VM-Based Real-Time Services for Automotive Control Applications

Alejandro Masrur*, Sebastian Drössler†, Thomas Pfeuffer* and Samarjit Chakraborty*
*Institute for Real-Time Computer Systems, TU Munich, Germany
{Alejandro.Masrur, Thomas.Pfeuffer, Samarjit.Chakraborty}@rcs.ei.tum.de
†ReliaTec GmbH, Garching, Germany
s.droessler@reliatec.de

***Abstract*—Techniques for hardware virtualization have been successfully used to provide hardware-independent services and increase isolation between applications in the desktop domain. However, these characteristics make hardware virtualization also interesting for other domains like those involving control tasks. Since these techniques were initially not conceived for this kind of environments where, in particular, timing constraints must be guaranteed, it is necessary to analyze their behavior and investigate the viability of possible solutions based on them. In this paper, we are concerned with using VMs (Virtual Machines) to provide real-time services in the context of automotive control applications. For this purpose, we make use of the Xen hypervisor to design a real-time control loop on the top of a virtualization layer. We first analyze a typical Xen configuration and identify problems that arise when it is used for real-time applications. We show that the worst-case performance of Xen's standard SEDF scheduler (Simple Earliest Deadline First) can be improved by incorporating some minimal modifications. In addition, in order to reduce latency and jitter in a real-time control loop, we propose a new scheduler for the Xen hypervisor that uses the concept of a *real-time VM*. Real-time VMs are then scheduled before any other VM and under a fixed-priority policy. The proposed VM-based solution is shown to guarantee timing constraints typically encountered in automotive control applications. We further illustrate this through an extensive set of experiments.**

I. Introduction

In order to prevent errors from propagating, some form of isolation between the different tasks or applications if often required in embedded systems. Of course, standard features like user/supervisor modes and the MMU (Memory Management Unit) allow us to isolate the operating system (OS) from user applications. However, isolation between multiple user-level applications—especially those with varying levels of criticality—is often required.

In such cases, introducing a virtualization layer between hardware and software turns out to be an effective solution. Such a virtualization layer abstracts the hardware and is referred to as a *virtual machine monitor* or a *hypervisor*. The OS and application software now run on one or more so-called *virtual machines* (VMs) and not directly on the hardware. The VM monitor traps all requests directed to shared resources (like processors and I/O devices) and administers access to them by scheduling the VMs. This way, a malfunctioning task can at most affect other tasks on the same VM, but not tasks on other VMs running in the system.

However, most of the virtualization techniques available today are directed towards the desktop domain (here we are referring to system virtualization rather than program virtualization—as in the Java Virtual Machine—techniques). As a result, most VM monitors focus on optimizing the average performance of systems and do not deal with issues typically encountered in embedded systems such as timing constraints, output jitter, etc. Hence, analyzing and appropriately configuring existing VMs in this latter case has lately attracted a lot of attention.

Our contributions: In this paper we use the Xen hypervisor, an open-source VM monitor, for providing real-time services in the context of automotive control applications. Traditionally, different functionalities or applications in the automotive domain are implemented on different electronic control units (ECUs). This has led to a large number of ECUs in high-end cars, which increases cost and leads to wiring complications. As a result, lately there is an increasing focus on integrating multiple applications on a single ECU, along with a VM layer to provide isolation between them. The work we report in this paper follows this direction, with the aim of supporting a mix of hard-real-time control applications with non- or soft-real-time applications on a commodity VM, *viz.*, Xen.

In order to design a real-time control loop upon Xen, we first analyze its timing behavior. Currently, Xen provides two schedulers—the credit-based scheduler, which was designed to fairly share hardware resources, and the SEDF (Simple Earliest Deadline First) scheduler. In this paper we focus on the SEDF scheduler since it is based on the well-known EDF and suits better a hard-real-time application. However, SEDF differs from EDF, so we discuss these differences later.

