# Exploiting Sporadic Servers to provide Budget Scheduling for ARINC653 based Real-Time Virtualization Environments

Matthias Beckert, Kai Björn Gemlau and Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig, Germany
{beckert | gemlau | ernst}@ida.ing.tu-bs.de

***Abstract*—Virtualization techniques for embedded real-time systems typically employ TDMA scheduling to achieve temporal isolation among different virtualized partitions. Due to the fixed TDMA schedule, worst case response times for IRQs and tasks are significantly increased. Recent publications introduced slack based IRQ shaping to mitigate this problem. While providing better response times for IRQs, those mechanisms neither improve task timings nor provide a work conserving scheduling. In order to provide such capabilities while still providing temporal isolation, we introduce a method based on the well known sporadic server model. In combination with a proposed budget scheduler the system is able to schedule a TDMA based configuration while providing better response times and the same amount of temporal isolation. We show correctness of the approach and evaluate it in a hypervisor implementation.**

## I. Introduction

Virtualization is a well known topic within the field of general purpose computing. Often one physical machine hosts 20 virtual ones or even more. Since hardware within the embedded domain gets more and more powerful, techniques like virtualization gain attention. Modern cars include more than 50 embedded electronic control units (ECUs). These ECUs host the software for more or less each function inside a car. The ECUs are connected among themselves with different interconnects like CAN and/or FlexRay. A virtualization of different ECUs on a single hardware would reduce the overall number of ECUs. It would therefore relax load on interconnects.

Though combining applications of different ECUs on a more powerful (multi-core) hardware is not straight forward. Safety standards like the IEC61508 [1] or the automotive ISO26262 [2] require a *sufficient independence* of different applications running on the same hardware. One possible method to provide such sufficient independence is a certified run-time environment (RTE) that provides a temporal and spatial isolation. Such an RTE based on a virtualizing micro kernel is often called hypervisor. The usual approach is often a partitioned RTE with one instance per core. Commercial virtualization environments such as PikeOS [3] or OKL4 [4], are already available hypervisor implementations and provide real-time capabilities to virtualized applications. Such systems are already present and used within the avionic industry. With the IMA architecture of the ARINC653 standard [5] virtualization techniques have become part of a standardized software architecture in safety-critical systems.

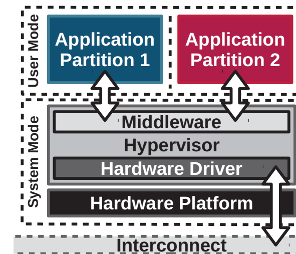



Fig. 1. Hypervisor Architecture

Fig. I shows a simple setup of a hypervisor based system with two virtualized application partitions. Both application partitions are executed within the processor's user mode, isolated via a memory management or protection unit (MMU/MPU). Only the hypervisor is executed within the processor's system mode. Communication between applications and access to shared peripherals is handled by a middleware as part of the hypervisor. Temporal isolation is usually achieved in ARINC653-like systems with a simple time-division-multiple-access (TDMA) based time-partitioning. A problem arises when considering interrupt requests (IRQ) in TDMA scheduled systems. The authors in [6] already showed that the response times for IRQs in such systems are dominated by the TDMA schedule. As IRQs for user level applications are only partly handled by the hypervisor, most of the IRQ related work must be done within a partition context. This means that a part of the IRQ is delayed until the corresponding partition is scheduled by the TDMA scheduler. While providing a perfect temporal isolation, the standard TDMA scheduling is often only suitable for systems with a periodic task activation pattern, handling sporadic events instead is challenging. Serving IRQs outside of the TDMA scheduling would therefore violate the temporal isolation. In order to service those sporadic events better, we will present within this paper a method to map ARINC653 based systems to a sporadic server mechanism[7].

The authors of [6] showed the influence of the TDMA scheduling on IRQ response times. They also proposed a monitoring based modification to the TDMA scheduling called *IRQ shaping*, which utilizes unused slack and processes IRQs outside a regular TDMA schedule. In order to provide a sufficient degree of temporal independence a monitor from [8] was used to bound the slack usage. The paper showed a significant improvement compared to standard TDMA scheduling.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

It introduced a method how available slack could be used, but without considering the design of such systems. This was faced in a followup paper [9], where a mechanism was proposed to generate an optimal system configuration for such systems based on the internal parameters of the ARINC653-like system.

