Flattening Hierarchical Scheduling

Adam Lackorzyński, Alexander Warg, Marcus Völp, Hermann Härtig
Technische Universität Dresden
Department of Computer Science
Operating Systems Group
{adam,warg,voelp,haertig}@os.inf.tu-dresden.de

ABSTRACT

Recently, the application of virtual-machine technology to integrate real-time systems into a single host has received significant attention and caused controversy. Drawing two examples from mixed-criticality systems, we demonstrate that current virtualization technology, which handles guest scheduling as a black box, is incompatible with this modern scheduling discipline. However, there is a simple solution by exporting sufficient information for the host scheduler to overcome this problem. We describe the problem, the modification required on the guest and show on the example of two practical real-time operating systems how flattening the hierarchical scheduling problem resolves the issue. We conclude by showing the limitations of our technique at the current state of our research.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Real-time and embedded systems

Keywords

Virtualization, real-time, scheduling, embedded systems

1. INTRODUCTION

Given that we want to integrate two or more real-time systems as guests onto a single host system, what are the challenges and opportunities that we are faced with? How do we maintain timeliness properties, such as meeting all deadlines in such a system when using virtual machines (VMs)? Different if not contradictory answers are given to those questions in the recent real-time literature. One side, for example well represented by Heiser's paper on "The Role of Virtualization in Embedded Systems" [14], claims a "mismatch between embedded-systems requirements and the virtual-machine model is evident in scheduling." He argues "The integrated nature of embedded systems requires that scheduling priorities of different subsystems must be interleaved. This is at odds with the concept of virtual machines." On the other side, an argument well presented by Sisu Xi et al. on RT-Xen [38] claims, that using bandwidth servers with small enough and frequently replenished budgets (1 ms) is sufficient for most real-time systems and that the performance overhead is negligible. In another recent publication, Masrur et al. motivate [25], "VMs are rewarding in the context of mixed-criticality applications to provide isolation between critical and non-critical tasks running on the same processor.", and "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."

The short dispute between the two sides following the presentation of Sisu Xi's paper at EMSOFT 2011 remained inconclusive. As a key point — apart from Heiser's doubt regarding Xen's suitability as a real-time kernel — issues with additional non-real-time tasks in *practical* real-time guests, causing problems, were brought forward. In this paper, we want to clarify the arguments.

We argue that at the root of the problem lies the insight that mixed-criticality systems across guests in virtual machines are not compatible with current virtualization technology. We describe two example task sets and show the limitations once they are integrated without using run-time knowledge of scheduling events in the task sets. We then describe a small modification of virtualization technology that allows to overcome these limitations: through a small enhancement of the scheduler in the guest operating system we export sufficient information about the guest task sets for the host scheduler to integrate these workloads onto a single system (e.g., by interleaving guest priorities). Having applied our approach in two practical real-time operating systems (RTOSs), we are confident that the modification to the guests are well in line with widely used virtualization techniques such as paravirtualization and the use of enlightened drivers for simplified virtualized devices. The contributions of this paper are:

- Two practical scheduling examples, which cannot be solved using a plain hierarchical scheduling approach.
- A mechanism to export relevant scheduling information from virtualized subsystems to allow the host to integrate these subsystems while preserving their timing requirements.

The remainder of this paper is organized as follows: after introducing the terminology, we construct two examples using the task sets of two virtual machines that are to be integrated into a single host in Section 3. We demonstrate that the assignment of a single budget (i.e., treating the VM-internal scheduling as a black box) does not suffice to meet the timing requirements of both VMs. Section 4 presents in greater detail how guest scheduling information can be exported to the host in order to flatten the hierarchical schedule and resolve the issue of integrating these subsystems. Section 5 demonstrates these modifications on the example of FreeRTOS and Linux-RT. In Section 6, we evaluate our approach with regards to the lines of code that had to be changed for these case studies and the performance implications that resulted form these changes. With Section 7 we conclude the paper by discussing the limitations and preliminary ideas how to overcome them.

2. TERMINOLOGY

Mixed-criticality systems [5] integrate tasks of different importance (criticality) into a single system while preserving run-time robustness. That is, they drop tasks in the order of increasing criticality if not all tasks can be serviced. In safety-critical settings, assurance in the execution of critical tasks is typically established by certifying more critical tasks at higher assurance levels and with tools that are more pessimistic in the characterization of these tasks. As usual, we assign each task τ_i the assurance (or criticality) level L_i up to which τ_i is certified and assume that it is also analyzed with the tools of all lower criticality levels. More precisely, we assume that for any two VMs (A and B), the criticality levels of the tasks that execute inside these VMs are comparable (i.e., $L_i^A \geq L_j^B \vee L_j^B \geq L_i^A$) and that all tasks are analyzed at all lower criticality levels (including those of tasks in other VMs).

For real-time systems, we obtain a task model for τ_i by replacing the worst-case execution time C_i of τ_i with a vector of worst-case execution times —one per criticality level— such that $\mathbb{C}_i(L) \geq \mathbb{C}_i(L')$ for $L \geq L'$. For better readability, we set $\mathbb{C}_i(L) := \mathbb{C}_i(L_i)$ for $L \geq L_i$. In the remainder of this paper, we shall use the sporadic task model. That is, a task τ_i is characterized by its period (minimal interarrival time), relative deadline, criticality level and worst-case execution time vector: $(T_i, D_i, L_i, \mathbb{C}_i)$. We assume implicitly constrained tasks (i.e., $D_i = T_i$). The *mixed-criticality* (MC) scheduling problem can be phrased as follows: *for each criticality level L, if no job of a task (τ_j) with criticality level $L_j \geq L$ executes longer than $\mathbb{C}_j(L)$, find a schedule such that all jobs of all tasks with criticality level $\geq L$ complete by their deadline.*

As usual, we say a schedule is *feasible* if it is a solution to the scheduling problem. A scheduler is *optimal* if it finds a feasible schedule whenever there exists one. The MC scheduling criterion gives rise to schedulers that deny the k^{th} job $\tau_{i,k}$ of a lower-criticality task τ_i its requested service if a job $\tau_{j,l}$ of a higher-criticality task τ_j executes longer than $\mathbb{C}_j(L_i)$. In this case, we say $\tau_{j,l}$ denies $\tau_{i,k}$ and call the earliest point in time by which $\tau_{j,l}$ has executed longer than $\mathbb{C}_j(L_i)$ without completing the *criticality decision point* of L_i.