SEDF can be configured so as to satisfy real-time constraints, however, a significant amount of pessimism needs to be introduced resulting in a poor resource utilization. We show that it is possible to improve the worst-case behavior of SEDF by appropriately modifying it. Nevertheless, both SEDF and the modified SEDF presented in this paper suffer from poor performance when a combination of time-critical and non-critical VMs are scheduled together. To overcome this problem, we propose a new scheduler for Xen that further reduces latency and jitter in a control loop. The proposed scheduler is based on the notion of a *real-time VM*. Such a VM is scheduled separately with a fixed-priority scheme, which leads to a better worst-case behavior. Since the Xen hypervisor schedules multiple VMs and isolates them from one another, we can run safety-/time-critical applications (e.g., those related to airbag control and brake system) on the same hardware with general-purpose applications (e.g., navigation

and multimedia). This allows more design flexibility and supports merging multiple functionalities onto fewer processors to reduce the cost and the weight of a vehicle.

The rest of the paper is organized as follows. First, we give an overview of a regular Xen configuration and discuss the SEDF scheduler in Section III. In Section IV, we analyze the use of Xen in the context of real-time applications, for which the standard SEDF scheduler needs to be configured carefully. We further propose in Section VI a new scheduler for Xen that improves the average system utilization and helps reducing latency and jitter in a control loop. Finally, in Section VII, we discuss a set of experiments on the basis of our setup and summarize our contributions in Section VIII.

II. Related work

Most related work in this area focus on analyzing the performance and fairness of VM scheduling policies. For example, in [1], Gupta et al. discuss performance issues concerning device accesses in Xen. They further propose techniques to allocate CPU time taking resource usage into account. Govindan et al. present mechanisms to take communication into consideration when scheduling CPU time [2]. In [3], Cherkasova et al. present a comparison between different schedulers in Xen, whereas Ongaro et al. study the impact of jointly scheduling CPU-demanding, bandwidth-intensive and latency-sensitive applications [4]. In [5], Kim et al. introduce scheduling techniques to allocate VMs according to their I/O demand, with the aim of performance improvement. Further, in [6], Weng et al. analyze problems that arise when concurrent tasks are scheduled based on VMs. They further propose the distinction between two types of VM: the concurrent and the high-throughput VMs. In this paper, we also propose distinguishing between VMs of two types as discussed later: real-time and non-real-time VMs.

More recently, Mangharam and Pajic introduced the concept of an embedded VM for wireless networks [7]. Here, besides providing hardware abstraction, a embedded VM defines mechanisms for fault tolerance against node and communication failures in wireless networks. In this paper, we study the use of VMs and, in particular, of Xen for supporting hard-real-time applications—a general description of the Xen hypervisor may be found in [8]. We mainly focus on automotive control applications, but the techniques presented below can also be applied to other domains. To the best of our knowledge, there is no published work on using VMs for scheduling hard-real-time tasks or safety-critical automotive applications.

III. A typical Xen configuration

In this section, we briefly describe a typical configuration of Xen. Only one domain, denoted as the host domain or dom0, has enhanced rights to manage the others domains called unprivileged domains (domUs)—a VM or virtualized environment is referred to as *domain* in the Xen terminology. From dom0, it is possible to create, pause or stop the domUs running on Xen. The Xen hypervisor itself does not provide any drivers for accessing hardware devices. Instead, the OS running within dom0 has to provide the device drivers, for example, for network interface cards (NICs) or other I/O devices. The domUs cannot access the hardware directly but over dom0, however, a hardware device can be hidden from dom0 and assigned to a domU. In this case, the domU has to provide the corresponding device drivers.