The sporadic server (SPS) was first proposed in [7] and has become a part of the IEEE Portable Operating System Interface (POSIX)[10] as a scheduling policy. Also, there have been suggestions to implement this policy into the Linux kernel[11] to provide additional real-time capabilities and temporal isolation. The first SPS specification in the POSIX standard was not entirely correct, causing *budget amplification* and a *unreliable temporal isolation*. This was addressed and solved in [12]. The usage of an SPS for virtualization environments was already used in [13], [14]. The proposed kernel Quest V uses so called VCPUs which are scheduled on different physical CPUs. Quest V differentiates between two types of VCPUs, called *Main VCPUs* for applications and *I/O VCPU* for IRQ based events. In contrast to our work the SPS was only considered for *Main VCPUs* and not used for the IRQ based workload of *I/O VCPUs*. Also the authors in [15] used a standard priority based sporadic server to provide soft real-time guarantees. Nevertheless, the proposed method was not designed to cover TDMA based systems like ARING653.

The mechanisms in [6], [9] showed, how IRQ processing in ARINC653-like system could be improved while maintaining a sufficient degree of independence. Nevertheless, the proposed method is not optimal as the underlying monitoring from [8] scales linear in memory usage and runtime with the processed number of IRQs during the available slack. Runtime and memory overhead therefore depend on the chosen system configuration. Also the TDMA like scheduling is still not work conserving, as it doesn't consider idle times within partitions. We propose in this paper a method to map the existing configurations from [9] to an SPS based budget scheduling. We will show how to provide the same level of sufficient independence as the previous TDMA system with IRQ shaping, while providing a work conserving scheduling with improved response times for IRQs and tasks. We show correctness of the approach and evaluate it with a modified implementation of the $\mu$C/OS hypervisor ($\mu$C/OS-MMU) [16].

## II. Hierarchical Scheduling in ARINC653

In this section we will explain the hierarchical scheduling used in ARINC653 and introduce our system model. As already mentioned, real-time hypervisors often use TDMA scheduling to enforce temporal isolation. For the hypervisor a partition is (nothing more than) a TDMA scheduled task. Those partitions form the task-set $\Gamma_{HYP}$ of the hypervisors TDMA scheduler. Each partition consists of a set of tasks that form the isolated application.

If we consider a setup with $P$ partitions, each partition $p$ obtains an execution time of $T_p$ time units, called time partition. The hypervisor cycles through all $P$ time partitions and repeats this every $T_{TDMA}$ time units. Therefore we get:

$$T_{TDMA} = \sum_{j \in \Gamma_{HYP}} T_j \tag{1}$$

Each partition $p$ consists of a task-set $\Gamma_p$. As specified in the general purpose APEX interface of the ARINC653 stan-

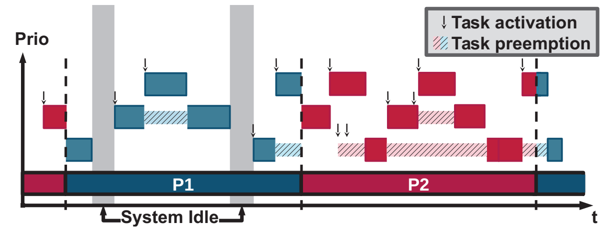



Fig. 2. Simple hierarchical scheduling with TDMA and SPP

dard [5], those task-sets are scheduled with a static priority preemptive (SPP) policy.

An example for such a scheduling with two partitions $P1$ and $P2$ is shown in Fig. 2. Inside the partitions the SPP scheduling is used to increase the overall utilization of the system. Nevertheless, this hierarchical scheduling technique is still not work conserving. As shown in Fig. 2, there are two points in time during $P1$ where the system is idle although $P2$ has a pending not yet completely served task activation. The response time analysis of such a system is explained in great detail in [9], as it forms the basis for the time partition optimization. Therefore we won't explain it further in this paper.

### A. Temporal Isolation

The main reason why TDMA is so frequently used in real-time virtualization environments is the provided temporal isolation. An exemplary TDMA schedule with four time partitions is shown in Fig. 3. The scheduling is fixed and repeated each $T_{TDMA}$ time units. When considering the interference $\widehat{I}_p$ a partition $p$ can see in any time window of size $\Delta t = T_{TDMA}$ this is given as:

$$\widehat{I}_p(\Delta t = T_{TDMA}) = T_{TDMA} - T_p = \sum_{j \in \{\Gamma_{HYP} \setminus p\}} T_j \tag{2}$$

This also means, that the longest delay until the partition is scheduled after an incoming event is equal to $\widehat{I}_p(\Delta t = T_{TDMA})$. The received service $\widehat{\beta}_p$ of $p$ in any $\Delta t = T_{TDMA}$ is also simply given as:

$$\widehat{\beta}_p(\Delta t = T_{TDMA}) = T_p \tag{3}$$

For a new scheduler which should provide a temporal isolation, (2) and (3) are used as bounds. Based on this, $I_p$ and $\beta_p$ are given as

$$I_p(\Delta t = T_{TDMA}) \leq \widehat{I}_p(\Delta t = T_{TDMA}) \tag{4}$$

$$\beta_p(\Delta t = T_{TDMA}) \geq \widehat{\beta}_p(\Delta t = T_{TDMA}) \tag{5}$$

As long as this can be achieved for each $\Delta t = T_{TDMA}$ and for each partition during operation, the scheduler provides the same temporal isolation as a TDMA scheduled system.