Virtualization is a technology to run legacy systems, that is operating systems and their applications on a Virtual Machine Monitor (VMM), sometimes also called hypervisor. Following commonly used terminology, we refer to the entities provided by the VMM as Virtual Machines (VM), the legacy systems running in VMs as guest (operating) systems and the (operating) system running the VMM as host. We encounter two forms of virtualization technology, one that runs guests without any modification, sometimes called faithful virtualization, the other using small changes to guest operating systems, usually called paravirtualization. Examples for paravirtualization systems are Xen [4][1], L^{4}Linux [13] and OKLinux [28]. Faithfully virtualized systems require certain hardware properties that have been added to many common architectures, for example the Intel-VT and AMD-SVM to the x86 architectures. Embedded platforms like ARM will also get virtualization functionality [27]. Current virtualization technology comes in two architectural variants often referred to as type I and type II: type I (bare metal) uses a small kernel and runs the VMM and most of its hosting software on top of this kernel, examples being Xen [4], OKL4 Microvisor [16], the NOVA microhypervisor [33] and the VMware vSphere Hypervisor™ [36]. Type II (hosted) includes the VMM/Hypervisor in a fully-fledged operating system, examples are KVM [23] and VirtualBox [34] running on Linux. In this paper, we reserve the term hypervisor for the small kernel in type I systems and discuss our approach in the setting of a hypervisor-based system with deprivileged virtual machine monitors. Figure 1 illustrates this setting and highlights the components that are important for our work. Deprivileged VMMs execute guest operating systems and their applications inside virtual machines. The hypervisor offers virtual CPUs (vCPUs) as an abstraction of physical CPUs (pCPUs). The host scheduler in the hypervisor schedules vCPUs. On top of vCPUs, the guest scheduler runs the tasks of its VM. An extension to Type II VMMs is straightforward.

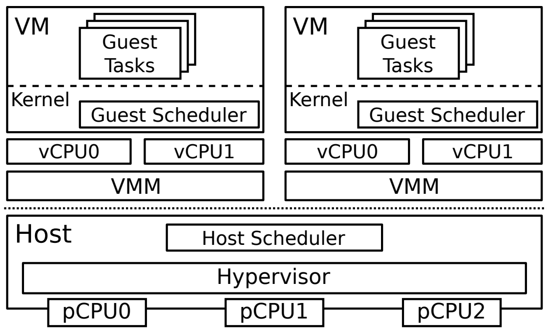

Figure 1: System architecture showing a system with two running VMs.

Scheduling in virtualized systems is typically strictly hierarchical. Each VM contains a scheduler whose responsibility it is to meet all the deadlines of the tasks that belong to this VM. The host scheduler then combines these guest schedules by assigning each VM a fraction of the CPU time. This fraction is typically characterized by a budget.

Besides having to meet the timeliness guarantees of the VMs, the host is also responsible for enforcing a certain degree of isolation between all guests. For safety-critical systems, the host must at least ensure that:

(R1) Scheduling in VM A must not depend on another VM

[1]Current versions of Xen also support faithful virtualization.

B providing information about the tasks it schedules; and

(R2) The scheduling and, in particular, the feasibility of a schedule in a VM A must not depend on the correctness of any component (including the scheduler) in any other VM B.

Because a guest scheduler in a virtual machine must be certified at least up to the criticality level of the tasks it schedules, we can relax the latter requirement for mixed-criticality systems:

(R2') The feasibility of a schedule produced by VM A for a certain criticality level L_i must not depend on the correctness of lower than L_i certified schedulers and components in other VMs.

Referring again to Figure 1, the hypervisor including the host scheduler must necessarily be trusted by all guests and hence certified at the highest criticality level. The VMMs must be trusted only up to the extent that the guest OS and its scheduler have to be trusted. That is, they must be certified at the highest criticality level of the tasks in the task set of the VM.

Other settings may impose further constraints on the isolation of VMs such as for instance the complete absence of covert channels [32]. However, for this paper, we restrict ourselves only to the above two constraints for mixed-criticality systems.

3. MOTIVATING EXAMPLES

The following two examples illustrate the need for multiple and, at the level of the host scheduler, interleaved budgets. In the first example, this need arises with as low as two mixed-criticality virtual machines (VM A and B), which schedule task sets comprised of two respectively three sporadic tasks with two criticality levels $HI > LO$ and different periods. All tasks in the second example share the same period of 8 units of time. The criticality levels of these tasks are $HI > MED > LO$. Table 1 contains the parameters for Example I, Table 2 for Example II. All units are normalized to the host system. We assume optimal mixed-criticality schedulers in both VMs and neglect all times spent in the host or guest operating systems. Both examples assume that the hypervisor schedules VMs strictly hierarchically. That is, it assigns exactly one budget to each of the two virtual machines VM A and VM B.

Sisu Xi et al. [38] propose to meet guest system timeliness properties by running each VM as a bandwidth server and allocate budgets proportional to the utilization. To ensure that even systems with very small periods can be handled, the budgets are allocated in very small portions. The paper reports that splitting up budgets into chunks of 1ms do not lead to significant performance problems, but state that with much smaller chunks the additional scheduling overhead becomes prohibitive. In our examples, we ignore the overheads introduced by small chunks and assume, arbitrarily small chunks can be selected without penalty. For our first example, we shall further allow the host scheduler to adjust budgets dynamically. In this way, the examples we give describe idealized systems.

VM	Task	T_i	L_i	WCET
A	τ_1	8	HI	$\mathbb{C}_1(HI) = 4$, $\mathbb{C}_1(LO) = 1$
	τ_2	4	LO	$\mathbb{C}_2(LO) = 1$
	τ_3	16	LO	$\mathbb{C}_3(LO) = 4$
B	τ_4	16	HI	$\mathbb{C}_4(HI) = 6$, $\mathbb{C}_4(LO) = 2$
	τ_5	4	LO	$\mathbb{C}_5(LO) = 1$

Table 1: Task parameters for Example I.

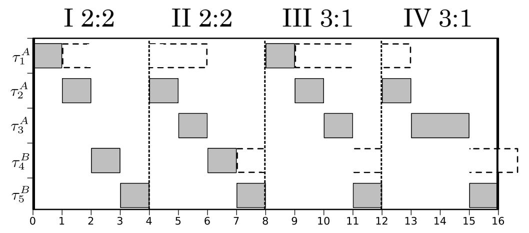

Figure 2: Schedule for the simultaneous release of the task set in Table 1 on top of a mixed-criticality hypervisor. Filled bars show LO WCETs ($\mathbb{C}_i(LO)$), dashed bars show the time $\mathbb{C}_i(HI)-\mathbb{C}_i(LO)$ that is required to complete high tasks that do not complete before $\mathbb{C}_i(LO)$.