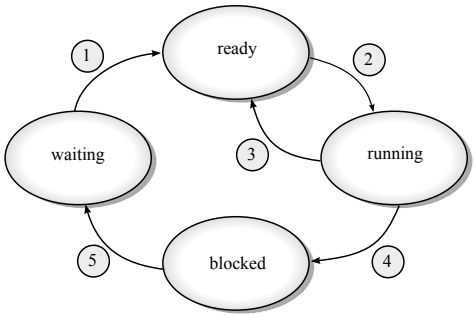

Figure 1. Domain states and state transitions in SEDF

In Xen, the domain scheduler is responsible for assigning CPU time to domains such that every domain gets to run. For this purpose, Xen uses virtual CPUs (VCPUs) as an abstraction layer of the physical processor. For every available physical CPU/core, there can be several associated VCPUs. Further, several VCPUs can be assigned to the same domain. However, once a VCPU is allocated to a given domain, it cannot be allocated to any other domain. Currently Xen includes two different schedulers: the credit-based and the SEDF scheduler. The credit-based scheduler was primarily developed for achieving a fair sharing of resources. On the other hand, SEDF is more suitable for real-time applications because of being based on EDF as discussed next.

A. The domain scheduler: SEDF

SEDF offers a series of configuration parameters that allow tuning it for the application at hand. First, domains need to be assigned to physical CPUs at configuration time. In addition, the following parameters can be adjusted independently for each domain:

- Slice s_i^x: This is the maximum amount of CPU time that may be assigned to a domain upon activation.
- Period p_i^x: This is the expected minimum separation between two consecutive activations of a domain.

SEDF then schedules domains in a preemptive manner using dynamic priorities. At a given release time t', the priority of a domain is given by the sum of the current time plus the domain's period $t' + p_i^x$: the less the value of $t' + p_i^x$, the higher the priority of the domain.

Figure 1 illustrates the states and the corresponding state transitions for domains under SEDF. There are four domain states and five possible transitions between them. In the ready state, a domain has been released, but it is not being executed yet. If a domain is ready, it can only change to the running state. From the running state, a domain can either go back to the ready queue, if a higher-priority domain preempts it, or it can be blocked. A domain passes to the blocked state whenever it finishes executing or it consumes its whole time

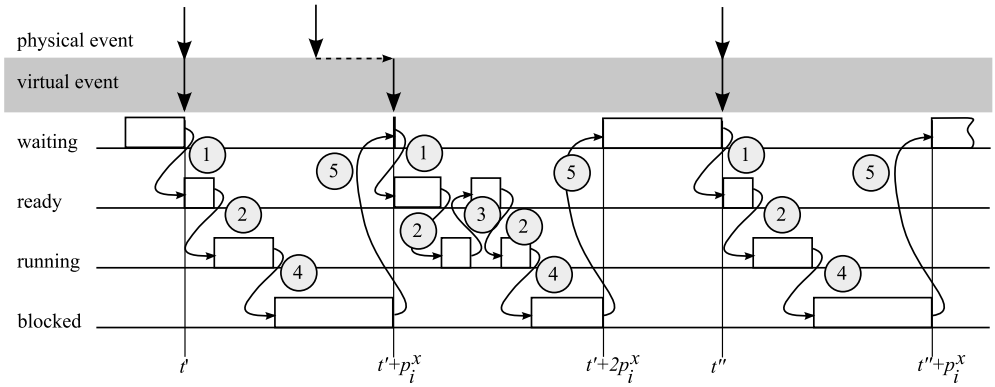

Figure 2. Transition scenarios for a domain with period p_i^x. Short unblocking occurs in $[t', t' + p_i^x]$: The virtual event is deferred until the next configured release time. The preemption case is illustrated in $[t' + p, t' + 2p_i^x]$ and long unblocking in $[t' + 2p_i^x, t'' + p_i^x]$.

slice s_i^x. Domains that are blocked can only change to the waiting state when their configured period elapses, i.e., when the current time t is greater than $t' + p_i^x$ where t' denotes the release time of the domain in blocked state. Finally, a waiting domain can leave this state and becomes ready when an incoming event produces a new release.

