Response Time Analysis for Sporadic Server based Budget Scheduling in Real Time Virtualization Environments

MATTHIAS BECKERT, Institute of Computer and Network Engineering
ROLF ERNST, Institute of Computer and Network Engineering

Virtualization techniques for embedded real-time systems typically employ TDMA scheduling to achieve temporal isolation among different virtualized applications. Recent work already introduced sporadic server based solutions relying on budgets instead of a fixed TDMA schedule. While providing better average-case response times for IRQs and tasks, a formal response time analysis for the worst-case is still missing. In order to confirm the advantage of a sporadic server based budget scheduling, this paper provides a worst-case response time analysis. To improve the sporadic server based budget scheduling even more, we provide a background scheduling implementation which will also be covered by the formal analysis. We show correctness of the analysis approach and compare it against TDMA based systems. In addition to that, we provide response time measurements from a working hypervisor implementation on an ARM based development board.

CCS Concepts: • **Computer systems organization → Real-time systems**; **Real-time operating systems**; *Real-time system architecture*;

Additional Key Words and Phrases: RTOS, Virtualization, Hypervisor, Sporadic Server, Formal Analysis

1 INTRODUCTION

The size of modern embedded systems is growing rapidly. Not only the size of the software on a single unit increases in size, but also the number of units in a connected environment like cars or airplanes. The more and more complex architecture of such systems requires a step forward when considering the performance of single units. As an example, 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. In many cases those interconnects have become the systems bottle neck over time. But as the performance of modern CPU architectures for embedded systems increases, a more centralized approach relaxing interconnect loads might be a solution. With a higher computation power per unit, the overall number of ECUs might decrease. The migration from a distributed to a more centralized system is challenging. Safety standards like the IEC61508 [13] or the automotive ISO26262 [14] require a *sufficient independence* of different applications running on the same hardware.

To achieve this sufficient independence, a first idea might be a multi-core setup where each application is executed on its own core. But again even this is not straight forward, as often arbitration methods to shared resources do not provide any isolation. Another method would be

Authors' addresses: Matthias Beckert and Rolf Ernst, Institute of Computer and Network Engineering, TU Braunschweig, Hans-Sommer Str. 66, 38106 Braunschweig, Germany.

1539-9087/2017/10-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

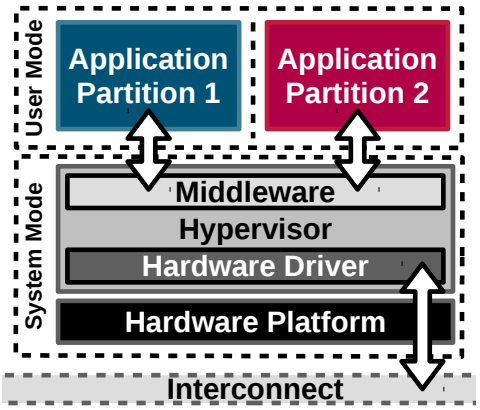


Fig. 1. Hypervisor Architecture

to combine different applications with strong data dependencies on the same core. This way it would minimize the access to shared resources which are often used for communication. But again, combining former distributed applications on the same hardware does only work with a piece of trusted hardware or software in between, which is able to guarantee a sufficient independence. A 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. Commercial virtualization environments such as PikeOS [15] or OKL4 [11], are already available hypervisor implementations and provide real-time capabilities to virtualized applications. The usual approach is a partitioned RTE with one instance per core. In addition to that a management instance on another core for configuration and runtime control is possible. Although its origin laid in the general purpose computing and server domain, virtualization based systems are already present in the real-time domains. As an example, the IMA architecture of the ARINC653 standard [19] defines usage of virtualization techniques in the avionic domain.

An example for a simple hypervisor setup with one processor core is shown in Fig. 1. Two applications are executed on the same hardware and each is isolated in a so called *Partition*. Usually the provided isolation for such a system is achieved with hardware and software support. Therefore the applications are executed within the processor's user mode, isolated via a memory management or protection unit. This way a spatial isolation is achieved. The communication between both applications is realized with a middleware controlled by the hypervisor. In order to provide a perfect temporal isolation ARINC653 uses a time-division-multiple-access (TDMA) based time-partitioning. For better resource utilization, an additional static priority preemptive (SPP) scheduling is used inside time-partitions. While providing a perfect isolation, those systems are often not suitable for event- or interrupt-driven systems. This problem has already been addressed in previous work [2, 4] in regards to TDMA-based scheduling. Nevertheless, for systems that are mainly driven by interrupts like communication stacks, TDMA is still not the perfect solution. The authors in [3], addressed this issue and realized a sporadic-server [24] (SPS) based system in order to provide more flexibility. In addition to that, the SPS enables an additional layer for background scheduling on the hypervisor level. But while providing an idea and an implementation, [3] does neither provide a response-time analysis nor implemented a background scheduling. Therefore we will cover both in this paper.

The remainder of the paper is structured as followed, we will first give an overview over the related work and then introduce our system model. This is followed by a response-time analysis for

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

SPS-based systems with and without background scheduling. Next we will give a short overview for the background scheduling implementation based on a modified implementation of the μC/OS hypervisor (μC/OS-MMU) [7]. Analysis and implementation are then evaluated and followed by the papers conclusion.

2 RELATED WORK

The issue about response-times in such TDMA-based systems was already discussed [4]. The paper used a monitoring from [18] to weaken the TDMA schedule without causing deadline misses. In general the paper tried to use idle times in time-partitions to provide better response-times for IRQs. This was then reused in [2], which proposes a time-partitions design method. Output of the proposed method are time-partition sizes, which are longer than actual needed. The included slack in this schedule was intended for IRQ handling in foreign time-partitions, bounded by the proposed monitoring from [18]. Based on the idea of larger time-partitions with additional slack, the SPS based budget scheduling [3] was designed. As the budget scheduling from [3] performs best if a larger amount of slack is available, we will later evaluate it based on generated settings from [2].

In general the SPS was first proposed in [24] and had later become a part of the IEEE Portable Operating System Interface (POSIX) [25] as a scheduling policy. The specified POSIX standard defines also the use of a priority based background scheduling. Those priorities are used whenever the budget of a task is depleted. An analysis for tasks scheduled under a SPS policy has been proposed in [1, 21]. Nevertheless, the provided analysis did neither cover background scheduling nor is it applicable for a hierarchical scheduling with SPP. There was not a lot of work considering the SPS mechanism in the last years. A major reason for this is, that the implementation of an SPS based system with several servers and different replenishment periods can be very complex. The benefit compared to other server based scheduling solutions might be moderate [5]. In contrast to this a simplified solution with only a single SPS and one replenishment time as used in [3] can be implemented with minor effort and overhead.

3 SYSTEM MODEL

We will now describe the used system model. In general we use the methods defined in the context of the compositional performance analysis (CPA) [12]. As an existing implementation we use the PyCPA [6] framework. In CPA each system consists of a set of resources, which provide service. Such resources might be a CPU, an interconnect or also an interface to a shared memory. To consume the provided service, a set of tasks is mapped to the resources. Within the context of this paper, we consider only processor cores as resources. CPA usually maps a single task-set to one resource. In order to implement an analysis for partitioned scheduling used in ARINC653 and [2–4], we slightly modified the existing CPA model. In our model a task-set Γ_p with multiple tasks is mapped to a *partition* p and a partition-set Γ_{HYP} with multiple partitions is then mapped to a core. Based on the used scheduling for partitions, there is either a TDMA slot size T_p or an execution budget $b_{p,max}$ for SPS based scheduling. In summary there is for each p:

Γ_p: Task-set
T_p: Slot length for TDMA scheduling
$b_{p,max}$: Initial execution budget for SPS scheduling

For the different scheduling approaches, two additional system parameters per core are needed. First, for the TDMA based scheduling, a cycle T_{TDMA} must be defined. This is given based on the different slot lengths as:

$$T_{TDMA} = \sum_{p \in \Gamma_{HYP}} T_p \tag{1}$$

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

Second, for the SPS based scheduling a replenishment period T_R is needed. As described in [3], this period is equal for all partitions. The initial idea in [3] was to provide a new scheduling technique based on TDMA parameters, therefore the replenishment period is equal to T_{TDMA}. Considering the initial execution budgets, those must be equal to the corresponding TDMA slot length in order to provide a TDMA-like service. This gives us:

$$T_{TDMA} = T_R = \sum_{p \in \Gamma_{HYP}} b_{p,max}$$

One might think, that a separation of TDMA based and SPS based parameters does not make sense as those are equal for systems described in [3]. But keep in mind, this is only the case if the SPS is configured to provide an TDMA-like service and interference bound. In contrast to this, it is only mandatory for the provided SPS analysis in this paper that all partitions use the same replenishment period. Therefore the analysis also works for SPS based systems whose configuration has not been derived from a TDMA based system.