3.1 Example I

Figure 2 shows a schedule for the simultaneous release of the task sets of Example I. We can distinguish four phases which correspond to the periods of the tasks τ_2^A and τ_5^B. Irrespective of when the hypervisor switches to VM A or VM B in the first phase, VM A and VM B cannot both execute τ_1^A for $\mathbb{C}_1(LO)$ and τ_4^B for $\mathbb{C}_4(LO)$ while meeting the low deadlines of τ_2^A and τ_5^B if both τ_1^A and τ_4^B would complete before their criticality decision points. For the same reason τ_1^A cannot completely be delayed to Phase II. As a consequence, A needs at least a budget of 2 in Phase I and B a budget of at least 1. Figure 2 illustrates the case where both VMs receive the same budget of length 2. It is easy to see that the arguments that we give hold also for all other sensible budget assignments. With a $2:2$ budget assignment in Phase I, VM A can execute τ_1^A for one unit of time, which allows the scheduler in VM A to decide whether τ_2 is released or, if τ_1^A does not complete by $\mathbb{C}_1(LO)$, whether to allow further execution of τ_1^A. Remember the mixed-criticality scheduling rule gives no further guarantees to LO tasks if a HI task executes longer than its LO WCET. VM B has to execute τ_5^B because VM isolation (R1) prevents B's scheduler from knowing whether or not τ_1^A has already completed by $\mathbb{C}_1(LO)$. It may drop τ_5^B only after τ_4^B has executed longer than $\mathbb{C}_4(LO)$. Following Baruah et al. [6], we call the situation caused by τ_5^B *criticality inversion*. Following a $2:2$ budget in Phase I, A needs at least a budget of 2 in Phase II to guarantee completion of τ_1^A in the situation when τ_1^A did not complete by $\mathbb{C}_1(LO)$. An assignment of a larger budget to A is counterproductive as this would result in a remaining LO utilization for the two remaining phases of 1 and a remaining HI utilization of 9/8 (i.e., > 1). If both τ_1^A and τ_4^B are not completed by their LO WCETs, a completion by their HI WCETs can therefore no longer be guaranteed. The key insight that completes this example is that any execution of τ_3^A for longer than 1 unit of time may result in τ_4^B missing its deadline if it has executed longer than $\mathbb{C}_4(LO)$. However, without knowing the progress of τ_4^B, VM A cannot decide whether or not to execute τ_3^A at time 14 in Phase IV. Following the same line of argumentation, it is easy to see that also for other budget assignments the taskset in Table 1 is not feasible for mixed-criticality hypervisors that assign only one budget per VM. A priority assignment π with $\pi(\tau_3^A) < \pi(\tau_4^B) < \pi(\tau_5^B) \leq \pi(\tau_2^A) \leq \pi(\tau_1^A)$, that is two budgets for VM A to execute τ_3^A and τ_2^A, τ_1^A interleaved with the tasks of VM B, however leads to a feasible (fixed-priority) schedule if we assume that VM A stops τ_2^A latest after $\mathbb{C}_2(LO) = 1$ and that it switches to its low budget if τ_1^A completes before $\mathbb{C}_1(LO)$. To fulfill the Isolation Requirement (R2'), we do not have to require a similar precaution for τ_3^A.

3.2 Example II

Example II demonstrates the possibility of infeasible schedules when integrating two task sets with two tasks each and where all tasks share a single global strict period of 8 units of time. Table 2 contains the parameters of these task sets.

VM	Task	T_i	L_i	WCET
A	τ_1	8	HI	$\mathbb{C}_1(HI) = 4$, $\mathbb{C}_1(MED) = 2$, $\mathbb{C}_1(LO) = 2$
	τ_2	8	LO	$\mathbb{C}_2(LO) = 1$
B	τ_3	8	HI	$\mathbb{C}_3(HI) = 4$, $\mathbb{C}_3(MED) = 2$, $\mathbb{C}_3(LO) = 2$
	τ_4	8	MED	$\mathbb{C}_4(MED) = 3$, $\mathbb{C}_4(LO) = 3$

Table 2: Task parameters for Example II.

To ensure that all high-criticality tasks meet their deadlines, a minimum of four execution units per period must be allocated to each VM. Otherwise, τ_1^A, τ_3^B, or both may miss their deadline if they fully need the execution time $\mathbb{C}_i(HI) = 4$ ($i \in \{1, 2\}$) determined by the high WCET analysis tools. However, if τ_1^A executes only for $\mathbb{C}_1(MED) = 2$ of the four allocated units, the local scheduler of VM A will run τ_2^A on the remaining time (2 units) because it has only a local view on its task set (see Isolation Requirement (R1)). If τ_2^A then uses more than its low-criticality execution time $\mathbb{C}_2(LO) = 1$, for example because of an error or because the scheduler in VM A does not enforce $\mathbb{C}$ for low-criticality tasks, then the medium-criticality task τ_4^B may miss its deadline if both τ_3^B requires $\mathbb{C}_3(MED) = 2$ units and if τ_4^B requires the third unit as predicted by the medium WCET analysis tool ($\mathbb{C}_4(MED) = 3$). Notice, the violation of the mixed-criticality scheduling criterion does not depend on a particular guest or host scheduler but merely on the assigned budgets. For as long as the host scheduler allows VM A to consume 4 units, τ_3^B or τ_4^B may miss their deadlines because the remaining 4 units do not always suffice for $\mathbb{C}_3(MED) + \mathbb{C}_4(MED)$.

An assignment of multiple interleaved budgets again resolves this violation. Table 3 lists the global (i.e., host) priorities, parameters and tasks to run on these budgets. Like for Example I, we have to assume the scheduler in VM A to switch to its low budget if τ_1^A completes before $\mathbb{C}_1(MED)$.

Our general approach to enable the interleaved execution of virtual machines is to flatten the hierarchical scheduling problem by exporting some parts of the guest scheduling to the host. At the current state of our research, exporting the required information to the host scheduler requires small modifications of the guest operating system. Moreover, we have to require that the results of the real-time analysis and hence the parameters of all real-time tasks in the task sets of all guests are available to the host scheduler for the purpose of a global admission. We believe that both requirements are adequate given that the host has to guarantee that all VMs meet the deadlines of all their tasks. In particular, deployment of binary guests remains possible after they have been enlightened for hosts that support flattening.