Figure 2 shows three different transition scenarios under SEDF. From time t' to $t' + p_i^x$, a domain is initially in the waiting state as an external event arrives to the system. This is translated into a virtual event by Xen releasing the domain execution with minimum delay. The domain starts executing and finishes before its time slice expires such that it becomes blocked. Now, if another external event occurs before the domain's configured period elapses (i.e., before $t' + p_i^x$), this is going to be delayed until $t' + p_i^x$, i.e., the time at which a new release of the corresponding domain is allowed by SEDF. We refer to this situation as *short unblocking*.

The second scenario, from time $t' + p_i^x$ to $t' + 2p_i^x$, illustrates the case where the domain gets preempted by another domain with higher priority. Finally, the third scenario shows the situation where a domain activation does not occur at $t' + 2p_i^x$ but at a later point in time t''. This latter situation in which the time between two activations is longer than the configured period p_i^x is referred to as *long unblocking*.

Clearly, in case of a long-unblocking activation, we have more relaxed conditions than the ones considered in the configuration of SEDF. On the contrary, short-unblocking leads to additional delay in reacting to an external event and should be analyzed in more detail for the purpose of real-time applications.

IV. Our setup for real-time applications

In our setup, Xen runs on an x86 processor and is connected via a switched Ethernet network to other processors and through gateways to sensors and actors. We further hide the NIC drivers from dom0 and assigned them to a separate domain called domN. This way, it is possible to improve the responsiveness of the system to packets arriving from the communication network.

Neither dom0 nor domN contain application tasks and, hence, they consume little CPU time (e.g., domN only reacts to network packets). In our setup, we use standard OSs that are already available for Xen and provide the necessary device drivers for accessing the hardware. However, these are not real-time OSs and we need to make provisions to ensure correct timing behavior of the system. In particular, the NIC drivers cannot differentiate between real-time and non-real-time packets. As a consequence, in order to be able to guarantee a maximum response time to real-time packets, we need to make sure that at most k packets need to be processed at any given point in time. This can be achieved by restricting the number of communicating domains in the system (real-time and non-real-time ones) to be at most k—considering that a domain answers to a packet before the next packet arrives.

We introduce the concept of a *real-time domain* (domRT) to denote an unprivileged domain running a real-time task. A domRT is also based on a standard OS available for Xen, which also does not have real-time capabilities. For example, it does not feature any real-time scheduler (such as EDF or fixed-priorities). However, if we have no more than one real-time task per domRT, we can still guarantee correct timing behavior. This is because the real-time task will be scheduled whenever the corresponding domRT is scheduled by Xen independently of the scheduler used by the OS in domRT.

A. Tuning SEDF for real-time applications

The slice s_i^x is the maximum running time of a domain within p_i^x time units, where p_i^x is the period of the domain. Hence, as SEDF implements a partitioned EDF scheduling, the system will be feasible if the sum of the domain utilizations $\frac{s_i^x}{p_i^x}$ does not exceed 1 on every available CPU/core [9]: $\sum_{i=1}^{l} \frac{s_i^x}{p_i^x} \leq 1$, where l is the number of domains running on a given CPU. Clearly, if this does not hold for any CPU in the system, we cannot guarantee real-time constrains [9]. On the other hand, if this inequality always holds, we can, however, neither conclude that all deadlines will be met by the system. In fact, s_i^x and p_i^x have to be configured taking the application requirements into account, otherwise, some deadlines may still be missed.

As mentioned above, because a domRT is based on a non-real-time OS available for Xen, we can only execute one real-

time task per domRT. Now, all domRTs in the system are triggered by network packets arriving from sensors. A sensor sends packets to a domRT periodically with period p_i. In order to guarantee the correct behavior of real-time applications, Xen has to react to an incoming packet within d_i time units. That is, d_i is the deadline or the maximum acceptable delay measured between incoming and outgoing packet. Within d_i, domN has to process the packet arriving from the sensor, the domRT has then to compute a new output value and, finally, domN has to send out the new output value.