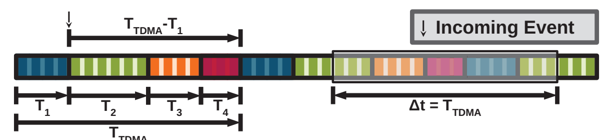



Fig. 3. Temporal isolation in any time window of size $T_{TDMA}$

### B. IRQ Processing and Slack based IRQ Shaping

The IRQ processing in virtualized systems has already been discussed in [6], [9] and will only be outlined in this paper. In general the IRQ processing is distributed into two parts. On occurrence of an IRQ the processor switches to the privileged mode and executes trusted code from the hypervisors context. This code is called *top handler* (TH) and does some basic IRQ handling stuff, like resetting IRQ bits or releasing hardware resources. When the TH has finished execution, the hypervisor redirects the IRQ to the corresponding partition where the application specific *bottom handler* (BH) is executed. The problem lays in the fact, that the BH is scheduled according to the TDMA scheme. Therefore the BH may suffer the worst case interference from the TDMA scheduling.

In order to mitigate this problem, the slack based IRQ shaping was introduced. The IRQ shaping is based on the assumption, that each time partition is larger than needed and therefore includes some slack. The authors implemented a monitoring mechanism from [8] to ensure that only available slack was used to process a BH in a foreign time partition. The system model of [9] defined, that each time partition $T_p$ consist of two parts. A part $T_{p,min}$, necessary each $T_{TDMA}$ time units in order to meet all deadlines and another part $T_{p,S}$, which can be shared to process BH from other partitions.

$$T_p = T_{p,min} + T_{p,S} \tag{6}$$

If we insert this into (1), we get

$$T_{TDMA} = \sum_{p=1}^{P} T_{p,min} + \sum_{p=1}^{P} T_{p,S} \tag{7}$$

where $\sum_{p=1}^{P} T_{p,S}$ denotes the overall slack of the TDMA scheduling. The algorithm from [9] was designed to search for a valid TDMA configuration, where each partition reaches all deadlines and the overall slack is maximized. In the context of Fig. 2 this means, the algorithm designs a configuration where the system idle times within the TDMA schedule are maximized. Within the remainder of this paper we will exploit those idle times in order to provide a work conserving scheduler for such systems.

## III. Sporadic Server

The original SPS algorithm was developed by Sprunt et al. and presented in [7]. The general idea was to provide a performant scheduling mechanism with temporal isolation for aperiodic (sporadic) task activation. For each aperiodic task, the SPS defines an execution budget and a replenishment time. In contrast to other mechanisms like the deferrable server mentioned in [7] or the priority exchange server, the SPS does only replenish an execution budget if it was used before.

An example with two tasks is shown in Fig. 4. Each task $\tau_i$ has an execution time budget $B_i$, which at the beginning is initialized to its maximum value $B_{i,max}$. When a task $\tau_i$ is executed, the corresponding budget $B_i$ is decreased. Therefore a downwards slope in Fig. 4 indicates a period of time, where the corresponding task is executed. Due to the specification of the SPS algorithm from [7], each task $\tau_i$ has a replenishment period $T_{R,i}$, which is always relative to the latest start of execution. In Fig. 4 $\tau_1$ starts execution at $t_0$ and stops at $t_1$. During this time a budget of $t_1 - t_0$ time units was consumed. For the sake of simplicity, both tasks have