VM	Budget	Time	Priority	Tasks to run on
A	A_1	4	1	τ_1^A
	A_2	1	3	τ_2^A
B	B_1	5	2	τ_3^B, τ_4^B

Table 3: Example II priority/budget allocation (smaller numbers denote higher priority).

4. EXPORTING GUEST SCHEDULING TO THE HOST SCHEDULER

A scheduling property common to virtualization technologies is that schedulers in guest operating systems operate independently from the schedulers in other guests and in the host: The host schedules VMs by selecting the budget that it has associated with the virtualized CPU (vCPU). The schedulers in the guest operating systems make use of this budget to schedule their tasks.

In our approach, we attenuate this strict separation of schedulers by introducing an interface in the host, which allows VMs to allocate multiple budgets and to switch between these budgets on their demand. More precisely, during the startup phase of a virtual machine or later upon request from the VM, the host scheduler allocates the budgets in the form of *scheduling contexts* (SCs). After it has validated the schedulability of the system, it attaches the SCs to the virtual CPUs of the requesting VM. A virtual CPU (vCPU) may have multiple SCs attached in which case the guest can select the SC to run on. The set of parameters of a SC includes at least a global priority π and a budget b that is subject to some replenishment rule. In this sense, our approach is generally applicable to all host scheduling policies that select VMs (in our case SCs) from the set of highest prioritized VMs (SCs) with a positive remaining budget. In particular, our approach extends to scheduling schemes such as RT-Xen with small and frequently replenished budgets (if we limit the selection to the highest prioritized SCs) and to more classical global or partitioned fixed-priority schemes where each SC has a period T to denote when budgets are replenished and hence after which time the next job of a task is released. Of course, for a partitioning host scheduler, the physical CPU becomes an additional parameter of an SC and all SCs that are associated with a vCPU must agree on this parameter.

In addition to an interface for requesting an SC, which the host scheduler is able to validate before it associates this SC with a vCPU of the VM, the interface offered to guest schedulers consists of the following two functions:

`set_sc(id_sc)` deactivates the current SC of the vCPU that invokes this function and activates the SC referred to by `id_sc`. Identifiers like `id_sc` are local to the invoking VM and have to be translated by the host to the actual SC. During this translation, the host scheduler also validates that the referred SC is associated with the requesting vCPU of this VM.

`register_event(id_sc, event, function)` associates the specified event (e.g., an interrupt) with the SC. Upon occurrence of this event, the host activates this SC and, if this SC gets selected, invokes the VM at the specified function.

We defer the discussion of `register_event(id_sc, event, function)` and its use for triggering interrupt service routines to Section 4.2. For now, let us focus on `set_sc(id_sc)` to see how voluntary switches between SCs help resolve the scheduling problems raised in Section 3.

To keep the operation of the above two functions simple, it is convenient to assign each vCPU a default scheduling context (Default-SC). The priority of this Default-SC is the lowest host priority level. The budgets and in particular the replenishment and switching times of course depend on the host scheduling policy. However, to ensure progress of the non-real-time tasks of the individual VMs a Round-Robin or weighted Round-Robin scheme suggests itself. That is, VMs running on the Default-SC receive an equal or weighted proportional share of the time that remains after scheduling the real-time workload of all VMs.

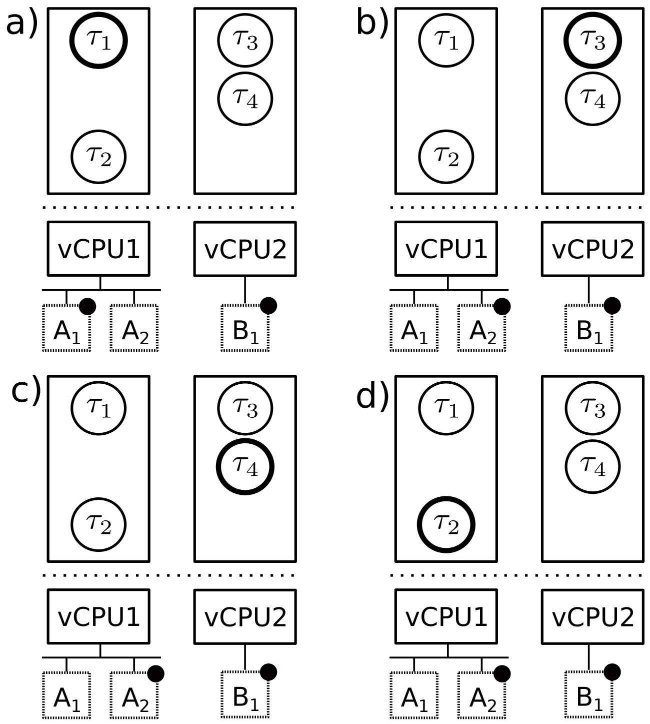

Figure 3: Step-by-step illustration of a sequence of scheduler context switches (`set_sc()`). Bold circles indicate the active task and the currently active and highest prioritized scheduling contexts.

Figure 3 illustrates the use of `set_sc()` to resolve the mixed-criticality scheduling problem of Example II. Of course, the three budgets in Table 3 already resolve this problem if the host creates three SCs with the same parameters as the budgets in this table and if the guest scheduler in VM A invokes `set_sc()` to switch from the SC for budget A_1 to the SC for budget A_2 in the event that τ_1^A completes before $\mathbb{C}_1(LO)$. However, more insights in our approach can be drawn from a discussion of this scenario with two budgets for each of the two VMs (i.e., one SC (SC_i) for each of the four tasks (τ_i) parametrized as described in Table 2). The priorities π of these SCs are $\pi(SC_1^A) = \pi(SC_3^B) = 1$, $\pi(SC_4^B) = 2$, and $\pi(SC_2^A) = 3$. Smaller numbers stand for higher priorities. We explain the solution for Example II for the simultaneous release of all tasks. At time 0 (relative to this simultaneous release), both SC_1^A for τ_1^A and SC_3^B for τ_3^B are active. Irrespective of the host scheduling policy, both VMs receive a share of 4 units at the highest priority to complete τ_1^A and τ_3^B in the event that not both complete before $\mathbb{C}_1(LO)$, respectively before $\mathbb{C}_3(MED)$. If one of these tasks completes latest after 2 units, the corresponding guest scheduler invokes `set_sc(`SC_2^A`)` (for VM A) or `set_sc(`SC_4^B`)` (for VM B) to switch to the respective lower prioritized scheduling context. Fig. 3a and b depict this situation for the case where τ_1^A completes first. After both VMs have dropped to their lower prioritized budgets (Fig. 3c), SC_4^B has a higher priority than SC_2^A, which allows τ_4^B to complete even in the case that τ_1^A exceeds its budget. Finally τ_2^A runs (Fig. 3d), completing the sequence.