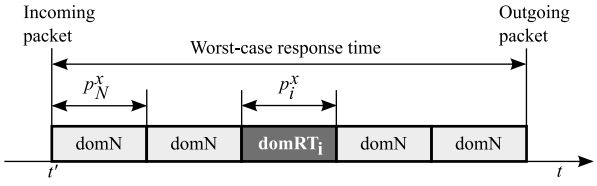

Figure 3. Worst-case response time under SEDF

Let us denote by s_N^x and p_N^x the slice and period configured in SEDF for domN where $s_N^x \leq p_N^x$ holds. In the same manner, s_i^x and p_i^x are the slice and period for the i-th domRT and $s_i^x \leq p_i^x$ also holds. Figure 3 illustrates the system's worst-case response time to an incoming packet directed to the i-th domRT. In this figure, the packet arrives at time t' immediately after domN gets into the blocked state and has to wait until $t' + p_N^x$—this is known as short unblocking, see Section III. However, if domN has the lowest priority in $[t'+p_N^x, t'+2p_N^x)$, the packet processing will be delayed until time $t' + 2p_N^x$. At time $t' + 2p_N^x$, the i-th domRT is activated by the scheduler. The execution of this domRT can also be delayed up to $t' + 2p_N^x + p_i^x$ by higher-priority domains running on the same CPU. When the domRT finishes executing, it sends the output value to domN at time $t'+2p_N^x+p_i^x$. Nevertheless, the outgoing packet can again be delayed by $2p_N^x$ time units because of a short-unblocking activation of domN is also possible at this point. As a result, the following inequality must hold in worst case for the system to meet the deadline d_i associated with the incoming packet:

$$4p_N^x + p_i^x \leq d_i. \tag{1}$$

Let us now denote by d_{min} the minimum d_i among all deadlines in the system. Further, p_{min}^x is the Xen period of the domRT reacting to the packet with deadline d_{min}. If we consider $p_{min}^x = p_N^x$ in Eq. (1), the period of domN can be obtained as follows:

$$p_N^x \leq \frac{d_{min}}{5}. \tag{2}$$

This value of p_N^x satisfying the minimum deadline will also allow the system to meet longer deadlines. With p_N^x and d_i, we can use Eq. (1) to compute p_i^x for the different domRTs.

Now, if e_i denotes the worst-case execution time of the real-time task running in the i-th domRT—recall that there is only one task per domRT, we can set its slice to $s_i^x = e_i$.

To find a suitable value for s_N^x, we need to consider that neither the NIC nor its drivers in domN can distinguish between real-time and non-real-time packets. However, we can enforce that only a maximum number of k packets need to be processed at any point in time. As stated before, this can be done by allocating at most k domains to the system that receive/send packets over the network. In this case, s_N^x can be chosen as follows: $s_N^x = k \cdot e_N$, where e_N stands for the worst-case processing time of a packet.

V. Improving Xen's worst-case response time

The so-called short unblocking is a mechanism provided by SEDF to guarantee a fair scheduling of domains. That is, a domain can only be activated once within a configured time period independently of whether it has used its whole slice or not. In our case, this becomes a disadvantage because domN gets triggered by network packets. Each time that domN finishes processing all pending packets, it blocks itself for the remainder of its current period. Now, if packets arrive while domN is blocked, they will have to wait until the next possible activation irrespective of whether domN has used its complete slice in the current period or not.

This behavior of domN results in a quite pessimistic worst-case response time. Hence, we propose removing the short-unblocking activation from SEDF such that domN can utilize its whole slice s_N^x within its period p_N^x independently of its current state (waiting or blocked). In what follows, we denote by SEDF' this modified version of SEDF where short unblocking has been eliminated.

For a given maximum number of packets k, domN can now react to an arriving packet within at maximum p_N^x time units, if $s_N^x = k \cdot e_N$ holds where e_N is the worst-case processing time for a single packet. The worst-case response time to an incoming packet is now given by $2p_N^x + p_i^x$. As a result, a deadline d_i can be guaranteed if the following holds: $2p_N^x + p_i^x \leq d_i$. Using this inequality, it is possible to obtain periods and slices for all domains just proceeding analogously to case of the standard SEDF.

