x = d_1 = 2.5ms$ since d_1 is the minimum deadline. We also know that $s_{ESC}^x \geq e_1 = 1ms$ is needed in order that this domain can schedule T_1.

Now, for T_2 to be schedulable, we need to recompute s_{ESC}^x. First, we compute the T_2's worst-case execution demand within d_2 (i.e., $\hat{w}_2$) using Eq. (5) – only T_1 is considered since T_1 and T_2 run alone on the same VM. As it can be seen, $\hat{w}_2 = 3ms$ is less than $d_2 = 5ms$, which is a necessary condition for the design procedure to be valid. Further, we can apply Eq. (10) where the time slice s_1^x is given here by s_{ESC}^x and the period p_1^x is p_{ESC}^x, respectively:

$$(s_{ESC}^x)^2 + (5 - 2.5) \cdot s_{ESC}^x - 3 \cdot 2.5 = 0.$$

The two roots of this equation are 1.76 and −4.26. Clearly, the negative root can be discarded. So, the approximated value of s_{ESC}^x is 1.76. With this s_{ESC}^x, we can compute $t_2 = 5 - (2.5 - 1.76) = 4.26$ and $k_{ESC,2} = \lfloor \frac{t_2}{2.5} \rfloor = 1$. Then, we verify whether $s_{ESC}^x = 1.76$ satisfies Eq. (9) where $k_{1,i} = k_{ESC,2}$:

$$1 \cdot 1.76 + \min(1.76, 4.26 - 1 \cdot 2.5) \geq 3,$$

which happens to hold. Consequently, we compute next the value of Δs_{ESC}^x:

$$\Delta s_{ESC}^x = 3 - 1 \cdot 1.76 - \min(1.76, 4.26 - 1 \cdot 2.5) = -0.52.$$

Then, we calculate the corresponding η_{ESC}^x:

$$\eta_{ESC}^x = 1 + \left\lceil \frac{\min(1.76, 4.26 - 1 \cdot 2.5)}{1.76} \right\rceil = 2.$$

This η_{ESC}^x is the number of times that the ESC domain executes before d_2. Then, we can recompute s_{ESC}^x:

$$s_{ESC}^x = 1.76 + \frac{(-0.52)}{2} = 1.5.$$

Replacing again $s_{ESC}^x = 1.5$ in Eq. (9), we can see that this is the minimum possible value of s_{ESC}^x. So, the ESC domain can meet all deadlines if $s_{ESC}^x = 1.5ms$ and $p_{ESC}^x = 2.5ms$.

Since the EM domain runs together with the higher-priority domN on the same core, we need to use the method described in Section 5.2 to find the necessary time slice and period. Due to lack of space, we do not show this whole procedure here, but this is similar to the one shown above. The resulting parameters for the EM domain are $s_{EM}^x = 3.85ms$ and $p_{EM}^x = 6.7ms$.

6.2 Experimental Comparison

In this section, we show the results of a set of experiments that we have conducted upon this setup. Fig. 6 to 10 show the response times of T_1 to T_5 with respect to an increasing higher-priority CPU load. This is the processor utilization produced by higher-priority domains running on both cores.

A remote computer was connected to our setup via Ethernet and simulates the sensors generating packets for the different real-time tasks in the system. For every measurement (i.e., for every marker on the curves) and each of the figures, 20,000 different packets were sent over the network to each task in the system.

On average, in Fig. 6, the response time of T_1 is below its deadline of $2.5ms$ until around 70% of higher-priority load. There is not much difference between the two compared design cases. The variability (jitter) in the response time is represented by vertical bars. Also from the point of view of jitter, there is no significant difference between case A and B in what respects to T_1.

In Fig. 7, the average response time of T_2 is less than $d_2 = 5ms$ up to approximately 50% of higher-priority load. Here, the average response time is better (less) for case A and a higher-priority load in the range of zero to 30%. This is an expected behavior, since T_2 shares the time slice with T_1 in case B. However, T_2 can still meet its deadline up to 50% of higher-priority load. There is no meaningful difference between the two compared design cases with respect to T_2's jitter – also represented by vertical bars in Fig. 7.

For T_3 in Fig. 8, the response time is always less than $d_3 = 7ms$ irrespective of the higher-priority load. There is no significative difference between case A and B, neither with respect to average response time nor to jitter. This behavior is probably because T_3 has a comparatively large deadline and small worst-case execution time.