4.1 Guest Task to Host SC Mapping

The two examples and in particular the two solutions to Example II show that the number of exported SCs heavily depends on the host and guest scheduling policies and on the workload to be scheduled. For fixed-priority schedulers in all guests and in the host and for criticality monotonic priority assignment [6], a relatively easy mapping of guest tasks to host SCs is demonstrated in the 4 SC variant of Example II: The scheduling parameters of every task are directly exported and the local priorities are interleaved in such as way that criticality levels are preserved. That is, the priorities of all high-criticality tasks are strictly higher than the priorities of all medium-criticality tasks and all low-criticality tasks, etc. The interleaving within these priority bands must of course be validated by the admission test performed by the host.

The two examples also show possibilities for reducing the number of SCs. For example, Table 3 shows a mapping for Example II with one SC per criticality level. As the focus of this paper is on introducing an easy to use mechanism for integrating VM workloads by flattening the hierarchical scheduling problem, we leave an exhaustive analysis of guest task to host SC mapping for future work.

4.2 Interrupt Service Routines

We now turn our attention to the second function `register_event()` and on one specific implementation detail of VM internal scheduling: interrupt service routines. At the same time, we relax our assumption that host and guest scheduling comes at no cost.

Scheduling decisions in VMs are triggered by injecting interrupts such as timer or device interrupts. Upon receiving these interrupts, the guest runs the corresponding interrupt service routine to decide how to react on these asynchronous events and how to adjust the VM internal scheduling. For example, a guest with one-shot timer may have programmed a timer to the minimum of the absolute deadline of the currently active task and to the point in time when the budget of this task will be depleted. Upon receiving this timer, the timer service routine invokes the scheduler to select the next task to switch to. From the perspective of the VM, this service routine runs non-preemptively, that is, effectively at a priority above the priorities of all tasks. However, to limit the interference from these events on other VMs and to maintain the principle operation of the VMs in the first place, the host must be able to integrate interrupts into the SC scheme and activate SCs in the course of injecting interrupts into the VM. The `register_event()` function serves the purpose of informing the host about which SC to activate for which event. The connection between the interrupt and the to-be-invoked service routine is already known to today's VMMs. Our implementation therefore differs in that the SC-to-event assignment and the event-to-function assignment are realized as two separate functions.

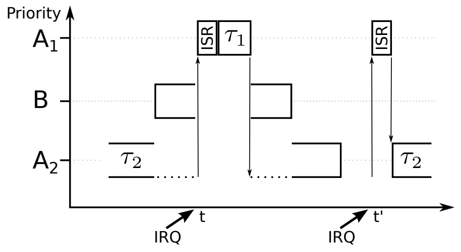

Figure 4: Scheduling context activation at the occurrence of asynchronous events such as interrupts or the expiration of a timer. The interrupt service routine always runs on A_1 until the guest scheduler, which it invokes, decides which task to run.

Figure 4 shows a detail of the release of τ_1 and the activation of the interrupt service routine that follows. Fig. 5 presents the same step-by-step illustration as Fig. 3 but for the scenario of Fig. 4, which includes interrupts. At time t, the host receives an interrupt (IRQ), which triggers the release of τ_1 and later (at time t') of τ_2 in VM A. The hypervisor therefore switches to the VMM of VM A, which in turn injects the interrupt into this VM. Because the interrupt is associated with SC_1^A (i.e., budget A_1), the interrupt service routine always runs on this highest prioritized scheduling context. In the first situation (at time t), the guest scheduler releases τ_1 and drops to SC_2^A (i.e., budget A_2) only if τ_1 completed before $\mathbb{C}_1(LO)$. In the second event (at time t'), the guest scheduler immediately switches to SC_2^A to release the second job shown for τ_2.

4.3 Required Guest OS Modifications

At the current state of our research, small modifications to guest operating systems are required to make use of multiple scheduling contexts. For an arbitrary guest, we have to add the following functionality:

- After every priority change, the corresponding SC must be activated by means of `set_sc()` (if this SC is not already active). A common place where this call to `set_sc()` must be made is after the invocation of the scheduler before the code for switching tasks is invoked.
- For every interrupt service routine that has an SC associated with it, `set_sc()` must be called when the service routine returns to the currently active task in the guest. Otherwise, the task would continue executing at the level of the interrupt routine, which in many guests corresponds to a non-preemptive execution.

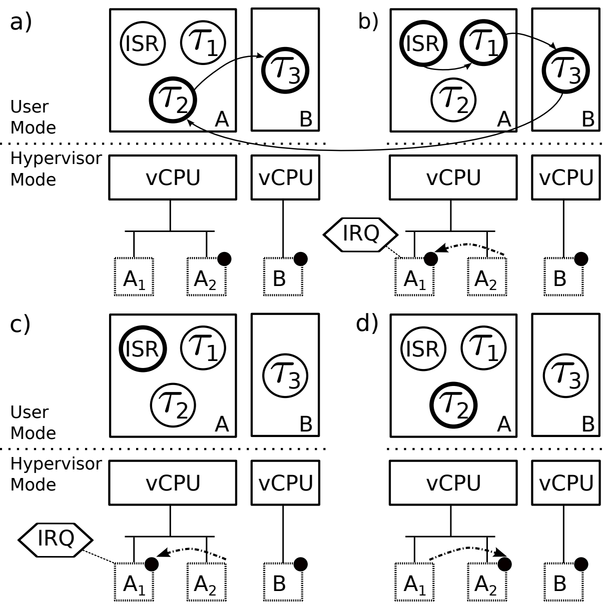

Figure 5: Example sequence of actions as depicted in Figure 4. a) and b) illustrate the first ISR invocation, after τ_3 has finished, the state of a) is restore. The second ISR invocation is depicted by c) and d).

For specific guests a subset of these modifications may suffice depending on the features the guest already provides. In the next section we discuss how the functionality is added to two real-time guest operating systems (RTOSs).