To analyze a given task $\tau_{p,i}$ inside a partition p, the analysis requires several task specific parameters in order to calculate the worst-case response time $R_{p,i}$. First we need the worst-case execution times $C_{p,i}$ and its activation pattern $\eta^+_{p,i}/\delta^-_{p,i}$. The maximum arrival function $\eta^+_{p,i}(\Delta t)$ returns the maximum number of events that can arrive within any time windows of size Δt. A dual representation of the maximum arrival function is the *minimum distance function* [20] $\delta^-_{p,i}(q)$ which returns the minimum distance between the first and the $q'th$ event on any sequence of events. This way we are able to cover not only periodic activation patterns for tasks, but also more complex ones like burst or sporadic activation. Also each task defines a relative deadline $D_{p,i}$. In summary for each task $\tau_{p,i} \in \Gamma_p$ we define the following parameters:

- i: Index/Priority for partition internal scheduling
- p: Corresponding partition
- j: Index of an interfering partition
- $C_{p,i}$: Worst-case execution time (WCET)
- $R_{p,i}$: Worst-case response time (WCRT)
- $D_{p,i}$: Relative task deadline
- $\eta^+_{p,i}/\delta^-_{p,i}$: Bounds of the task activation pattern

In order to calculate worst-case response-times, PyCPA implements the concept of *busy-window analysis* [17, 22], which is well known in the embedded real-time domain. The busy-window analysis determines an upper bound on the amount of time a resource requires to service q activations of task $\tau_{p,i}$. To describe not only periodic activations, we use the *multiple-event busy-window* (or multiple-event busy time) defined by [22], based on $\eta^+_{p,i}/\delta^-_{p,i}$ as task activation pattern. To achieve a conservative response time the multiple-event busy-window is initiated with the *critical instant*, where all tasks are released simultaneously and thus create the highest load possible. The multiple-event busy-window is given through the following iterative formula, which is evaluated until convergence to a fixed-point

$$w_{p,i}(q) = q \cdot C_{p,i} + B_{p,i}(w_{p,i}(q)) \tag{2}$$

where $B_{p,i}$ is a generic term including interference from either other tasks in the same partition or entire other partitions. (2) does not contain the activation pattern directly, instead we will reveal a more detailed description for $B_{p,i}(\Delta t)$ in Sec. 4.

To determine the worst-case response time $R_{p,i}$ of $\tau_{p,i}$, the first $Q_{p,i}$ activations of $\tau_{p,i}$ have to be considered, where

$$Q_{p,i} = \max\left(n : \forall q \in \mathbb{N}^+, q \leq n : \delta^-_{p,i}(q) \leq w_{p,i}(q-1)\right) \tag{3}$$

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

The worst-case response time is then given as

$$R_{p,i} = \max_{q \in [1, Q_{p,i}]} \left(w_{p,i}(q) - \delta^{-}_{p,i}(q) \right) \tag{4}$$

4 RESPONSE TIME ANALYSIS

For the response time analysis, we will provide different definitions for the blocking term $B_{p,i}(\Delta t)$ from (4). The different blocking terms will be named according to their scheduling techniques, we therefore get:

$B^{TSPP}_{p,i}(\Delta t)$: Partition-level TDMA, Task-level SPP
$B^{SSPP}_{p,i}(\Delta t)$: Partition-level SPS, Task-level SPP
$B^{SBSPP}_{p,i}(\Delta t)$: Partition-level SPS + background scheduling, Task-level SPP

In addition to this three terms we introduce several new notations used in the corresponding response time analysis. In order to provide a better overview and improve readability, we added a notation lookup table Tab. 2 at the end of Sec. 4.3.

4.1 TDMA with SPP

First of all we start with the simplest scheduling mechanism, which is a standard TDMA-based partition scheduling with a distinct SPP scheduling inside each partition. An analysis for this scheduling was already provided in [2] and used as basis for the proposed optimization algorithm. As we use this scheduling for comparison, we will shortly introduce the used analysis. In general, $B^{TSPP}_{p,i}(\Delta t)$ consists of two parts. A first one $B^{SPP}_{p,i}$ which models the partition internal blocking time based on SPP scheduling and a second one B^{TDMA}_{p} based on the TDMA scheduling. For general SPP interference, we define

$$B^{SPP}_{p,i}(\Delta t) = \sum_{k \in hp(\tau_{p,i})} \eta^{+}_{p,k}(\Delta t) \cdot C_{p,k} \tag{5}$$

where $hp(\tau_{p,i})$ is the set of tasks inside partition p with a higher or equal priority as the considered task $\tau_{p,i}$. The TDMA based blocking $B^{TDMA}_{p}(\Delta t)$ is simply constructed based on the corresponding slot size T_p and the overall TDMA cycle length T_{TDMA}.

$$B^{TDMA}_{p}(\Delta t) = (T_{TDMA} - T_p) \cdot \lceil \Delta t / T_{TDMA} \rceil \tag{6}$$

Based on $B^{TDMA}_{p}(\Delta t)$ it is assumed that for the critical instant a task will be activated right after the time slot of the corresponding partition was finished. Therefore a task $\tau_{p,i}$ will always see at least a blocking time of $(T_{TDMA} - T_p)$ right at the beginning even if it has the highest priority inside its partition. The overall blocking time $B^{TSPP}_{p,i}(\Delta t)$ for $\tau_{p,i}$ is now given as:

$$B^{TSPP}_{p,i}(\Delta t) = B^{TDMA}_{p}(\Delta t) + B^{SPP}_{p,i}(\Delta t) \tag{7}$$

For more information about the construction of this blocking term, we refer to [2].

4.2 SPS with SPP

In order to construct a response time analysis for an SPS based budget scheduling, we will first give a short overview of the SPS budget replenishment and the scheduler's functionality. The existing implementation from [3] divides the scheduling subsystem into two parts. First the SPS, which provides service to partitions, enforces budged usage and is in charge for context switching. And second the scheduler, which performs dispatch decisions in order to determine which partition is scheduled next. The connection between both parts is formed by a set of callback functions.

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

Fig. 2 shows the budget replenishment according to the original SPS definition from [24]. In general, the execution budget of a task is only replenished if it was used before. Executing a task and therefore consuming budget is marked with a downwards slope in Fig. 2. In the given example, a task consumes its budget b_1 in between t_0 and t_1, as well as between t_2 and t_3. The point in time, where a budget is replenished depends on the start of the execution section and the replenishment period T_R. A budget of the size $t_1 - t_0$ will therefore be available again at $t_0 + T_R$. This leads to the fact, that in each window of size $\Delta t = T_R$ a task can never use more budget than its defined maximum $b_{p,max}$ (e.g. $b_{1,max}$ in Fig. 2).