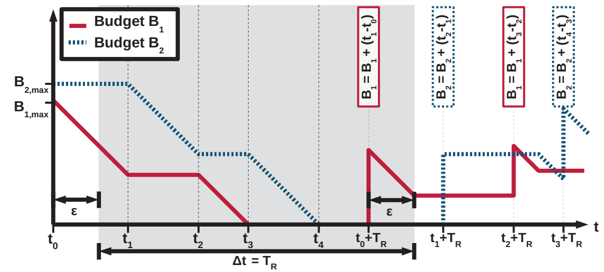



Fig. 4. SPS Budget

the same replenishment time $T_{R,1} = T_{R,2} = T_R$. Therefore an amount of $t_1 - t_0$ time units is replenished at $t_0 + T_R$. The same is done for $\tau_2$ when it stops execution at $t_2$ and so on. [7] also defined priorities for sporadic server tasks. Those priorities define the order in which the tasks are scheduled. In our example from Fig. 4, $\tau_1$ would have a higher priority as it is scheduled first. The later defined POSIX standard states that each sporadic server task would have two priorities. One foreground priority, used as long as the task has remaining budget and a lower background priority, used when the budget is depleted. However, [12] already showed the pitfalls when it comes to the implementation which is why we won't use this solution. Therefore when the budget depletes at $t_3$, $\tau_1$ is preempted until the budget is replenished at $t_0 + T_R$.

The advantage of the SPS is its ability to provide the interference bound as required in (4). If we consider a time window $\Delta t = T_R$ and move it across the time line, any task $\tau_i$ can never execute more than $B_{i,max}$ time units during this time window. The worst-case behavior of a task $\tau_i$ is given, when $\tau_i$ uses its budget as soon as it is available. Let us consider $\tau_1$ which depletes its budget $B_1$ within the first $T_R$ time units ($t_0 \to t_0 + T_R$). Within this time window the caused interference by $\tau_1$ is equal to its maximum value $B_{1,max}$. Right at $t_0 + T_R$ the budget of $\tau_1$ is replenished and $\tau_1$ starts executing again. If we shift the considered time window by $\epsilon$ time units to the right, the budget used by $\tau_1$ inside the time window is still $B_{1,max}$ and can never be greater than that.

Therefore based on the SPS algorithm a task $\tau_i$, as part of the SPSs task-set $\Gamma_{SPS}$, can never use more budget than defined in $B_{i,max}$, in any considered time window of size $T_{R,i}$. If all SPS tasks use the same replenishment period $T_R$, a single SPS task will never see more interference than the summarized budgets of all other tasks in $\Gamma_{SPS}$ within a time window $\Delta t = T_R$. This leads us to:

$$I_{SPS,i}(\Delta t = T_R) = \sum_{j \in \{\Gamma_{SPS} \setminus i\}} B_{j,max} \tag{8}$$

Comparing (2),(4) and (8) show, that for a hypervisor's TDMA based partition level task-set $\Gamma_{HYP}$ the sporadic server satisfies the interference bound, if the time partitions sizes $T_j$ are interpreted as maximal budget values $B_{j,max}$ and the replenishment period is set to $T_R = T_{TDMA}$.

## IV. SPS Architecture and Budget Scheduler

As described above, the SPS is able to provide the same interference bound as a TDMA scheduler. In order to also satisfy (5), we modified the known model of the sporadic server and extracted the scheduler as an own component. Instead of a budget state based priority switching (as defined