5. CASE STUDIES

For our experiments, we implemented flattening support in the Fiasco.OC microkernel [1] and into two operating systems, which we run as guests. Fiasco offers virtualization support in the form of specialized threads called *virtual CPUs* (vCPUs), which in addition to the user accessible registers provides also storage for the state that is typically accessible only from the kernel. That is, vCPUs abstract from the implementation details of the virtualization support in modern hardware architecture (for example, from the Virtual Machine Control Structure (VMCS) of Intel x86 CPUs [17]). Virtualization events are, as common for microkernel-based systems, reflected as messages and delivered with the inter-process communication (IPC) mechanism to application level virtual machine monitors. Therefore, taken together, the unprivileged VMMs and the microkernel serve as a type I VMM in the scenario depicted in Figure 1. The VMM for faithful virtualization was built using the Palacios VMM library [24].

Fiasco implements a generic interface, which allows application level schedulers to set the parameters for the in-kernel scheduling policy. Building upon the internal infrastructure of the kernel, exposing multiple SCs and attaching them to interrupts was straightforward. Regarding the handling of multiple SCs by the in-kernel scheduler, there are two principle ways this can be accomplished: Either the scheduler selects threads (or vCPUs) based on their position in the ready list, which is determined by the highest prioritized active SC that is attached to this thread; or active SCs occupy the ready list in the first place and the scheduler selects first the SC and then the thread attached to it. Both variants have their benefits and drawbacks depending on the frequency of SC switches and on the likelihood of finding a blocked thread, which in the second variant implies a lazy dequeue operation of all its SCs.

For this paper, we used Fiasco's fixed-priority scheduler. That is, the parameters for SCs are periods (implicit deadlines), budgets and priorities and SCs are scheduled according to these fixed priorities with a Round Robin scheduler for those SCs with the same priority.

In the remainder of this section, we describe some important details on the changes required to two popular operating systems: FreeRTOS and Linux-RT. In the examples, `set_sc(ID)` denotes a switch of the SC of the calling vCPU. The ID identifies the SC for activation. The specific implementation varies because of the different implementations of hypercalls. Faithfully virtualized guest operating systems invoke the hypervisor and hence the VMM through a special machine instruction (`vmcall` on x86), which in turn the hypervisor reflects as an IPC message to the VMM. Paravirtualized guests can directly invoke system-calls of the hypervisor to communicate with their VMM. As both techniques build upon vCPUs, the detailed implementation is transparent to the host scheduler.

Depending on the host scheduler, it is possible to implement optimizations for deferred scheduling when `set_sc(ID)` gets invoked. For example, the switch to a higher prioritized SC may be deferred to the point in time of the next preemption or the switch could be dropped if the current SC gets activated.

To simplify the presentation of our guest OS modifications, we restrict our examples to two SCs: the Default-SC and one SC to handle high priority work. Elevation of a VM to a higher host priority (i.e., to the SC for the high priority work) is triggered by a single interrupt. Switching away from this SC targets the Default-SC. An extension to multiple different SCs at different priorities is straightforward.

5.1 FreeRTOS

The real-time operating system FreeRTOS typically comes with no memory protection between tasks[2]. The scheduler in FreeRTOS allows multiple tasks to run concurrently at static priorities. Preemptive and non-preemptive variants of this scheduler exist. In our implementation, we exclusively use the preemptive version, which calls the internal scheduler for each timer tick. This timer tick is associated with the high priority SC. After the new task to be run has been chosen, the function listed in Figure 6 is called, handing over the new priority. Referring to FreeRTOS v6 and v7, the function must be added as `xvPortPostSchedule(uxTopReadyPriority)` in the function `vTaskSwitchContext()` after the while loop, which calculates the new priority. The function uses a barrier priority `RT_BASE_PRIO` to split FreeRTOS tasks into a real-time and a time-sharing category. Based on this barrier priority, the FreeRTOS scheduler decides whether the selected task should continue to use the high priority SC or fall down to the default one (with `set_sc(ID_SC_DEFAULT)`).

[2] FreeRTOS support memory protection units (MPUs) however their use is not typical for the application fields of FreeRTOS.

```
void xvPortPostSchedule(unsigned prio)
{
  if (prio < RT_BASE_PRIO)
    set_sc(ID_SC_DEFAULT);
}
```

Figure 6: SC switching function for FreeRTOS.

5.2 Linux

Linux is a widely used and popular operating system that can be used for a wide range of use cases. With the ongoing work on improving the preemptiveness of the kernel and with the merge of a significant part of the Linux-RT patch, it is also increasingly used for real-time workloads. Linux priorities are divided into a range for time-sharing and an exclusive range for real-time processes. This distinction makes the implementation of the SC switching function straightforward as listed in Figure 7. Referring to Linux kernel version 3.3, the function `post_sched_sc(current)` is called within the function `finish_task_switch()` in `kernel/sched/core.c`[3].

To switch back to the Default-SC in the case no scheduling decision will be made after an interrupt has occurred, we introduce the function `irq_no_sched()`. It is called in the code paths for exiting interrupts as shown in Figure 8. Referring to Linux 3.3, a convenient location to call `irq_no_sched()` is the function `irq_exit()` in `kernel/softirq.c`. The synchronization in `irq_no_sched()` is required to atomically check the rescheduling condition with the actual operation. Otherwise, if an interrupt occurred meanwhile, a possible RT-task would be switched back to the default SC.

Using and mapping multiple RT-tasks within Linux to different SCs is also possible by enhancing the two presented functions.

6. EVALUATION

6.1 Further Use Cases of Flattening

Although we draw examples from mixed-criticality scheduling, the application field of flattening and of the mechanism to switch between multiple interleaved-prioritized SCs is broader. We now introduce two further use cases, which benefit from these enhancements.

The "rare-alarm" example includes two subsystems: $S1$ consisting of τ_{alarm} and τ_{book} and $S2$ consisting of τ_2^M. τ_{alarm} is a sporadic task and τ_{book} is a best-effort task, which gathers statistics for maintenance purposes. τ_{alarm} has a low minimum inter-arrival time and a WCET nearly as high as the inter-arrival time leading to very high utilization. τ_2^M is a high-utilization task. τ_{alarm} has an extremely low probability for high-frequency alarm showers, but the importance

[3] `kernel/sched.c` prior to version 3.3.

```
void post_sched_sc(struct task_struct *p)
{
        if (!rt_task(p))
               set_sc(ID_SC_DEFAULT);
}
```

Figure 7: Post scheduling function for Linux.

```
void irq_no_sched(void)
{
        unsigned long flags;
        local_irq_save(flags);
        if (!need_resched()
            && !rt_task(current))
                set_sc(ID_SC_DEFAULT);
        local_irq_restore(flags);
}
```

Figure 8: Function to be called in case no scheduling decision has been made upon interrupts.

of meeting the deadline in these rare situations is very high. τ_2^M is much less important than τ_{alarm}, but more important than the maintenance task τ_{book}. Without guest-host interaction in the form described in this paper, τ_{book} will use budgets reserved for the rare occasions of alarm showers and block τ_2^M in most situations.