Fig. 3 shows the original partition-state graph from [3] without background scheduling for the proposed budget scheduler. On a core there is such a state machine for each scheduled partition and only one partition is scheduled at a time (indicated by the state *Run*). Based on different events the SPS executes one of the scheduler's callback functions. Those callback functions do then return the partition which should be dispatched next. Each callback function might change the scheduler's internal partition-states. The most relevant callback functions are:

$Empty(p)$: Partition p depleted its budget. This event is generated by the SPS
$Refill(p)$: Budget for partition p was replenished. This event is generated by the SPS
$Idle(p)$: Partition p is idle. This event is generated by the calling partition, when there are no outstanding activations inside p
$Resume(p)$: A task inside partition p was activated, this event can be generated by either a timetick or any other kind of IRQ source

In general the proposed budget scheduler from [3] is constructed around two prioritized first-in-first-out ordered queues. A higher priority queue Q_{Run} containing partitions preempted during execution and a lower priority queue Q_{Resume} containing recently reactivated partitions with a budget greater than zero. Both queues are drained based on a *PopQ* command issued by the scheduler for several reasons. Tab. 1 shows the scheduling decisions for the above mentioned callback functions. *PopQ* takes the next partitions from one of both queues. If both queues are empty, the system is idle (*SPS.idle*). *CurrentPart* does not interrupt the current running partition and p schedules the partition next which was assigned to the callback. A partition can only be stored in one of both queues. Therefore the maximum length of a queue, and in combination with the SPS also the caused interference from all partitions stored in those queues, is limited. If a partition is stored in Q_{Run}, it can also be scheduled based on a $Refill(p)$ callback. When this happens, the entry inside Q_{Run} is removed. As the queues are based on double linked lists, this can be done with constant overhead.

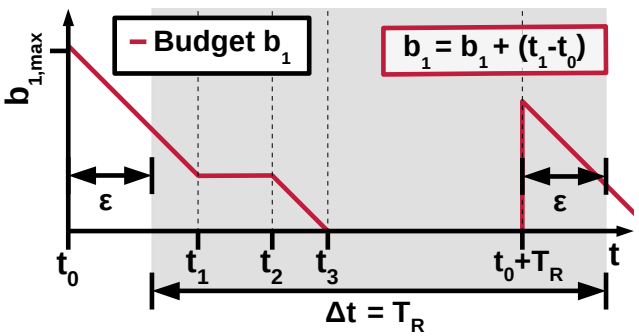


Fig. 2. Budget replenishment defined by a SPS

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

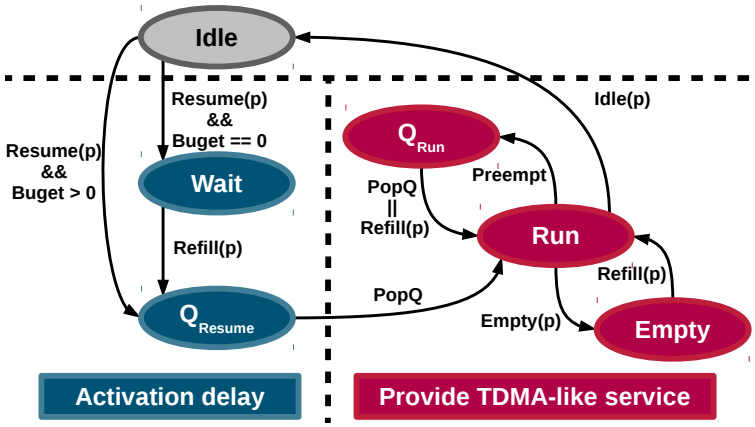


Fig. 3. Partition-state graph for budget scheduler

Table 1. Callback based scheduling decisions for SPS and SPS with background scheduling

Callback	SPS		SPS with background scheduling	
	Condition	**Next Partition**	**Condition**	**Next Partition**
Empty(p)		PopQ	$Q_{Run}.len == 0$ && $Q_{Empty}.len == 0$	PopQEmpty
			!($Q_{Run}.len == 0$ && $Q_{Empty}.len == 0$)	PopQ
Refill(p)	$p.in(Q_{Run})\|\|p.in(Empty)$	p	$p.in(Q_{Run})\|\|p.in(Q_{Empty})$ || $p.in(RunBS)$	p
	!($p.in(Q_{Run})\|\|p.in(Empty)$) && !$SPS.idle$	Current Partition	!($p.in(Q_{Run})\|\|p.in(Q_{Empty})$ || $p.in(RunBS)$)&&!$SPS.idle$	Current Partition
	!($p.in(Q_{Run})\|\|p.in(Empty)$) && $SPS.idle$	PopQ	!($p.in(Q_{Run})\|\|p.in(Q_{Empty})$ || $p.in(RunBS)$)&&$SPS.idle$	PopQ
Idle(p)		PopQ	$Q_{Run}.len == 0$ && $Q_{Empty}.len == 0$	PopQEmpty
			!($Q_{Run}.len == 0$ && $Q_{Empty}.len == 0$)	PopQ
Resume(p)	$SPS.idle$	PopQ	$SPS.idle$	PopQ
	!$SPS.idle$	Current Partition	!$SPS.idle$	Current Partition

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

Based on its design, the presented scheduler enforces two different things. First, an initial activation delay after leaving the *Idle* state and second a constant service allocation during execution. When a partition leaves its *Idle* state based on a task activation, it is either pushed directly to a queue or delayed until budget is available and then pushed afterwards. At this point the initial activation delay is limited, as the interference from other partitions within any time window of size

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

$\Delta t = T_R$ is upper bounded by the SPS to $(T_R - b_{p,max})$. According to a TDMA-like configuration as used in [3], this upper bound is also equal to $(T_{TDMA} - T_p)$. After this initial activation delay, a partition enters the *Run* state until its budget is exhausted.

Based on T_R this budget will be refilled and the partition is scheduled again. Without a background scheduling, a partition only executes if its execution budget is greater than zero. This is repeated until the partition switches back to the *Idle* state. A more detailed explanation of Fig. 3 is given in [3].

With the previous introduction we will now construct the response time analysis for the proposed budget scheduler. Fig. 4 shows the general interference a task can observe after activation during execution. Again the interference is split into two parts, one based on higher priority activation (HP) inside the same partition and second the budget provision based on the SPS. Like in TDMA this leads us to:

$$B_{p,i}^{SSPP}(\Delta t) = B_p^{SPS}(\Delta t) + B_{p,i}^{SPP}(\Delta t) \tag{8}$$

The example in Fig. 4 considers a lower priority task (LP) in partition p that is activated at t_0. The SPP based blocking is demonstrated with two HP activations of a task in the same partition at t_2 and t_6. The blocking time based on the SPS can be seen from $t_0 - t_1$ and $t_3 - t_5$. At this point the drawback of a hard SPS based budget limiting without background scheduling can be seen at t_4. Although outstanding workload does not exist in other partitions at that point in time, p is still delayed until t_5 because of a missing budget.

The critical instant for a response time analysis can be constructed with the two following constraints. First, we assume that all higher priority tasks inside the partition are activated at the same time. As a result we can reuse the SPP blocking from (5). Second, we consider the activation of the task under analysis right at the point where the entire budget of the corresponding partition has just been used in one block before. As long as we have not finished the execution of the corresponding task, we will always see $(T_R - b_{p,max})$ as interference repetitively. Therefore we get

$$B_p^{SPS}(\Delta t) = (T_R - b_{p,max}) \cdot \lceil \Delta t / T_R \rceil \tag{9}$$

as interference based on the SPS. Comparing (9) with (6) show, that the SPS mechanism introduces the same amount of interference compared to a TDMA based system. Therefore, we observe the same worst-case response time for a task in both systems, if the SPS configuration is derived from a working TDMA configuration. Nevertheless, [3] observed a better average case performance for an SPS based system, if the configuration includes some slack time. The reason for this is the self-suspend mechanism which is used if a partition enters its *Idle* state. For a configuration with

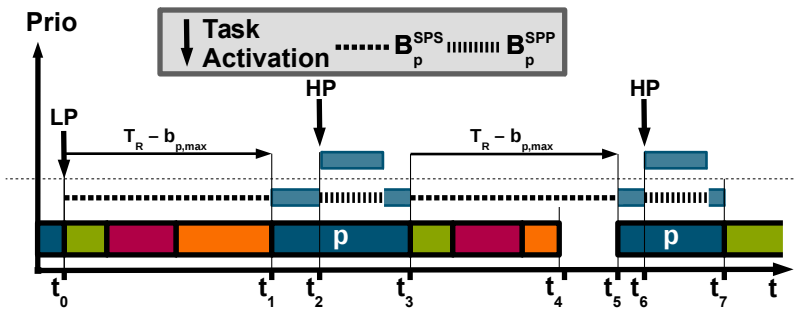


Fig. 4. SPS and SPP based interference

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

slack this results in a fragmentation of the budget usage, which leads to a lower initial blocking time in the average case.