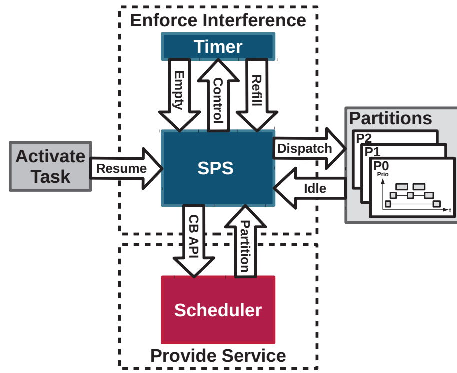



Fig. 5. SPS and scheduler system architecture

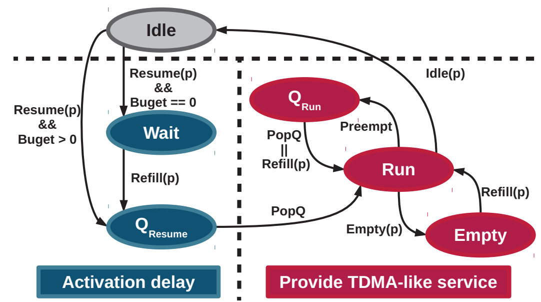



Fig. 6. Partition-state graph for budget scheduler

in POSIX), we present a model where an arbitrary scheduler with a defined callback interface (CB API) can be used. This model is shown in Fig. 5. In general the model consists of two parts. A first one to enforce the interference (eq. (4)) and a second one responsible for the provided service (eq. (5)). As explained in the previous section, the SPS is able to provide an interference bound. The main duty of the SPS in order to provide this bound is the control of a timer module. The timer provides two different signals, one for budget depletion (*Empty*) and another one for budget replenishment (*Refill*). Both signals can be provided by a single hardware timer, either with additional software or with multiple capture compare units in hardware. The *Empty* signal belongs always to the current scheduled partition. Based on the equal replenishment time for each partition, the *Refill* signal is based on a simple first-in-first-out (FIFO) buffer. Another important thing is, while being in charge for dispatching (*Dispatch*) partitions, the SPS does not decide which partition should be dispatched, as this is done by the scheduler instead. Also the SPS keeps track, if a partition would like to suspends its execution (*Idle*). This happens when all tasks inside a partition served their current activation and do not have any pending activations. An idle partition can be reactivated (*Resume*) by different events like time ticks, partition-2-partition communication or any other type of subscribed hardware IRQs.

We already mentioned, that the SPS itself doesn't decide which partition should be dispatched next. Instead, this decision is taken by the scheduler based on the CB API including different scheduling related events. The most important events for scheduling decision are:

$Empty(p)$: Partition $p$ depleted its budget
$Refill(p)$: Budget for partition $p$ was replenished
$Idle(p)$: Partition $p$ is idle
$Resume(p)$: Partition $p$ was reactivated

In addition to these callbacks, the scheduler does also have access to the current budget state of each partition. Each of these callbacks return the partition which should be dispatched next. Therefore the behavior of the system and the provided service for each partition is under control of the scheduler. In order to satisfy the service bound (5) a specific scheduler implementation is needed.

Fig. 6 shows the partition-state graph of the Budget Scheduler. Such a state machine exists within the system for each scheduled partition. Only one partition at a time can be in the $Run$ state indicating that the partition is scheduled at the moment. In general, the scheduler needs to enforce two different things. First, when a partition $p$ is activated and leaves the $Idle$ state it must be ensured, that the partition won't be delayed longer than $T_{TDMA} - T_p$ until it is scheduled. The second thing is, a partition must always see the same service as for a TDMA scheduler as long as it isn't within the $Idle$ state.

In general, the system is constructed around two queues. One, holding partitions with outstanding workload and another one for partitions ready to be scheduled (e.g. their current budget is greater zero). $Q_{Resume}$ stores partitions, which just have been activated and $Q_{Run}$ stores partitions which have been preempted during execution. The order inside the queues is FIFO based and both queues are drained based on the *PopQ* command, which is shown in Fig. 6 and Table I. As only one partition at a time can be scheduled by the SPS, the queues are prioritized and $Q_{Resume}$ is only drained if $Q_{Run}$ does not hold partitions any more. If both queues are empty and the *PopQ* command is executed, the entire system is idle (*SPS.idle* in Table I). A partition can only be stored in either $Q_{Resume}$ or $Q_{Run}$, therefore the number of stored partitions in $Q_{Resume}$ and $Q_{Run}$ can't be greater than $P$ at any point in time. In general Table I shows the scheduling decisions based on callbacks and internal states. *PopQ* indicates, that the next partition is taken from the queue subsystem. $p$ indicates, that the partition, which is assigned to the callback will be scheduled next. *Current Partition* indicates, that the currently scheduled partition wouldn't be preempted independent from the passed callback parameter $p$. A general design idea of the budget scheduler is, to preempt a running partition only if it is necessary to provide the temporal isolation.

When a partition is reactivated and leaves the $Idle$ state, it is checked if the partition has some budget. If this is the case it is stored at the end of $Q_{Resume}$, otherwise this step is delayed

TABLE I
SCHEDULING DECISION BASED ON SPS CALLBACKS

| Callback | Condition | Next Partition |
|---|---|---|
| Empty($p$) | | PopQ |
| Refill($p$) | $p.in(Q_{Run}) \| \| p.in(Empty)$ | $p$ |
| | $!(p.in(Q_{Run}) \| \| p.in(Empty))$ $\&\&\ !SPS.idle$ | Current Partition |
| | $!(p.in(Q_{Run}) \| \| p.in(Empty))$ $\&\&\ SPS.idle$ | PopQ |
| Idle($p$) | | PopQ |
| Resume($p$) | $SPS.idle$ | PopQ |
| | $!SPS.idle$ | Current Partition |