VI. A scheduler for real-time applications

The main disadvantage of SEDF and also of its modified version SEDF' is that they do not distinguish between domRTs running real-time tasks and domUs running non-critical tasks. To overcome this problem, we propose a new scheduler for Xen called PSEDF (Priority-based scheduling plus SEDF). The main difference to SEDF is that PSEDF allows configuring which domains are real-time (domRTs) and which are not. If a domRT is active, this is executed before any domU.

The domRTs are further scheduled under a preemptive fixed-priority policy, so priorities need to be assigned to them. As in the case of SEDF, we configure periods p_i^x and slices s_i^x for PSEDF as well. The non-real-time domUs will continue being scheduled as usual under SEDF.

Because domN is the interface between any domRT and the network, it has to be configured as a real-time domain. Further, all real-time domains (from the highest- to the lowest-priority domain), are released by domN. As a consequence, to allow for preemptive scheduling, domN needs to be assigned the highest

priority among all domRTs running on the same CPU/core as domN. On the other hand, domN can also be released by non-real-time packets directed to ordinary domains (domUs) in the system. After these non-real-time packets are processed, the corresponding domUs will be released, but they will not be executed as long as there is an active domRT.

Now, to find proper values of p_i^x and s_i^x for all domRTs in PSEDF, we can proceed as follows. We first assign the highest priority to domN. The remaining domRTs are then given priorities according to the DM (Deadline Monotonic) scheme and considering the d_i associated with the packets they have to react to. The shorter the deadline d_i, the higher the priority assigned to the corresponding domRT. This priority assignment is known to be optimal for the considered case where deadlines d_i are less than the periods of packets p_i [10]. Again, if d_{min} is the minimum deadline among all d_i, the corresponding domRT has the second highest priority after domN. The worst-case response time of this domain is illustrated in Figure 4. The domain reacting to the packet with d_{min} is released after domN finishes executing. In Figure 4, it is assumed that this domain finishes before the next activation of domN. However, its 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.

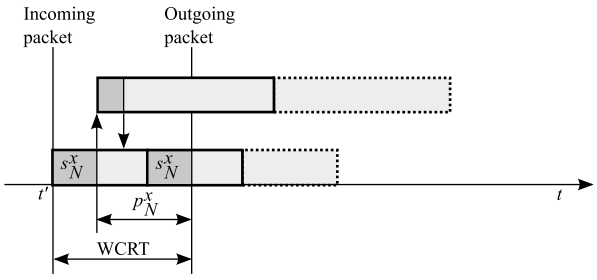

Figure 4. Worst-case response time (WCRT) under PSEDF

From Figure 4, the following inequality must hold for the system to meet d_{min}: $s_N^x + p_N^x \leq d_{min}$. We can now obtain $p_N^x = d_{min} - s_N^x$, where s_N^x is given by $k \cdot e_N$, i.e., the worst-case processing time of k packets.

For the remaining domRTs, we set $p_i^x = p_i$ and $s_i^x = e_i$, where p_i represents the minimum separation between two consecutive packets directed to the i-th domRT and e_i is the worst-case response time of the real-time task in this domain.

Now, we can compute the worst-case response times of the i-th domRT as follows [11], [12]: $t^{(c+1)} = s_i^x + \sum_{\forall j \in HP(i)} \left\lceil \frac{t^{(c)}}{p_j^x} \right\rceil s_j^x$, where $HP(i)$ denotes the subset of tasks with higher priority than or equal priority to the i-th domRT (i.e., domN and all domRTs reacting to packets with deadlines d_j such that $d_j \leq d_i$ according to the DM policy). This equation can be solved iteratively starting from $t^{(1)} = s_i^x$ and until $t^{(c+1)} = t^{(c)}$ is satisfied for some $c \geq 1$. This resulting value of $t^{(c+1)}$ is the worst-case response time of the i-th domRT which we denote by r_i^x. Now, all deadlines can be met, if the following condition holds for every i-th domRT in the system: $r_i^x \leq d_i$.