Another example are systems with interactive processes. Common desktop operating systems have means to discover highly interactive tasks and use this information to boost their priorities. If several virtual machines with such interactive tasks change focus, single-budget allocation schemes cannot consider these priority boosts without knowledge of guest-task priorities. A way of solving this issue is to use a high switching frequency between the VMs. However, this increases the overhead as the VM switching has to be done periodically. The assignment of multiple (keyboard and mouse) interrupt triggered SCs allows to minimize these switches to when they are needed.

6.2 Guest Modifications

To quantify the changes required on the guest operating system, we count the number of lines that had to be added to each of the guest variants for switching SCs. We count here only the changes required for SC switching, not for paravirtualization itself. Moreover, we assume the availability of basic infrastructure code, for example, to issue hypervisor calls. Table 4 summarizes these changes.

FreeRTOS requires only the addition of one function plus the calls activating this function after scheduling decisions. Both sum up to a 10 line patch. We applied the two virtualization techniques to the same Linux version. The same modification was required to switch SCs after scheduling. That is, the modification differs only in the IRQ exit paths, however, not in the amount of code that has to be added.

SC setup requires additional code during the startup of the VM or in the event that additional real-time tasks arrive.

Guest	Added Source Code Lines
FreeRTOS (para-virtualized)	10
Para-virtualized Linux	22
Fully-virtualized Linux	22

Table 4: Added source lines of code to each guest variant.

However, because this code is largely dependent on the guest and host scheduling policy, lines-of-code statistics would not easily be comparable.

6.3 Runtime Costs

The modifications to the guest operating system incur no measurable overhead when SCs are not switched. Assuming that `set_sc()` will call out only when the SC must actually be changed, the overhead is negligible because in this situation `set_sc()` boils down to a simple check for equality between the IDs of the current and the targeted SC. In addition, inlining of `set_sc()` eliminates the function call overhead.

6.4 Scheduler Context Activation Latency

The costs for actually switching to a new SC are dominated by the costs to call out of the VM into the hypervisor. We have evaluated these costs in a benchmark running on an AMD Phenom 8450 based system clocked at 2.1GHz.

	Cycles	µs
Para-virtualized	795	0.4
Fully-virtualized	5863	2.8

Table 5: Average SC activation latencies. Cycles were measured using processor's time stamp counter (TSC), times are given for comparison.

Table 5 lists the average SC activation cost for the two types of virtualization used. A call from within a faithfully virtualized environment is more costly than in a paravirtualized guest. In a faithfully virtualized system, the hypercall consists of a `vmcall` instruction. Calls our of paravirtualized guests can directly use the host system call interface and hence benefit from the faster kernel entry.

7. LIMITATIONS AND FUTURE WORK

Currently, we require explicit call-outs of the guests to trigger SC switches. This of course only works if the source code of the guest is available for modification and can be recompiled. As this possibility might not always exist, a way to run unmodified guests would be desirable. The challenge of such a solution is to detect the locations where priorities change within the guest. Whether or not these limitation can be overcome, for example using techniques such as binary rewriting, has to be subject to further research. Considerable knowledge of guest OS kernels is required to identify those locations and it remains open whether or not this can be achieved.

For the overall system an admission must be performed which decides whether a VM, or more general, a subsystem, can be admitted to run on the system. Due to a possibly dynamic nature of subsystems, the admission should be done at runtime. Finding ways to dynamically generate a set of scheduling parameters that can be handled practically is a challenging task. Connected with that is the challenge of extracting runtime information out of guests by techniques such as profiling and event logging.

So far our guests and the host use static-priority based scheduling, which allows the construction of a mapping from guest to host priorities and especially to consider priorities across multiple VMs. Dynamic-priority based algorithms in the guest do not map as easily to a host with static priorities. Guests may use algorithms such as Earliest Deadline First (EDF) if the task set, which is scheduled according to this algorithm, is only exposed to the host as a single parameter set. Interleaving tasks scheduled with EDF across VMs or treating tasks from a single VM differently would require an EDF scheduler in the host as well. This also raises the challenge of mapping EDF guest tasks to another set of EDF host parameters.

It remains to be seen whether the presented technique is applicable for other resources, such as disk, network, or graphics. Generally, multiple scheduling parameters are not only useful for VMs but can also be beneficial for work loads with – for example – differing quality levels. Primary targets might be video decoding and game engines.

8. RELATED WORK

Virtualization of timing-critical systems has been considered before. In the commercial context several vendors offer real-time operating systems that also allow running virtualized guest operating systems in compartments, among them being VirtualLogix [35], PikeOS [29] and OKL4 [28]. The virtualized operating system needs to be adopted to run in those environments. For example, this has been implemented on a commercially deployed mobile phone, running a Linux and the UMTS software stack side by side [15]. Forthcoming hardware support for virtualization on the ARM architecture will improve currently used virtualization techniques [27]. Time partitioning is a popular method to ensure temporal isolation. PikeOS uses a certified time partitioning technique to isolate subsystems and those partitions can run a paravirtualized Linux version [18]. In the research community virtualization in embedded systems has also been worked on. Proteus [3, 19] is using periodic tasksets to fully virtualize operating systems without the need to modify them on the PowerPC architecture. Xen [4] is a popular virtualization solution for servers but has also been ported to the ARM architecture [37] and then evaluated for real-time use [7, 31]. Kinebuchi et al. [20] implemented a hypervisor to host an RTOS and commodity operating systems and evaluated against an L4 solution. Using an L4-based system, it has been researched which costs are induced by introducing address space isolation in Linux-RT environments [26] by building real-time applications using an improved environment that implements real-time threads using host threads. Using specific features of ARM CPUs and their utility in real-time environments has been evaluated in [12]. Kiszka evaluated the use of Linux and KVM as a real-time hypervisor [21] by measuring scheduling latencies and introducing a paravirtualized scheduling interface for guests. Follow-up work examined Linux-KVM for use in embedded systems [22]. Zuo et al. also examined a KVM-based solution for low-latency virtualization [40]. Cucinotta et al. applied hierarchical real-time theory in a system with KVM by scheduling VMs with reservations [8]. In IRMOS an EDF scheduler in Linux is used to schedule KVM VMs with a guaranteed share [9].