$p.in(X)$: True, if $p$ is in state X
$SPS.idle$: SPS is idle, no partition scheduled at the moment
$!, \&\&, \|\|$: C-style boolean logic

with the $Wait$ state until the budget of the reactivated partition is refilled. If we assume that partition $p$ was activated at $t_0$, the interference it might see up to $t_0 + T_{TDMA}$ is upper bounded by the SPS to $T_{TDMA} - T_p$. The partitions that might cause this interference can either be stored before $p$ in $Q_{Resume}$ or $Q_{Run}$, or they are in the $Empty$ or $Run$ state. Independent of the state of other partitions, the interference is bounded by the SPS. On the other hand, if we consider, that $p$ just depleted its whole budget $T_p$ at $t_0 - \epsilon$ and also entered the idle state, the earliest point in time, when this budget is refilled is:

$$T_{Refill} = t_0 + (T_{TDMA} - \epsilon - T_p) \quad (9)$$

With $\epsilon \to 0$ we get the worst case refill time $t_0 + (T_{TDMA} - T_p)$ for $p$ between $t_0$ and $t_0 + T_{TDMA}$, which would be seen by $p$ if it was reactivated right after depleting its entire budget. Even for this case $p$ would be able to receive its entire budget of $T_p$ before $t_0 + T_{TDMA}$. Therefore the activation delay of the budget scheduler before a partition enters the $Run$ state the first time after leaving the $Idle$ state, is bounded to $(T_{TDMA} - T_p)$, which satisfies (5).

After a partition entered the $Run$ state it must be ensured, that it always receives the same amount of service than in a TDMA scheduled system as long as the partition does not enter the $Idle$ state. This means in each time window of size $\Delta t = T_{TDMA}$, $p$ must receive $T_p$ service. In order to achieve this, a simple method can be used. As long as a partition is not idle it must be scheduled immediately, when budget is refilled. Otherwise it could not be guaranteed, that the corresponding service is received within each time-window of size $\Delta t = T_{TDMA}$. This is shown in Table I as first condition of the $Refill(p)$ callback, where $p$ is scheduled if it is currently in the $Q_{Run}$ or $Empty$ state. Those states show the two different reasons, why a partition stops execution while still having work to do. A partition enters the $Empty$ state during execution, when the budget is depleted and the $Empty(p)$ callback is called. The partition starts execution again when new budget is available, based on the $Refill(p)$ callback. On the other hand, a partition enters the $Q_{Run}$ state during execution, when it is preempted by another partition. This might happen, if a $Refill(p)$ callback was executed for another partition, which was either in the $Empty$ or the $Q_{Run}$ state. In order to leave the $Q_{Run}$ state, either the queue must be drained via *PopQ* (possible conditions in Table I) or as mentioned before with $Refill(p)$. If a partition leaves the $Q_{Run}$ state based on $Refill(p)$, the corresponding entry is deleted inside the queue. Again, since the interference from other partitions is bounded by the SPS the provided service is according to TDMA, as long as a non-idle partition is scheduled whenever new budget is available. This implies, that a partition $p$ will always get its assigned budget $T_p$ as long as it has work to do, which satisfies (5).

When a partition leaves its $Run$ state based on the $Idle(p)$ callback, the next partition is loaded from the queue system via *PopQ*. Compared to the scheduler used in PikeOS, service is provided at this point to the next waiting partition in the queue system. PikeOS tries something similar, but instead of assigning processor time to waiting partitions, the time is assigned to the privileged hypervisor partition, which is not necessarily work conserving. Also, if a time partition setup does not include any slack, a partition would never enter the $Idle$ state and the budget scheduler would work like a TDMA scheduler. Because of this we use the optimization method from [9] to construct valid, TDMA-like configurations for the SPS, which include a maximum amount of slack. This way the usage of the $Idle(p)$ callback can be maximized, which should lead to shorter response times for tasks and IRQs compared to a TDMA scheduler.

TABLE II
MEMORY USAGE IN BYTES, WITH GCC -OS

| | TDMA | TDMA with Shaping | Budget |
|---|---|---|---|
| Code | 121964 | 124808 | 125548 |
| Data | 381576 | 382576 | 381776 |
| **Sum** | **503540** | **507384** | **507324** |