4.3 SPS with Background Scheduling and SPP

Analysing the SPS with background scheduling starts the same way, as before. First we divide $B_{p,i}^{SBSPP}(\Delta t)$ into two parts, given as:

$$B_{p,i}^{SBSPP}(\Delta t) = B_{p}^{SPSBS}(\Delta t) + B_{p,i}^{SPP}(\Delta t) \tag{10}$$

$B_{p,i}^{SPP}(\Delta t)$ is the well-known internal SPP based blocking and $B_{p}^{SPSBS}(\Delta t)$ describes blocking based on the SPS with background scheduling. When analyzing a system with background scheduling, the interference caused by this mechanism must be considered separately. An example for this is given in Fig. 5. Like in Fig. 4, at t_0 a lower priority task inside partition p is activated. As critical instant we assume, that p just replenished its budget and all other partitions (e.g. $P1$, $P2$ & $P3$) have their entire budget left and outstanding workload. Therefore all other partitions are scheduled before p is scheduled at t_1. Again, like in Fig. 4 higher priority interference inside of p is shown at t_2 and also later at t_7. At t_3 partition p is preempted based on an empty budget. Due to $P1$'s budget replenishment (marked with T_R starting at t_0), it immediately starts execution. Same happens for $P2$, when the budget is replenished. At t_4 both, $P1$ and $P2$ don't have any outstanding workload and stop execution. Both partitions served their workload request only while having a budget available. The interference from those partitions on p based on usual budget replenishment is marked as $\tilde{B}_p^{SPS}$ in Fig. 5. $P3$ and p still have work to do at t_4 but both partitions do not have a budget as their replenishment lays in the future. At this point the standard SPS would delay both partitions until budget is available. With background scheduling the SPS still executes a partition without having budget. $P3$ is executed first and therefore causes additional interference to p, which is not included in $\tilde{B}_p^{SPS}$. This interference is marked as $\tilde{B}_p^{BS}$ and depends on how the background scheduling is implemented. A possible implementation could be a priority or queue based background scheduling. Partition p starts executing again at t_5 and finishes the considered activation at t_6. Compared to Fig. 4 the higher priority activation at t_7 would not cause any interference to the considered lower priority task. Due to the SPS budget enforcement, $\tilde{B}_p^{SPS}$ and $\tilde{B}_p^{BS}$ can in combination never cause more interference than the standard SPS interference without background scheduling from (9).

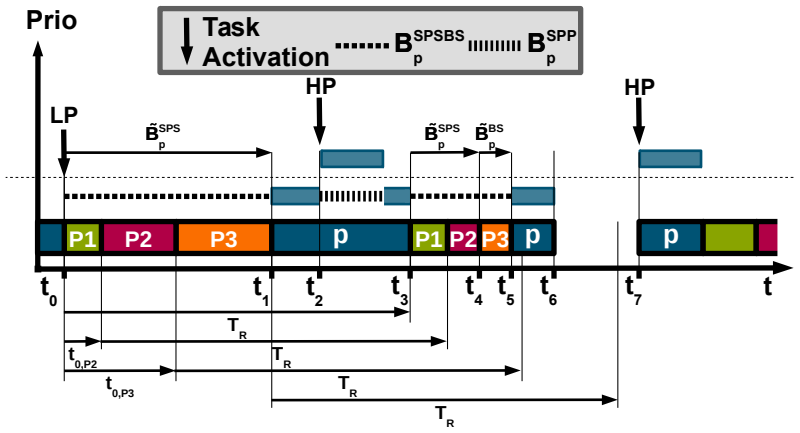


Fig. 5. SPS and SPP based interference

This leads to:

$$B_p^{SPSBS}(\Delta t) = \min\left\{\tilde{B}_p^{SPS}(\Delta t) + \tilde{B}_p^{BS}(\Delta t),\ B_p^{SPS}(\Delta t))\right\} \tag{11}$$

$\tilde{B}_p^{SPS}$ can be constructed based on the interference from all other partitions, which is limited with the SPS's replenishment mechanism. This gives us:

$$\tilde{B}_p^{SPS}(\Delta t) = \sum_{j \in \{\Gamma_{HYP} \setminus p\}} \tilde{B}_{p,j}^{SPS}(\Delta t) \tag{12}$$

In general $\tilde{B}_p^{BS}$ includes interference caused from other partitions during the background scheduling. As already mentioned this interference highly depends on the scheduling technique used for the background scheduling. For a general description we use the term $\mathcal{I}(\Gamma_{HYP} \setminus p)$ to define a set of partitions, that might interfere with partition p during background scheduling. As an example, for a priority based background scheduling a background priority is assigned to each partition. $\mathcal{I}(\Gamma_{HYP} \setminus p)$ then contains all partitions with higher or equal priority compared to p. The sum over this set gives us the interference for the background scheduling:

$$\tilde{B}_p^{BS}(\Delta t) = \sum_{j \in \mathcal{I}(\Gamma_{HYP} \setminus p)} \tilde{B}_{p,j}^{BS}(\Delta t) \tag{13}$$

Calculating the accumulated interference is straight forward, more challenging instead is the determination of $\tilde{B}_{p,j}^{SPS}(\Delta t)$ and $\tilde{B}_{p,j}^{BS}(\Delta t)$ for each interferer. This can be done based on the order of the interfering partitions, before the considered partition p is scheduled first. Within the remainder of this paper we will label the considered order as $\vec{\alpha}_{p,y}$. Fig. 5 shows this for the particular order $\vec{\alpha}_{p,y} = (P1\ P2\ P3\ p)$. Based on the order, the interfering partitions might start with an offset relative to t_0. In Fig. 5 this is shown for $P2$ and $P3$ with $t_{0,P2}$ and $t_{0,P3}$. Those offsets are important, as the budget replenishment is always relative to the budget consumption. Resulting from this, a partition j with a larger $t_{0,j}$ will be replenished later. In order to construct the budget based blocking within a time window Δt of j to the considered partition p, the minimum of requested workload and granted budget under consideration of an offset $t_{0,j}$ is needed. This leads to

$$\tilde{B}_{p,j}^{SPS}(\Delta t) = \min\left\{\beta_j(\Delta t),\ \max\left\{0, \left\lceil \frac{\Delta t - t_{0,j}}{T_R} \right\rceil \cdot b_{j,max}\right\}\right\} \tag{14}$$

with $\beta_j(\Delta t)$ representing the accumulated requested workload of all tasks in partition j (represented as taskset Γ_j) during Δt.

$$\beta_j(\Delta t) = \sum_{k \in \Gamma_j} \eta_{j,k}^{+}(\Delta t) \cdot C_{j,k} \tag{15}$$

The initial offset $t_{0,j}$ of a partition j is calculated based on the partitions which lay in front of j in the considered order $\vec{\alpha}_{p,y}$. For the offset calculation we define two look-up functions:

$f_{F,p,\vec{\alpha}_{p,y}}(j)$: Delivers position of partition j for partition order $\vec{\alpha}_{p,y}$ when analysing a task in partition p

$f_{B,p,\vec{\alpha}_{p,y}}(n)$: Delivers partition of position n for partition order $\vec{\alpha}_{p,y}$ when analysing a task in partition p

The calculation is then performed based on a fix point iteration. The first partition in the order will have an offset equal 0, the next partition offset is based on the previous offset and the requested workload of the previous partition upper bounded with its maximum budget. This way the offsets are given as:

$$t_{0,j} = \begin{cases} 0, & n = 0 \\ t_{0,m} + \max\left\{\min\left\{\beta_m(t_{0,j}),\ b_{m,max}\right\}, \epsilon\right\}, & n > 0 \end{cases} \tag{16}$$

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

$$n = f_{F,p,\vec{\alpha}_{p,y}}(j) \qquad m = f_{B,p,\vec{\alpha}_{p,y}}(n-1) \tag{17}$$

For the case $n = 1$, the previous offset would be $t_{0,m} = 0$. The fix point iteration would therefore start with $\beta_m(0)$ which returns based on (5) always 0. To enforce this the maximum in comparison to an infinitesimal small ϵ is used. This way the fix point iteration for $n = 1$ would not stop immediately.