VII. Case Study: Electronic Stability Control

In this section, we consider the Electronic Stability Control (ESC) application. In principle, ESC improves the steering capability of a vehicle by minimizing blocking and skidding on the wheels.

In our setup, the Xen hypervisor (version 3.4) runs on an Intel Core 2 Duo platform with 2.16 GHz. A Debian Lenny OS was used for dom0 and domN for the reason that it provides the device drivers required in these two domains. Both dom0 and domN only need to react to events from the hardware and do not execute any application task. Further, the mini-os, a light-weight OS included in the Xen sources, runs in every domRT. We additionally use Debian Lenny for the non-real-time domUs which have here no access to the network.

A remote computer was connected to our setup via Ethernet and simulates the sensors generating packets for the different domRTs. A total number of four domRTs (domRT1 to domRT4) run on the system (i.e., $k = 4$), each of which stands for the control of one wheel in ESC. The sensing of wheels is performed every $2.5ms$ (i.e., packets arrive with $p_i = 2.5ms$). The deadline for reacting to incoming packets is $d_i = 1.5ms$ for all domRTs. Further, the worst-case execution time of every domRT is given by $e_i = 0.06ms$ whereas the worst-case processing time of a packet is $e_N = 0.02ms$.

In order to improve the reactiveness of the system, domN runs on a separate core than the domRTs. Now, to configure the standard SEDF, we can proceed as in Section IV. First, we set domN's slice to $s_N^x = k \cdot e_N = 0.08ms$. Its period can be obtained from Eq. (2), which results in $p_N^x = 0.3ms$. The slice of all domRTs is configured as $s_i^x = e_i = 0.06ms$ and their periods are all given by $p_i^x = 0.3ms$ according to Eq. (1). Although SEDF' and PSEDF allow relaxing the configuration of periods and slices, we use the same values as for SEDF for the sake of this comparison.

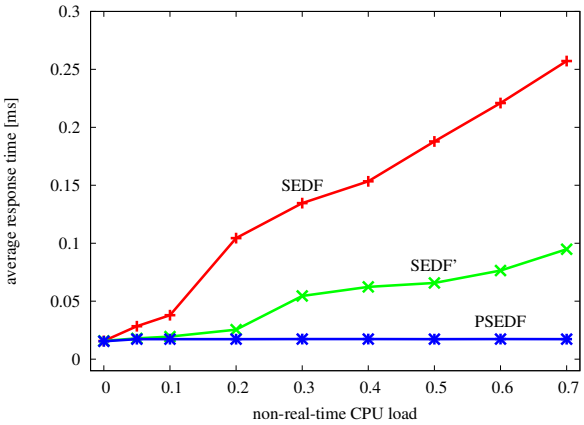

Figure 5. Average response time of domRT1

Figure 5 shows the average response times with respect to the non-real-time CPU load and for the case of domRT1 (the other domRTs have similar behavior). This is the processor utilization produced by non-real-time domains (ordinary domUs) on the different cores. For every measurement (i.e., for

every marker on the curves), 10,000 different packets were sent over the network to every domRT in the system. On average, the response times of domRT1 (and of the other domRTs) are below the $1.5ms$ deadline required by packets. However, only the response time under PSEDF remains constant as the non-real-time load increases. This characteristic of PSEDF results in a more predictable delay and less jitter for real-time tasks.

Figure 6 and Figure 7 show the different response times that we measured as the non-real-time CPU load increases. The vertical lines in these figures illustrate the variability of the response times for different CPU loads: the longer the line, the higher the variability. The maximum response times measured are indicated by dashes on the top of these lines. In the same manner, dashes at the bottom of the lines stand for the minimum response times that were measured. A third dash between the top and bottom ones shows the average response time on every line.