In Fig. 9, the response time of T_4 is always less than its deadline $10ms$ at least until a higher-priority load of 65%. Again, we could not observe any mentionable difference between case A and B for neither the average response time nor the jitter.

Finally, Fig. 10 illustrates the response time of T_5 for both design cases considered. As it can be seen, there is no deadline miss until around 80% higher-priority load. Here again, this relatively good behavior is most likely a consequence of T_5's large deadline and small execution time. For the task T_5, we can observe again that case A leads to a better average response time when the higher-priority load is below 35% of the total.

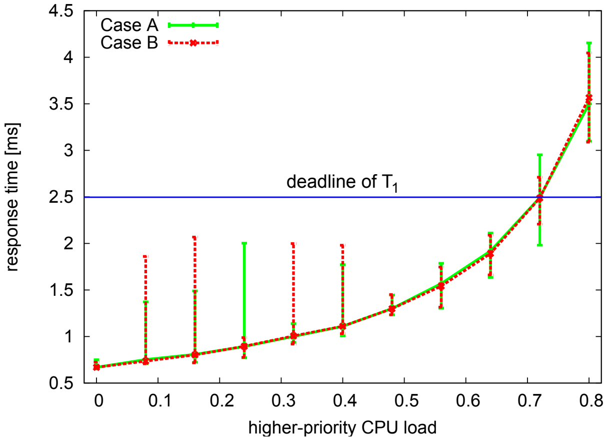

Figure 6: Response time of T_1

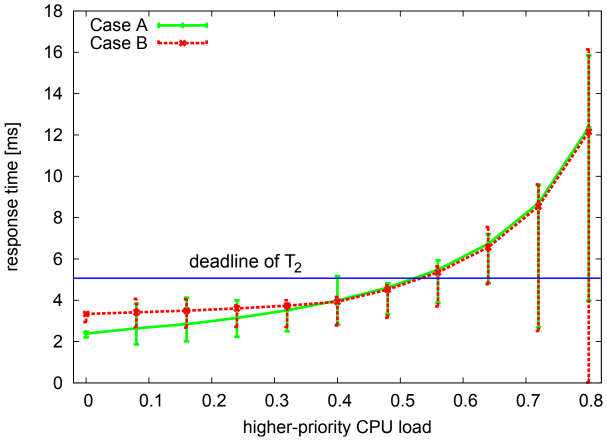

Figure 7: Response time of T_2

7. CONCLUDING REMARKS

In this paper, we proposed a method to design/synthesize a fixed-priority VM scheduler in the context of embedded real-time applications. We considered the case where multiple real-time tasks run on multiple VMs. Since time slices and periods need to be selected for every VM in the system, the design of a VM scheduler differs from the known techniques for analyzing hierarchical scheduling.

As expected, the period of a VM is determined by the minimum deadline that has to be scheduled on that VM. On the other hand, the selection of an efficient time slice requires an iterative procedure. By considering the minimum requirements for a VM and the schedulability condition of a task running on that VM, we first compute an estimate of the VM's time slice. This is used as an initial value, which can then be improved towards the optimum in a reduced number of subsequent steps.

In addition, to illustrate the proposed design technique, we presented a case study consisting of two automotive applications running upon the Xen hypervisor. The Xen version considered here was previously extended by a fixed-priority real-time VM scheduler.

Based on our setup, we compared the case where each real-time task runs on a dedicated VM to the case where a whole application (i.e., multiple tasks) runs on a single VM. From this comparison, we observed that the average response time improves when only one task is scheduled on a VM. This is expectable, since it is possible to configure the VM to the specific requirements of one task if the latter runs exclusively on that VM. On the other hand, running several tasks on the same VM allows for a better use of RAM memory, which is a scarce resource in embedded systems.

As future work, we plan to extend the presented approach to consider more general scheduling algorithms and, in particular, we are interested in analyzing the effect of VM/task migration in designing a VM scheduler under real-time constraints.

Acknowledgements: We would like to thank the reviews for their comments and contributions to this paper.