The residual workload of a partition j that might interfere during background scheduling can easily be determined if the partial workload covered by $\tilde{B}^{SPS}_{p,j}$ is known. $\tilde{B}^{BS}_{p,j}(\Delta t)$ is then given as difference between requested workload and granted budget included in $\tilde{B}^{SPS}_{p,j}$.

$$\tilde{B}^{BS}_{p,j}(\Delta t) = \begin{cases} 0, & \Delta t \leq t_{0,j} \\ \beta_j(\Delta t) - \tilde{B}^{SPS}_{p,j}(\Delta t), & \Delta t > t_{0,j} \end{cases} \tag{18}$$

According to (14) and (18) a partition j does not generate any blocking time for $\Delta t \leq t_{0,j}$. This might be confusing at first glance, but due to the fix point iteration of the busy-window technique and the additional $q \cdot C_{p,i}$ in (2) the calculation would not stop at $\Delta t = t_{0,j}$.

We already introduced the vector $\vec{\alpha}_{p,y}$ to define the order in which the partitions get scheduled. $\vec{\alpha}_{p,y}$ is also used to perform the forward and backward look-ups $f_{F,p,\vec{\alpha}_{p,y}}$ and $f_{B,p,\vec{\alpha}_{p,y}}$. A fixed part here is that the last partition in this order is always partition p which contains the task for the considered busy window. The overall number of partitions in the system is given as the cardinality of the hypervisor task-set $|\Gamma_{HYP}|$. Therefore the number of partitions interfering with p is given as $|\Gamma_{HYP}| - 1$. Resulting from this the number of possible permutations for the order is given as $(|\Gamma_{HYP}| - 1)!$. Those permutation can be constructed with several algorithms as shown in [10, 23]. Based on the permutations we define for each partition p a $Y \times X$ matrix $\underline{A}_p$ with $X = |\Gamma_{HYP}|$ and $Y = (|\Gamma_{HYP}| - 1)!$. Each row of $\underline{A}_p$ would then contain a possible partition order, indexing those rows for $\vec{\alpha}_{p,y}$ is done via y. For the example from Fig. 5 this would result in:

$$\underline{A}_p = \begin{bmatrix} P1 & P2 & P3 & p \\ P1 & P3 & P2 & p \\ P2 & P1 & P3 & p \\ P2 & P3 & P1 & p \\ P3 & P1 & P2 & p \\ P3 & P2 & P1 & p \end{bmatrix} \qquad \vec{\alpha}_{p,1} = (P1\ P2\ P3\ p)$$

A more generic definition for $\vec{\alpha}_{p,y}$ is then given as:

$$\vec{\alpha}_{p,y} = (a_{y1}\ a_{y2}\ \ldots\ a_{y(X-1)}\ p) \tag{19}$$

Due to the design of the SPS based budget scheduling, the order in which partitions might get scheduled is in contrast to TDMA not fixed. Therefore all possible combinations included in $\underline{A}_p$ must be checked for tasks located in partition p. As a result, for each task in the system the response time analysis from Sec. 3 is performed Y-times with (10) as blocking term $B_{p,i}$. Each analysis is performed with a different $\vec{\alpha}_{p,y}$, which might result in a different offset $t_{0,j}$ and a different load distribution between $\tilde{B}^{SPS}_{p}$ and $\tilde{B}^{BS}_{p}$ for each interfering partition. In the end this might lead to different worst case response times for different offset vectors. The worst case response time is then given as:

$$R_{p,i} = \max_{y=1\ldots Y} \left\{ R_{p,i,\vec{\alpha}_{p,y}} \right\} \tag{20}$$

For systems where dependencies are known (e.g. based on action chains), impossible partition orders in $\underline{A}_p$ can be removed and don't need to be considered. Finding those dependencies in existing software is an own complex and still relevant topic. Due to the high complexity we did not address this topic directly in our analysis.

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

4.4 Simplified WCRT calculation method

The simplest way of background scheduling is an additional first-in first-out (FIFO) based queue, which contains partitions without budget but outstanding workload. An entry is read from the queue if the system would be idle otherwise. For such a system not only the implementation is straight forward, but also the analysis can be simplified. For the worst case behavior of a queue we assume that all other partitions with load for background scheduling get scheduled before the considered partition. This way $\mathcal{I}(\Gamma_{HYP} \setminus p)$ would include all other partitions from the system. At this point the partition order does not matter anymore, as we do not exclude an interfering partition in $\tilde{B}_p^{BS}(\Delta t)$. It is sufficient to take the minimum of the accumulated workload of all other partitions and the maximum SPS interference $B_p^{SPS}(\Delta t)$).

$$B_p^{SPSBS}(\Delta t) = \min\{\sum_{j \in (\Gamma_{HYP} \setminus p)} \beta_j(\Delta t),\ B_p^{SPS}(\Delta t))\} \tag{21}$$

(21) does not include any separation of replenishment based blocking and background scheduling. For a queue based background scheduling this is not needed anymore as it doesn't matter if another partition interferes during normal or background scheduling. In both cases it is assumed that our considered partition p always suffers from interference. The sum over the requested workload from

Table 2. Notation summary

Term	Eq.	Description
$B_{p,i}^{TSPP}(\Delta t)$	(7)	Blocking of task $\tau_{p,i}$ for partition-level TDMA and task-level SPP
$B_{p,i}^{SSPP}(\Delta t)$	(8)	Blocking of task $\tau_{p,i}$ for partition-level SPS, Task-level SPP
$B_{p,i}^{SBSPP}(\Delta t)$	(10)	Blocking of task $\tau_{p,i}$ for partition-level SPS + background scheduling, task-level SPP
$B_{p,i}^{SPP}(\Delta t)$	(5)	Internal SPP based blocking in partition p, generated by $hp(\tau_{p,i})$
$B_p^{TDMA}(\Delta t)$	(6)	TDMA based blocking caused on partition p
$B_p^{SPS}(\Delta t)$	(9)	Maximum SPS based blocking caused on partition p
$B_p^{SPSBS}(\Delta t)$	(11)	SPS based blocking caused on partition p with background scheduling
$\tilde{B}_p^{SPS}(\Delta t)$	(12)	Accumulated blocking from other partitions to p during SPS enforced scheduling w/ budget
$\tilde{B}_p^{BS}(\Delta t)$	(13)	Accumulated blocking from other partitions to p during background scheduling w/o budget
$\tilde{B}_{p,j}^{SPS}(\Delta t)$	(14)	Partial blocking from j to p during SPS enforced scheduling w/ budget
$\tilde{B}_{p,j}^{BS}(\Delta t)$	(18)	Partial blocking from j to p during background scheduling w/o budget
$\beta_j(\Delta t)$	(15)	Requested workload of interfering partition j
$\underline{A}_p$	(19)	Matrix with all possible orders of preceding partitions considering p
$\vec{\alpha}_{p,y}$	(19)	Currently considered order of preceding partitions
$t_{0,j}$	(16)	Offset of partition j based on current order $\vec{\alpha}_{p,y}$

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

all other partitions indirectly includes the background scheduling. As the overall interference is only bounded by the maximum SPS based interference $B_p^{SPS}(\Delta t)$, a single partition j might request more workload than its own budget allows. Including this interference implies that the outstanding workload of partition j is then served via background scheduling. Resulting in a execution delay for the considered partition p. Without the need of testing each possible permutation for partition orders, (21) provides an efficient upper bound in regards to the analysis run time. Also, as a queue provides a conservative worst-case behavior, (21) can be used as an upper bound for other background scheduling algorithms like fixed priority.