As shown in Figure 6 for SEDF, some domRT1 packets miss their deadlines as the non-real-time CPU increases to 70% (i.e., 0.7). Here, the response time of domRT1 reaches $3.5ms$. SEDF' presents a similar behavior to SEDF and is not shown for lack of space. Under PSEDF domRT1 can always meet its deadline as shown in Figure 7, where its response time never exceeds $0.09ms$.

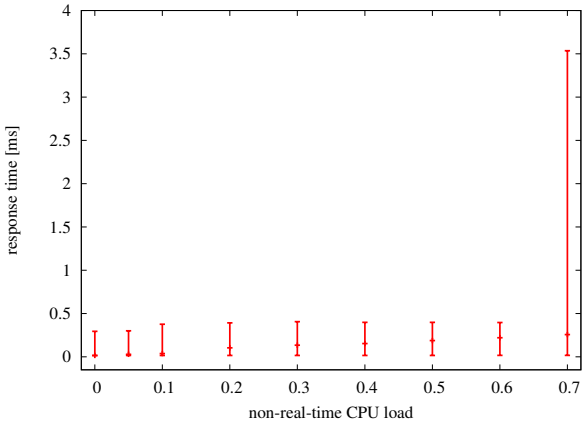

Figure 6. Response time of domRT1 under SEDF

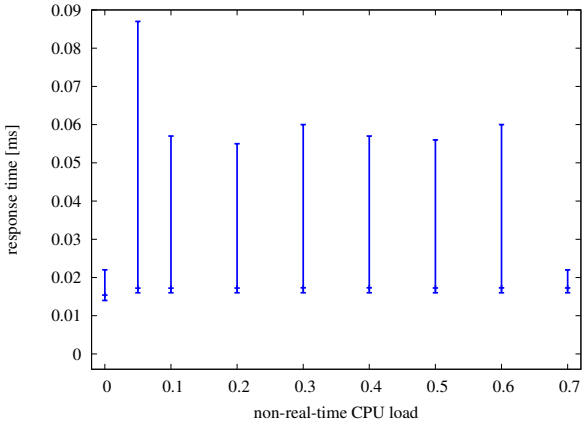

Figure 7. Response time of domRT1 under PSEDF

VIII. Concluding remarks

In this paper we studied the use of virtual machines (VMs) in the context of real-time applications such as those encountered in automotive control electronics. Further, we presented an approach based on the Xen hypervisor and analyzed the problems that arise when deadlines need to be guaranteed. We showed that the standard SEDF scheduler of Xen can be configured such that the different VMs (also called domains) can meet hard real-time deadlines. However, to ensure correct timing behavior in the worst case, SEDF incurs too much pessimism and yields a poor system utilization. In order to overcome this problem, a modified version of the SEDF scheduler (SEDF') was introduced, which exhibits an improved worst-case behavior.

Unfortunately, both SEDF and SEDF' are rather inefficient when both real-time and non-real-time workload needs to be scheduled. For this reason, we proposed a new scheduler called PSEDF (Priority-based scheduling plus SEDF), which separates real-time domains from non-real-time domains and, consequently, achieves a much lower delay and jitter. We illustrated this through a case study consisting of an Electronic Stability Control (ESC) system.

References

[1] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, 2006, pp. 342–362.

[2] 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*, 2007, pp. 126–136.

[3] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 2, pp. 42–51, 2007.

[4] 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*, 2008, pp. 1–10.

[5] 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*, 2009, pp. 101–110.

[6] 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*, 2009, pp. 111–120.

[7] R. Mangharam and M. Pajic, "Embedded virtual machines for robust wireless control systems," in *ICDCSW '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, 2009, pp. 38–43.

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

[9] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in hard real-time environments," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 40–61, 1973.

[10] J.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," in *Performance Evaluation*, vol. 2, 1982, pp. 237–250.

[11] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium*, December 1989, pp. 166–171.

[12] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, September 1993.