Virtualization inevitably includes stacked scheduling which has been researched in form of hierarchical schedulers. Real-time tasks are grouped into applications with their own local scheduler, multiple applications run on the host. Applications are modelled as servers. Zhang et al. investigated hierarchical scheduling on single processors with earliest deadline first schedule being used as the local scheduler and EDF or fixed-priority as the global one, using independent local tasks [39]. Saewong et al. analyse fixed-priority scheduling in hierarchical configurations [30]. Feng et al. propose a multi-level approach to partition subsystems [11]. Davis et al. investigate hierarchical scheduling on uni-processors with fixed-priority preemptive scheduling in both the global and local schedulers [10]. Time-demand analysis techniques [2] can be used to express waiting times of subsystems.

9. CONCLUSIONS

Some scheduling problems that arise from the integration of real-time systems as guests in a real-time capable virtualization system cannot be solved without VMM control over guest tasks. More generally, in a hierarchical scheduling setup an outer stage must have knowledge about an inner stage configuration, possibly flattening the scheduling. The core of the problem turns out to be a mixed-criticality problem. We showed, how central control exercised over guests tasks can be used to overcome these problems. We also demonstrated how such controlling mechanisms can be provided using several scheduling contexts for real-time guest schedulers in the host scheduler. At the current stage of our research, tiny modifications of the guest OS are needed and a mapping must exist from guest to host scheduling.

10. REFERENCES

[1] Fiasco.OC. `http://tudos.org/fiasco`.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[3] D. Baldin and T. Kerstan. Proteus, a hybrid virtualization platform for embedded systems. In A. Rettberg and F. J. Rammig, editors, *Analysis, Architectures and Modelling of Embedded Systems*. IFIP WG 10.5, Springer-Verlag, 14 - 16 Sept. 2009.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *Computers, IEEE Transactions on*, PP(99):1, 2011.

[6] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34 –43, 29 2011-dec. 2 2011.

[7] Chuck Yoo and team. Real-Time and VMM -

Real-Time Xen for Embedded Devices. `http://www.xen.org/files/xensummit_oracle09/RealTime.pdf`.

[8] T. Cucinotta, G. Anastasi, and L. Abeni. Respecting temporal constraints in virtualised services. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 2, pages 73 –78, july 2009.

[9] T. Cucinotta, F. Checconi, G. Kousiouris, D. Kyriazis, T. Varvarigou, A. Mazzetti, Z. Zlatev, J. Papay, M. Boniface, S. Berger, D. Lamp, T. Voith, and M. Stein. Virtualised e-learning with real-time guarantees on the irmos platform. In *Service-Oriented Computing and Applications (SOCA), 2010 IEEE International Conference on*, pages 1 –8, dec. 2010.

[10] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[11] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 26 – 35, 2002.

[12] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig. ARM TrustZone as a Virtualization Technique in Embedded Systems. In *Proceedings of Twelfth Real-Time Linux Workshop*, Nairobi, Kenya, October 2010.

[13] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*.

[14] G. Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

[15] G. Heiser. The Motorola Evoke QA4, A Case Study in Mobile Virtualization. *OK-Labs Technology White Paper*, 2009.

[16] G. Heiser and B. Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM.

[17] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Chapter 24*, December 2011.

[18] R. Kaiser and S. Wagner. Evolution of the PikeOS Microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, pages 50–57, Sydney, Australia, January 2007.

[19] T. Kerstan, D. Baldin, and S. Grösbrink. Full Virtualization of Real-Time Systems by Temporal Partitioning. In S. M. Petters and P. Zijlstra, editors, *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 24–32, 6 - 9 July 2010. in conjunction with the 22nd Euromicro Intl Conference on Real-Time Systems Brussels, Belgium, July 7-9, 2010.

[20] Y. Kinebuchi, M. Sugaya, S. Oikawa, and T. Nakajima. Task grain scheduling for hypervisor-based embedded system. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 190–197, Washington, DC, USA, 2008. IEEE Computer Society.

[21] J. Kiszka. Towards Linux as a Real-Time Hypervisor. In *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, 2009.

[22] J. Kiszka. KVM in Embedded: Requirements, Experiences and Open Challenges. In *KVM Forum 2010*, August 2010.

[23] Kernel Based Virtual Machine for Linux. `http://http://www.linux-kvm.org/`.

[24] J. Lange and P. Dinda. An Introduction to the Palacios Virtual Machine Monitor—Release 1.0. Technical Report NWU-EECS-08-11, Department of Electrical Engineering and Computer Science, Northwestern University.

[25] A. Masrur, T. Pfeuffer, M. Geier, S. Drössler, and S. Chakraborty. Designing VM Schedulers for Embedded Real-Time Applications. In *Proceedings of the 11th International Conference on Embedded Software*, EMSOFT, 2011.

[26] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 124–133, Austin, Texas, USA, Dec. 2002.

[27] R. Mijat and A. Nightingale. Virtualization is Coming to a Platform Near You. *ARM White Paper*, 2010.

[28] Open Kernel Labs. `http://www.ok-labs.com/`.

[29] SYSGO PikeOS. `http://www.sysgo.com/`.

[30] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein. Analysis of Hierarchical Fixed-Priority Scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 152 –160, 2002.

[31] Seehwan Yoo and Chuck Yoo. Real-time support for Xen. `http://www.xen.org/files/xensummit_intel09/Real-timesupportforXen.pdf`.

[32] J. Son and J. Alves-Foss. Covert Timing Channel Analysis of Rate Monotonic Real-Time Scheduling Algorithm in MLS Systems. In *7th Annual IEEE Information Assurance Workshop*, West Point, NY, USA, June 2006.

[33] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 209–222, New York, NY, USA, 2010. ACM.

[34] Oracle Virtualbox. `http://www.virtualbox.org/`.

[35] VirtualLogix. `http://www.virtuallogix.com`.

[36] VMware vSphere Hypervisor™ (ESXi). `http://www.vmware.com/products/vsphere-hypervisor/overview.html`.

[37] Xen ARM Project. `http://wiki.xensource.com/xenwiki/XenARM`.

[38] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards Real-time Hierarchical Scheduling in Xen. In *Proceedings of the 11th International Conference on Embedded Software*, EMSOFT, Oct. 2011.

[39] F. Zhang and A. Burns. Analysis of Hierarchical EDF Pre-emptive Scheduling. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 423 –434, Dec. 2007.

[40] B. Zuo, K. Chen, A. Liang, H. Guan, J. Zhang, R. Ma, and H. Yang. Performance Tuning Towards a KVM-Based Low Latency Virtualization System. In *Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on*, pages 1 –4, Dec. 2010.