5 EVALUATION

We will evaluate the proposed methods in different ways. First we provide a comparison of the calculated response times for different scheduling mechanisms. Second we evaluate the algorithm runtime of the proposed methods from Sec. 4. Third we provide an overview of the implemented budget scheduling with queue based background scheduling and evaluate the different implementations in a working hypervisor implementation.

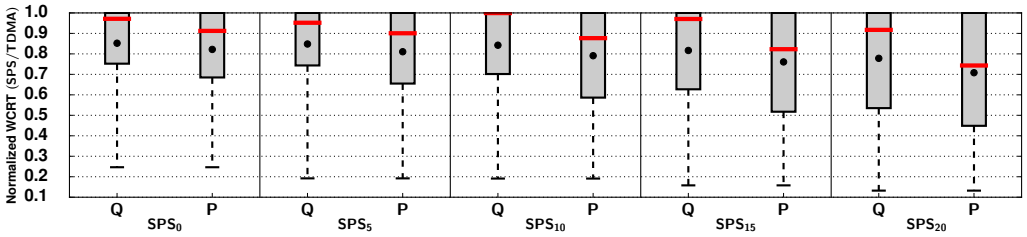
5.1 Response Time

In order to evaluate the response time we used a set of synthetic task-sets and compared the calculated response times to a TDMA+SPP based scheduling. The evaluation is based on the PyCPA framework and therefore mainly written in Python. For task-set generation we used a script from Paul Emberson[8], which is based on Roger Staffords randfixedsum algorithm and conveniently available as a python implementation. The provided script generates a set of periodic tasks for a given number, with randomly distributed periods over a defined range. In order to achieve a more arbitrary task behavior an additional a random jitter, minimum distance and relative deadline is generated for each task. The priorities are assigned in a rate-monotonic manner, according to the task periods.

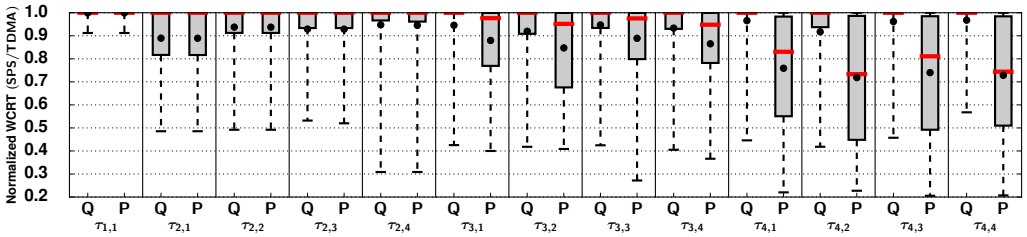
For the analysis evaluation we used two different scenarios, both with an overall number of four partitions, where one partition is reserved for the hypervisor itself. In a real implementation, this reserved partition is often used for housekeeping and can be characterized with a single task. For evaluation we fixed the utilization of the hypervisor partition to 4%. The three remaining *application partitions* got each a synthetic task-set assigned with a random utilization between $15\% \ldots 20\%$, generated by the previously mentioned script from Emberson et. al. For each task inside a partition an activation jitter was assigned randomly limited to 50% of the tasks period. Same was done for deadlines, which are defined randomly at generation between $75\% \ldots 125\%$ of the corresponding tasks period.

For the first scenario we generated 60 setups with a varying number of tasks from two to four inside the application partitions. For each setup we first run the TDMA slot optimization from [2]. The optimization provides a set of possible TDMA configurations for the hypervisor scheduling. From this set of configurations we picked all configurations with $0\%, 5\%, 10\%, 15\%$ or 20% slack. For each configuration we performed for each task in the system three response time analysis. The first analysis is based on the standard TDMA busy-window from (7). Without the usage of any background scheduling, the calculated worst-case response times for an SPS based system are equal to the TDMA values. Therefore we do not calculate them separately. Second the response times are calculated for an SPS system with queue based background scheduling, according to the conservative equation (21). Third we do the same for an SPS system with priority based background scheduling, according to the extensive method proposed in Sec. 4.3. This is repeated for all tasks in the system and all picked configurations. In the end the three analysis methods where calculated for $\sim$ 870000 configurations from 60 different task setups.

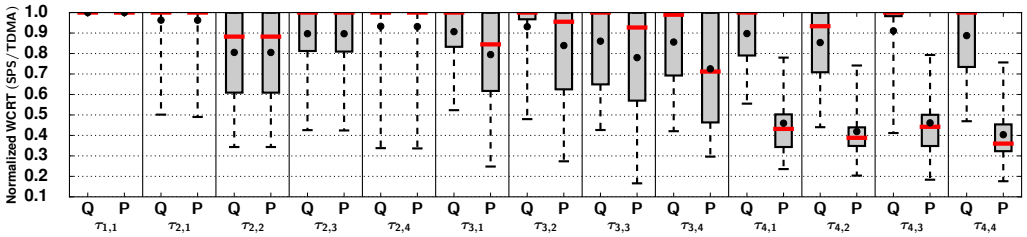
This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue



(a) Merged representation of calculated SPS based WCRTs relative to TDMA



(b) Comparison of queue and priority based background scheduling with 0% slack in the TDMA schedule



(c) Comparison of queue and priority based background scheduling with 20% slack in the TDMA schedule

Fig. 6. Response time distribution

The results for the first scenario are shown in Fig. 6a. Because of the varying number of tasks in a system setup, we tried to provide a compact representation based on box plots. The boxes denote the range between 25% → 75%, the black dot indicates the average and the red bar the median of all calculated values. In general we calculated for each task WCRT a normalized value relative to the corresponding TDMA based WCRT for the same configuration. Therefore a value equal 1.0 in this plot means, that the SPS based scheduling for the considered task and the current configuration does not provide a better WCRT compared to TDMA. Values < 1.0 do therefore represent an improvement compared to TDMA. This way we are able to compare the calculated WCRTs for all task in a single graph. As an example, the box plot Q from SPS_0 contains the WCRTs of all generated task under SPS based scheduling with queue based background scheduling, for all evaluated configurations with 0% slack in the corresponding TDMA schedule. In contrast to this, P from SPS_0 contains the WCRTs from the exact same configurations with 0% slack, but with a priority based background scheduling. The collected results show that for all calculated WCRTs both SPS based methods with background scheduling, do provide at least the same WCRT bound but often a way better value compared to TDMA. None of the calculated WCRTs for both methods,

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

were greater than the corresponding TDMA WCRT considering the same configuration. The results do also show that for some tasks and some configurations an SPS based system does not provide better WCRTs (as all box plots reach up to 1.0). This has several reasons.

First of all, the box plots include the results for all tasks of each generated setup. Therefore, also task are included which would never see any benefit, based on its task parameters (e.g. priority and period). Second, although we used also a priority based background scheduling the box plots in Fig. 6a do not show this directly. Based on the priority for background scheduling, some partitions might see better response times, some might not. Therefore, there are still tasks inside partition with a low background priority that do only see minimum improvement. This is also the reason, why the results for the queue based background scheduling show less improvement, as all partitions have the same background priority. The calculated interference is therefore always the worst possible.

In order to highlight those effects, we did a second evaluation. We used the same parameter setup with 60 task-sets as well as identical generation parameters for utilization, period, priorities, jitter and deadlines. Also, a distinct hypervisor partition was included in each task-set, leaving room for three additional application partitions. The major difference is, that we fixed the number of possible tasks for each application task-set to four. This way we are able to show the WCRT distribution for each task. Like before, we evaluated the system for TDMA configurations with $0\%, 5\%, 10\%, 15\%$ or 20% slack, which resulted in $\sim$ 990000 evaluated WCRT calculations for each task. As the results in Fig. 6b and Fig. 6c do list every task in the system individually, the influence of a priority based background scheduling can be seen. In general the tasks are described as $\tau_{p,i}$ with p indicating the partition and i a task inside this partition. The priority inside partitions is ordered in reverse, resulting in $i = 1$ as highest task priority. For the priority based background scheduling, we define partition $p = 4$ to be the highest priority and do the repeat in a descending order. This results in the lowest background priority for the hypervisor partition with $p = 1$.