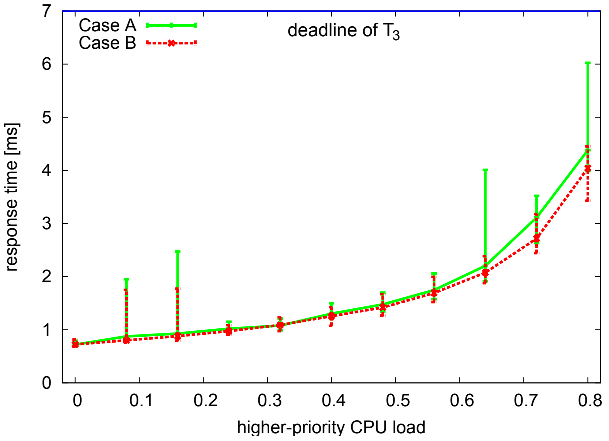

Figure 8: Response time of T_3

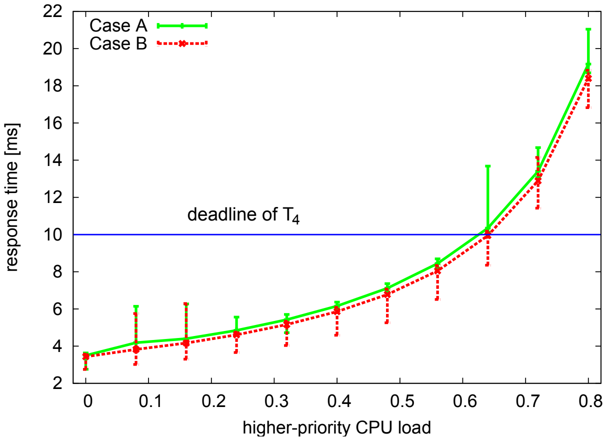

Figure 9: Response time of T_4

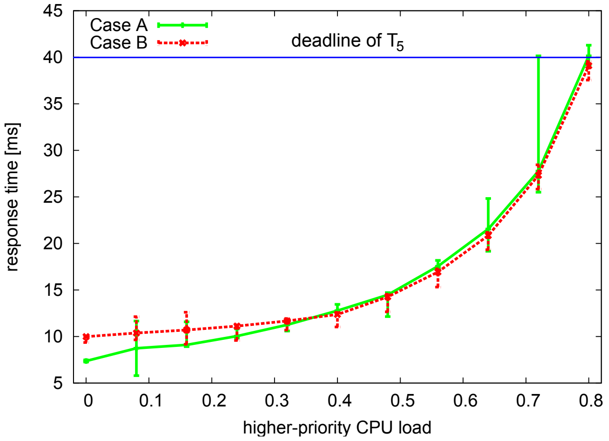

Figure 10: Response time of T_5

8. REFERENCES

[1] www.xen.org.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 1993.

[3] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[4] D. Chisnall. *The Definitive Guide to the Xen Hypervisor.* Prentice Hall, 2007.

[5] R. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 389–398, December 2005.

[6] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 308–319, December 1997.

[7] M. Di Natale. Design and development of component-based embedded systems for automotive applications. In *Proceedings of the 13th Ada-Europe International Conference on Reliable Software Technologies*, pages 15–29, June 2008.

[8] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 129–138, December 2007.

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

[10] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 126–136, 2007.

[11] H. Kim, H. Lim, J. Jeong, H. Jo, and J. Lee. Task-aware virtual machine scheduling for i/o performance. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110, 2009.

[12] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 256–267, December 1999.

[13] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[14] J.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. In *Performance Evaluation*, volume 2, pages 237–250, 1982.

[15] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 217–226, December 2000.

[16] A. Masrur, S. Drössler, T. Pfeuffer, and S. Chakraborty. VM-based real-time services for automotive control applications. In *Proceedings of the 16th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 218–223, August 2010.

[17] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, pages 75–84, April 2001.

[18] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10, 2008.

[19] A. Sangiovanni-Vicentelli and M. Di Natale. Challenges and solutions in the development of automotive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):937–940, July 2009.

[20] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 181–190, July 2008.

[21] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 2–13, December 2003.

[22] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Transactions on Embedded Computing Systems*, 7(3):1–39, 2008.

[23] C. Weng, Z. Wang, M. Li, and X. Lu. The hybrid scheduling framework for virtual machine systems. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120, 2009.