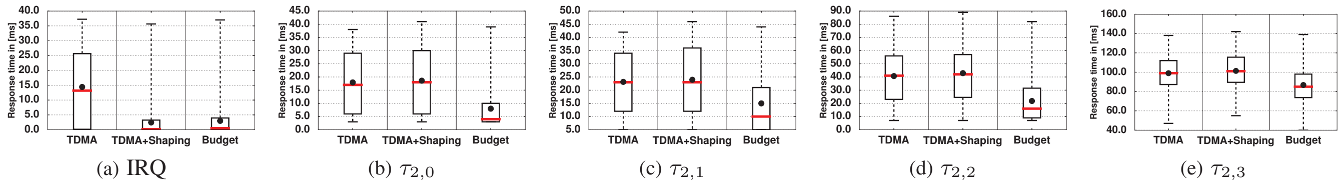
## V. EVALUATION

In order to evaluate the proposed combination of SPS and budget scheduler, we implemented it as a part of the $\mu$C/OS-MMU hypervisor [16]. $\mu$C/OS-MMU has already been used before to implement and evaluate the mechanisms from [6], [9]. Table II shows a comparison of the hypervisors memory usage for three different scheduler configurations. First the original TDMA scheduler, second the TDMA scheduler including the IRQ shaping mechanism and third the new SPS with the budget scheduler. All three configurations have been compiled with size optimization ($gcc - Os$) activated. The original TDMA scheduler is in all areas the smallest one. But keep in mind that here a commercial implementation is compared with two academic implementations, which might not have been optimized to the limit. Both modified versions instead show a similar code size and memory usage. An important thing to mention is, that the monitoring used for the IRQ shaping does not have a constant overhead for different scheduling parameters. When changing the monitors trace length (stored in the data segment) as mentioned in [8], this does effect memory usage as well as the static runtime overhead. Based on the method from [9] the trace length depends on the actual amount of slack. Therefore the overhead would increase for systems with more slack. As the SPS does not use this mechanism anymore, the memory usage and the runtime overhead are independent of the scheduler parameters.

For comparison we used the exact same task-set (Table III) from [9] for evaluation. Therefore the calculated time-partition sizes ($T_1 \ldots T_4$) are also equal to the configuration used in [9]. Again the mechanisms were evaluated on an ARM926EJ-S

TABLE III
EVALUATION TASK-SET, ALL TIMES ARE GIVEN IN [MS]

| Hypervisor ($T_1 = 2.8$) | | | | | Partition 1 ($T_2 = 11.4$) | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Prio** | **P** | **J** | **C** | **D** | **Prio** | **P** | **J** | **C** | **D** |
| 0 | 100 | 0 | 4 | 100 | 0 | 50 | 5 | 2 | 50 |
| | | | | | 1 | 100 | 5 | 4 | 100 |
| | | | | | 2 | 200 | 5 | 6 | 100 |
| | | | | | 3 | 400 | 5 | 10 | 200 |
| **Partition 2 ($T_3 = 18.0$)** | | | | | **Partition 3 ($T_4 = 16.1$)** | | | | |
| **Prio** | **P** | **J** | **C** | **D** | **Prio** | **P** | **J** | **C** | **D** |
| 0 | 50 | 5 | 3 | 50 | 0 | 100 | 5 | 4 | 75 |
| 1 | 75 | 5 | 6 | 75 | 1 | 150 | 5 | 6 | 85 |
| 2 | 150 | 5 | 7 | 150 | 2 | 200 | 5 | 8 | 150 |
| 3 | 175 | 5 | 10 | 175 | 3 | 250 | 5 | 12 | 175 |
| $T_{TDMA} = T_1 + T_2 + T_3 + T_4 = 48.3$ | | | | | | | | | |




(a) IRQ (b) $\tau_{2,0}$ (c) $\tau_{2,1}$ (d) $\tau_{2,2}$ (e) $\tau_{2,3}$

Fig. 7. Response times for timer IRQ and tasks in Partition 1

development board. As operating system on a partition level we used $\mu$C/OS-II [17] with an SPP scheduler. $\mu$C/OS-II supports up to 256 tasks, with one task per priority ($Prio$) where 0 denotes the highest possible priority inside the system. The application inside the partitions consists of a generic task-set implementation, that generates the required number of tasks with the specified parameters (Table III). Each task is characterized by a period ($P$), an activation jitter ($J$) and a worst-case execution time ($C$). One partition is assigned to the hypervisor, in order to do internal house keeping and provide a debug output. All times within this evaluation are given in milliseconds (ms), unless otherwise stated and for detailed information about the task-set we refer to [9]. In order to implement the $Idle(p)$ callback, we added a service call to the hypervisor inside the partitions idle-task. This way the service call is always invoked when a partition has nothing to do, which is exactly the behavior from Fig. 6. The shaping used in [6] and [9] was designed to improve only response times for partition level IRQs. Therefore we used, as the authors in both papers before, a timer to generate an exponential distributed IRQ pattern and measured response times in *Partition 1*. To achieve the described behavior in Sec. IV, the *Resume* callback was invoked for each partition level task activation as well as for an IRQ. The evaluation results are shown in Fig. 7. The response times for the exponential distributed IRQs and the four application tasks $\tau_{2,0} \ldots \tau_{2,3}$ have been measured during 14000 IRQs with a mean IRQ load of $30\%$ and are represented as boxplots. The boxes denote the range between $25\% \rightarrow 75\%$, the black dot indicates the average and the red bar the median of all measured values. We evaluated again the three different scheduler configurations. The results demonstrate clearly the difference between the budget scheduler and the IRQ shaping. The shaping was designed to improve only IRQ response times, which is impressively shown in Fig. 7a. But on the other hand, this does not effect other tasks of the same partition. And here you can see the big difference, the budget scheduler does not care if the outstanding workload of a partition is based on an IRQ or a task activation. It might not improve the IRQ response times as good as the IRQ shaping mechanism, but instead of this it improves the response times of all other tasks inside the partitions. Compared to the default TDMA scheduling, the IRQ response times are still significantly improved for the budget scheduler. The measurements for $\tau_{3,0} \ldots \tau_{3,3}$ and $\tau_{4,0} \ldots \tau_{4,3}$ are equivalent to those for $\tau_{2,0} \ldots \tau_{2,3}$ but omitted due to space limitations. And last but not least keep in mind, the partitions are still temporal isolated. The measurements verified this, as no deadline violation in any partition has been detected during the evaluation for all three test cases.

## VI. Conclusion and Future Work

In this paper we presented a method to replace the well known TDMA scheduling scheme in ARINC653 based real-time virtualization environments. While providing temporal isolation, TDMA might lead to long response times for IRQs as well as for tasks. The proposed budget scheduler mitigates this effect and shows improved response times in our evaluation for IRQs and tasks compared to the standard TDMA scheduling. In combination with a sporadic server, the temporal isolation can still be provided. TDMA scheduler parameters can be reused for the designed combination. It shows a significant improvement compared to TDMA, as long as the defined TDMA parameters include some slack. For a system with $100\%$ utilization, the budget scheduler performs exactly like TDMA. Our future work will include the implementation of background priorities in order to schedule partitions without budget, when the entire system would be idle otherwise.

## References

[1] International Electrotechnical Commission, "IEC61508 Ed.2 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems," 2008.

[2] International Standardization Organization, "ISO26262 - Road Vehicles. Functional Safety," 2011.

[3] R. Kaiser and S. Wagner, "Evolution of the PikeOS Microkernel," in *Proc. of 1st International Workshop on Microkernels for Embedded Systems (MIKES)*, 2007.

[4] G. Heiser and B. Leslie, "The OKL4 Microvisor: Convergence point of microkernels and hypervisors," in *Proc. of 1st asia-pacific workshop on Workshop on systems*, ACM, 2010.

[5] P. Prisaznuk, "ARINC 653 role in Integrated Modular Avionics (IMA)," in *Proc. of Digital Avionics Systems Conference (DASC)*, 2008.

[6] M. Beckert, M. Neukirchner, R. Ernst, and S. M. Petters, "Sufficient temporal independence and improved interrupt latencies in a real-time hypervisor," in *Proc. of 51st Annual Design Automation Conference (DAC)*, ACM, 2014.

[7] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, 1989.

[8] M. Neukirchner, T. Michaels, P. Axer, S. Quinton, and R. Ernst, "Monitoring arbitrary activation patterns in real-time systems," in *Proc. of 33rd Real-Time Systems Symposium (RTSS)*, IEEE, 2012.

[9] M. Beckert and R. Ernst, "Designing time partitions for real-time hypervisor with sufficient temporal independence," in *Proc. of 52st Annual Design Automation Conference (DAC)*, ACM, 2015.

[10] The IEEE and The Open Group, "Base Specifications Issue 7, IEEE Std 1003."

[11] D. Faggioli, A. Mancina, F. Checconi, and G. Lipari, "Design and implementation of a posix compliant sporadic server for the linux kernel," in *Proceedings of the 10th Real-Time Linux workshop*, 2008.

[12] M. Stanovich, T. P. Baker, A.-I. Wang, and M. G. Harbour, "Defects of the POSIX sporadic server and how to correct them," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, IEEE, 2010.

[13] Y. Li, M. Danish, and R. West, "Quest-V: A virtualized multikernel for high-confidence systems," *arXiv preprint arXiv:1112.5136*, 2011.

[14] M. Danish, Y. Li, and R. West, "Virtual-cpu scheduling in the quest operating system," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, IEEE, 2011.

[15] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: towards real-time hypervisor scheduling in xen," in *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, IEEE, 2011.

[16] Embedded Office, "$\mu$C/OS-MMU."

[17] Jean J. Labrosse, *MircoC/OS-II The Real Time Kernel*. CMP, 2002.