The WCRT distribution is shown for all settings with 0% slack in Fig. 6b and for 20% slack in Fig. 6c. Due to the lack of space, we omit the box plots for $5\%, 10\%$ and 15%, as the results for those do fit between the measurement for 0% and 20%. The results show two different things. First, a task with a high priority inside a partition might see much lesser improvement, especially for configurations where the amount of slack is small. A high priority task inside a partition is scheduled first, when the corresponding partition is dispatched by the hypervisor. This also means, that the busy-period of such a task might not contain any background scheduling, resulting in the same WCRT compared to TDMA. In contrast to this, a lower priority task inside a partition sees more interference, leading to a longer busy-period which therefor might contain background scheduling. Second, the benefit of a priority based background scheduling can be seen for tasks inside partitions 4 and 3. Comparing a queue based background scheduling (Q) with a priority based one (P) shows a significant improvement for tasks in partitions with a higher background priority. This is especially the case for configurations with more slack (e.g. Fig. 6c). Here the tasks inside a partition ($\tau_{4,1} \ldots \tau_{4,4}$) with a high background priority achieve significant better calculated WCRTs for all tested configurations with 20% slack.

5.2 Analysis Runtime

Analyzing the runtime of fixed-point iterations is always a challenge. We will therefore give only a short overview about the runtime complexity. In general, calculating the results in Fig. 6 for $\sim$ 1860000 configurations took $\sim$ 7 hours on an Intel Xeon E5645@2.40GHz. As an execution framework we used PyCPA, in combination with the PyPy interpreter and a multi-threaded execution on 4 cores. At this point the calculation time could have been improved, as the implementations parallelism was not at its limit.

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

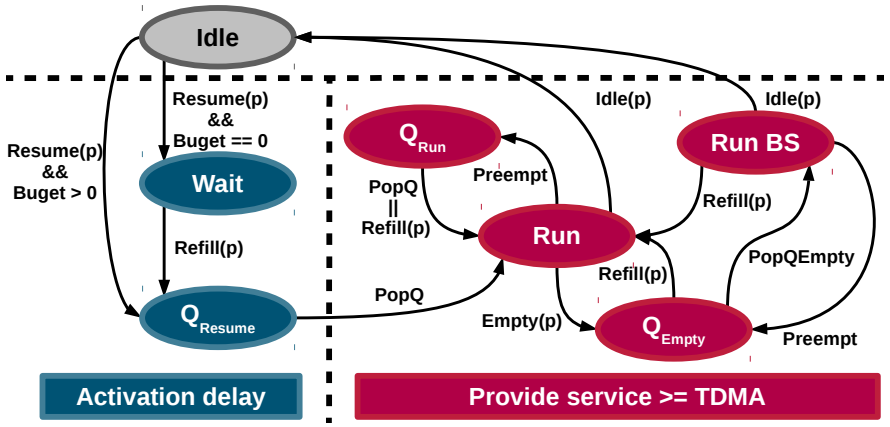


Fig. 7. Partition-state graph for budget scheduler with background scheduling

For the queue based SPS scheduling, we used (21) which results in a more or less equal runtime complexity compared to TDMA. Nevertheless, based on varying task activation patterns, different equations might lead to different busy-window sizes. Therefore the overall complexity is hard to evaluate. At least for the complex calculation method from Sec. 4.3, a comparison is possible. One might see, that based on the different orders from (19), the entire busy-window iteration is executed for each possible partition order. Therefore, the analysis runtime for such a system scales based on the number of possible partition orders Y, which is given as:

$$Y = (|\Gamma_{HYP}| - 1)! \tag{22}$$

Due to this fact, the complex calculation method takes much more time for computation. Nevertheless, only the complex analysis method for SPS based systems is able to handle different types of background scheduling. Instead, the simplified method from (21) can only handle queue based systems. It is still a design-time problem, which is not relevant during runtime. Also, calculating response times for a single configuration is comparatively fast on modern computers. What inflated up our computation time, was the sweep over the entire design space based on several TDMA configurations.

5.3 Implementation and Run Time Measurements

To test the background scheduling, we added it to an existing hypervisor setup with SPS based budget scheduling. The used hypervisor is a modified version of μC/OS-MMU [7], running on an ARM9 based LPC3250@200MHz. Inside each partition a μC/OS-II [16] with an SPP scheduling is used as operating system. The modified partition-state graph is shown in Fig. 7. Each time a partition depletes its budget and does still have outstanding workload, it is pushed to the end of a queue called Q_{Empty}. This queue is drained based on the *PopQEmpty* command as shown in Fig. 7 and Tab. 1. *PopQEmpty* is called each time, the system would be idle otherwise and switches the partition from the queue to the *RunBS* state. The partition stays in this mode, as long as it does not self-suspends itself (*Idle(p)*), is preempted by a partition with budget (*Preempt*) or receives a budget replenishment (*Refill(p)*). When budget is replenished for a partition in Q_{Empty}, the queue entry is deleted and the partition is switched back to the usual *Run* state. Comparing the conditions for partition callbacks in Tab. 1 shows that the decision which partition should be dispatched gets more complex. As an additional state (*RunBS*) and an additional queue command (*PopQEmpty*) needs to

be considered. In order to keep Tab. 1 understandable, we did not include each implementation specific detail for dispatch decisions. As an example, the implementation must consider the current timer state and the distance to the next timer IRQ. Otherwise we might dispatch a new partition while an *Refill(p)* callback is issued, which would overwrite the previously taken decision. Therefore the implementation is more challenging as it might look at first glance, when only considering Fig. 3, Fig. 7 and Tab. 1. Nevertheless, such a system can be implemented with a reasonable overhead.

For comparison we evaluated the SPS background scheduling implementation with the exact same task-set (Tab. 3) from [2]. This task-set contains three application partitions and one additional hypervisor partition for housekeeping. μC/OS-II supports up to 256 tasks, with one task per priority where 0 denotes the highest possible priority inside the system. The application partitions consists of a generic task-set implementation that generates the required number of tasks with the specified parameters (Tab. 3). Each task is characterized as a periodic task with jitter and deadline, where the jitter generation is based on a pseudo random *rand()* function. The corresponding time slots $T_1 \ldots T_4$ and T_{TDMA} from the TDMA configuration were used for the SPS based scheduling as budgets and replenishment period.

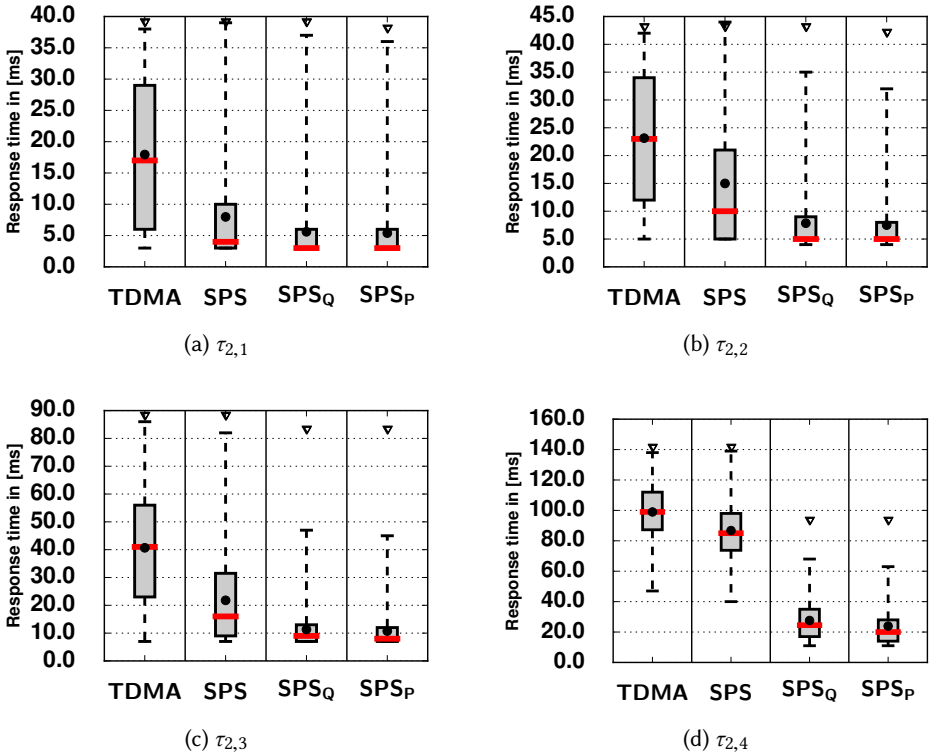
The presented implementation directly provides a queue based background scheduling. For other background scheduling policies, the queue insertion must be adapted. As an example, for a priority based background scheduling the position inside Q_{Empty} is determined by its priority. The highest priority is therefore always inserted at the head, the lowest at the tail and a medium somewhere in between. Although the runtime of such an operation might be state dependent, it is often used in real-time operating systems (e.g. ERIKA/OS[9]). This way the systems dispatch interface is the same, as always the partition at the queues head is dispatched during background scheduling. In order to provide measurements also for priority based background scheduling, we prioritized *Partition 1* over all other partitions. Adding *Partition 1* always at the head of Q_{Empty} instead of the tail is therefore sufficient for our measurements. Nevertheless, this does not resemble a full implementation for priority based background scheduling.

Fig. 8 shows the measured results for the tasks from application *Partition 1*. We collected the measurements for four different scheduling configurations. Pure TDMA, SPS without background

Table 3. 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**
1	100	0	4	100	1	50	5	2	50
					2	100	5	4	100
					3	200	5	6	100
					4	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**
1	50	5	3	50	1	100	5	4	75
2	75	5	6	75	2	150	5	6	85
3	150	5	7	150	3	200	5	8	150
4	175	5	10	175	4	250	5	12	175
$T_{TDMA} = T_1 + T_2 + T_3 + T_4 = 48.3$									

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue



(a) $\tau_{2,1}$ (b) $\tau_{2,2}$

(c) $\tau_{2,3}$ (d) $\tau_{2,4}$

Fig. 8. Response time measurements for *Partition 1*

scheduling, SPS with queue based and SPS with priority based background scheduling. The measurements were collected over a period of 2.5 minutes, due to a limited trace buffer size. The ∇ symbols inside Fig. 8 denotes the calculated WCRT for the corresponding task and scheduling configurations. The results show, that due to the SPS based budget scheduling, better response time can be achieved. Especially for tasks with a low priority inside a partition, the response times can be improved analytically when using background scheduling. This also the case for the measured average case response times. In contrast to the calculated WCRTs, the measured response times for all and not only low priority tasks show significant improvements when using the SPS based budget scheduling. During the entire evaluation on physical hardware, no deadline was missed. Comparing the code overhead for an implementation w/ and w/o background scheduling results in $\sim 0.1\%$ for an overall codesize of $\sim 120kByte$ for the hypervisor core binary (compiled with size optimization). Additional RAM usage is less than 0.1%. For both background scheduling techniques the execution time per scheduler callback was at $\sim 10\mu s$.

6 CONCLUSION

In this paper we provided a method to calculated task WCRTs for SPS based budget scheduling. While the analysis without background scheduling is straight forward, it does not provide a improvement for WCRT. Therefore we introduced an analysis methods for priority and queue based background scheduling, in order to improve WCRT estimations. For this purpose we provided two methods. A complex one enabling different types for background scheduling, and a simple one

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

only for queue based systems. The evaluation results show, that the background scheduling can lead to a massive WCRT improvements. Especially the priority based background scheduling can provide much better WCRTs, when assigning a high background priority to a single partition. This way it is also possible to schedule event or IRQ based applications inside a partition, while still preserving an upper bounded interference comparable to TDMA. The measurement results from a physical implementation on an ARM based development board validate the calculated WCRTs and show also an improved average case for response times.

REFERENCES

[1] Luis Almeida and Paulo Pedreiras. 2004. Scheduling Within Temporal Partitions: Response-time Analysis and Server Design. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04)*. ACM, New York, NY, USA, 95–103. DOI: http://dx.doi.org/10.1145/1017753.1017772

[2] Matthias Beckert and Rolf Ernst. 2015. Designing time partitions for real-time hypervisor with sufficient temporal independence. In *Proc. of 52st Annual Design Automation Conference (DAC)*. ACM.

[3] Matthias Beckert, Kai Björn Gemlau, and Rolf Ernst. 2017. Exploiting Sporadic Servers to provide Budget Scheduling for ARINC653 based Real-Time Virtualization Environments. In *Proc. of Design Automation and Test in Europe (DATE)*. Lausanne, Switzerland.

[4] Matthias Beckert, Moritz Neukirchner, Rolf Ernst, and Stefan M Petters. 2014. Sufficient Temporal Independence and Improved Interrupt Latencies in a Real-Time Hypervisor. In *Proc. of 51st Annual Design Automation Conference (DAC)*. ACM.

[5] Robert I Davis and Alan Burns. 2005. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*. IEEE, 10–pp.

[6] Jonas Diemer, Philip Axer, and Rolf Ernst. 2012. Compositional performance analysis in python with PyCPA. *Proc. of WATERS* (2012).

[7] Embedded Office. 2017. μC/OS-MMU. (2017). https://www.embedded-office.com/en/partitioning-system.html

[8] Paul Emberson, Roger Stafford, and Robert I Davis. 2010. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*. 6–11.

[9] Paolo Gai. 2002. ERIKA Enterprise, Open-source OSEK/VDX-certified RTOS. (2002). http://erika.tuxfamily.org

[10] BR Heap. 1963. Permutations by interchanges. *Comput. J.* 6, 3 (1963), 293–298.

[11] Gernot Heiser and Ben Leslie. 2010. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proc. of 1st asia-pacific workshop on Workshop on systems*. ACM.

[12] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. 2005. System Level Performance Analysis–the SymTA/S Approach. *IEEE Proceedings-Computers and Digital Techniques* 152, 2 (2005), 148–166.

[13] International Electrotechnical Commission. 2008. IEC61508 Ed.2 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems. (2008).

[14] International Standardization Organization. 2011. ISO26262 - Road Vehicles. Functional Safety. (2011).

[15] Robert Kaiser and Stephan Wagner. 2007. Evolution of the PikeOS Microkernel. In *Proc. of 1st International Workshop on Microkernels for Embedded Systems (MIKES)*.

[16] Jean J. Labrosse. 2002. *μC/OS-II The Real Time Kernel*. CMP Books.

[17] J.P. Lehoczky. 1990. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of 11th Real-Time Systems Symposium (RTSS)*. IEEE, 201–209.

[18] Moritz Neukirchner, Tobias Michaels, Philip Axer, Sophie Quinton, and Rolf Ernst. 2012. Monitoring arbitrary activation patterns in real-time systems. In *Proc. of 33rd Real-Time Systems Symposium (RTSS)*. IEEE.

[19] P.J. Prisaznuk. 2008. ARINC 653 role in Integrated Modular Avionics (IMA). In *Proc. of Digital Avionics Systems Conference (DASC)*.

[20] Kai Richter. 2004. *Compositional Scheduling Analysis Using Standard Event Models*. Ph.D. Dissertation. Technische Universität Braunschweig.

[21] Saowanee Saewong, Ragunathan Rajkumar, John P Lehoczky, and Mark H Klein. 2002. Analysis of Hierar hical Fixed-Priority Scheduling.. In *ECRTS*, Vol. 2. 173.

[22] Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. 2008. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In *Proc. of 6th International Conference on Hardware Software Codesign and System Synthesis(CODES+ISSS)*.

[23] Robert Sedgewick. 1977. Permutation generation methods. *ACM Computing Surveys (CSUR)* 9, 2 (1977), 137–164.

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue

[24] Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems* 1, 1 (1989).
[25] The IEEE and The Open Group. 2017. Base Specifications Issue 7, IEEE Std 1003. (2017). http://pubs.opengroup.org/onlinepubs/9699919799/

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue