Hierarchical Multicore-Scheduling for Virtualization of Dependent Real-Time Systems

Jan Jatzkowski, Marcio Kreutz, Achim Rettberg

Hierarchical Multicore-Scheduling for Virtualization of Dependent Real-Time Systems

Jan Jatzkowski[1], Marcio Kreutz[2], and Achim Rettberg[3]

[1] C-LAB, University of Paderborn, 33102 Paderborn, Germany,
jan.jatzkowski@c-lab.de
[2] UFRN - Department of Informatics and Applied Mathematics, Natal, Brazil
kreutz@dimap.ufrn.br
[3] Carl von Ossietzky University Oldenburg, 26129 Oldenburg, Germany,
achim.rettberg@iess.org

Abstract. Hypervisor-based virtualization is a promising technology to concurrently run various embedded real-time applications on a single multicore hardware. It provides spatial as well as temporal separation of different applications allocated to one hardware platform. In this paper, we propose a concept for hierarchical scheduling of dependent real-time software on multicore systems using hypervisor-based virualization. For this purpose, we decompose offline schedules of singlecore systems based on their release times, deadlines, and precedence constraints. Resulting schedule fragments are allocated to time partitions such that task deadlines as well as precedence constraints are met while local scheduling order of tasks is preserved. This concept, e.g., enables consolidation of various dependent singlecore applications on a multicore platform using full virtualization. Finally, we demonstrate functionality of our concept by an automotive use case from literature.

Keywords: Embedded Systems, Dependent Real-time Systems, Real-time Virtualization, Multicore Scheduling, Hierarchical Scheduling

1 Introduction

Nowadays, there is a raising interest in multicore technology for embedded real-time systems. Using multicore hardware promises not only more computational power but also reduced system size, weight, and power consumptions. However, many embedded applications require sequential interaction between different components. Increasing system performance is not reached by parallelization of dedicated software but rather by running various applications on one multicore platform concurrently [12]. Virtualization provides means to separate various applications. Multicore architectures and virtualization are therefore known as symbiotic technologies [9].

1.1 Hypervisor-based Virtualization

In this paper we focus on type-1 hypervisor-based virtualization, i.e. an additional software layer – the hypervisor – is placed between hardware and oper-

ating system (OS) respectively application software. As type-1 hypervisor run bare-metal, they must provide, e.g., device drivers either by their own (monolithic) or by means of some special guest system (console-guest). Hypervisor provide virtual machines (VM) that represent duplicates of the real hardware. These VMs allow to run various systems spatial and temporal separated on a single hardware platform. Literature distinguishes full and para-virtualization [9]. While guest systems running at full virtualization are not aware of the hypervisor, para-virtualized systems are adapted to run in VMs. Consequently, para-virtualization allows information exchange between guest system and hypervisor, but full virtualization does not.

1.2 Problem Statement

Temporal isolation is an important property of hypervisor-based virtualization for embedded real-time systems. Current real-time hypervisor ensure temporal isolation of various VMs by some cyclic scheduling on hypervisor-level (cf. Sect. 2.2). These approaches provide each VM a guaranteed share of processing time during a predefined period, but dependencies between VMs remain an open issue.

Dependencies between tasks hosted by the same VM must be solved by its local scheduler. But dependencies between VMs must be solved by hypervisor scheduler. Using para-virtualization, local schedulers could notify the hypervisor when tasks are finished. This may enable solutions based on servers to schedule VMs with precedence constraints. In contrast, full virtualization implies that local and hypervisor scheduler cannot actively exchange information. Hence, a-priori knowledge of local schedules and VM-dependencies are required to get an appropriate global scheduling.

1.3 Contribution

In this paper, we focus on hierarchical real-time scheduling of dependent VMs to enable full virtualization of singlecore systems deployed to multicore hardware. Here, dedendencies are given by precedence constraints. The challenge is to share execution time of $p > 1$ cores to $m > p$ VMs such that deadlines as well as precedence constraints are met. Each VM encapsulates a periodic real-time system driven by its separate local singlecore schedule. Time sharing shall be realized by a fixed cyclic scheduling that guarantees holding task deadlines and precedence constraints. We consider task sets with acyclic dependencies, because cyclic task dependencies imply non-deterministic behavior. Nevertheless, resulting VM dependency graph may contain cycles. To meet deadlines as well as precedence constraints of the overall system, hypervisor scheduler has to preempt execution of VMs. For this purpose, first we decompose local schedules and then allocate time partitions of various length to those parts of VM schedules. The result of our approach is an offline multicore schedule for VMs that provides not only sufficient execution time for each VM but also considers precedence constraints.

2 Related Work

In this paper, we address hierarchical scheduling of periodic tasks with precedence constraints on a multicore platform. We therefore divide related work into approaches related to multicore scheduling and hierarchical scheduling.

2.1 Multicore Scheduling

Multicore scheduling approaches are classified as partitioned or global [1]. Davis and Burns [5] state that i) most published research addresses independent tasks and ii) main advantage of partitioned multicore scheduling is reuse of results from singlecore scheduling theory after allocation of tasks to cores has been achieved. Considering periodic task sets with precedence constraints, partitioned scheduling allows to apply, e.g., adapted Earliest Deadline First (EDF*) presented by Chetto et al. [2] or Deadline Monotonic (DM) based scheduling proposed by Forget et al. [6]. Both approaches adapt deadlines to solve dependencies between tasks allocated to a singlecore and thus enable deadline-based scheduling as for independent task sets. But dependencies between tasks allocated to different cores are not considered. For global multicore scheduling, e.g., some scheduling policies from singlecore scheduling were adapted. For independent tasks, global EDF schedules p tasks with earliest absolute deadline at each time, where p is number of cores. Lee [11] extended global EDF to Earliest Deadline Zero Laxity (EDZL) that was proven to dominate global EDF [1]. Cho et al. [3] presented Largest Local Remaining Execution time First (LLREF). It is an optimal offline real-time scheduling approach for independent periodic tasks with implicit deadlines ($d = T$) and it performs non-work-conserving scheduling, i.e. cores can be idle even in case of ready tasks. Rönngren and Shirazi [15] proposed static scheduling of periodic tasks with precedence constraints for multiprocessor systems connected by a time division multiple access (TDMA) bus network. They adapt task deadlines – similar to [2], [6] – and apply a heuristic that schedules tasks w.r.t. earliest starting time, laxity, etc. In contrast to these approaches, our work aims at global offline scheduling that does not adapt local schedules, i.e. task parameters as well as local execution order keep untouched.

2.2 Hierarchical Scheduling

Most approaches for hierarchical scheduling at virtualization focus on independent sub-systems, while our work allows dependencies between those systems. In [7], Grösbrink and Almeida present hierarchical scheduling for hypervisor-based real-time virtualization of mixed-criticality systems. They address independent periodic VMs and apply partitioned hierarchical scheduling, i.e. VMs are allocated as periodic servers to cores and each core schedules its servers according to Rate Monotonic (RM). Masmano et al. [13] present the monolithic hypervisor XtratuM that provides para-virtualization. It schedules VMs – called partitions – globally by a static cyclic schedule and locally by a preemptive fixed priority-based policy [4]. Xi et al. [16] present the console-guest hypervisor RT-Xen. It

enables scheduling VMs as periodic or deferrable servers by EDF or DM priority schemes. Masrur et al. [14] proposed the priority-based scheduling plus simple EDF (PSEDF) to apply XEN hypervisor for mixed-criticality systems in automotive domain. But in contrast to our work, none of these approaches allows precedence constraints between VMs.

3 System Model

This paper focuses on hierarchical scheduling of periodic dependent real-time systems on a multicore platform. Usually, periodic embedded real-time systems get input from some sensors and compute output to control some acutators. But resources are limited to get input respectively set output via direct I/O access or network interfaces. To take this into account, we consider a periodic task model that allows asynchronous release of tasks:

$$\Gamma = \{\tau_i = (C_i, T_i, D_i, O_i) \mid 1 \leq i \leq n\}. \tag{1}$$

Each task $\tau_i \in \Gamma$ is characterized by its worst case execution time (WCET) C_i, period T_i, constrained deadline $D_i \leq T_i$, and offset O_i. By means of constrained deadlines and offsets, we are able to cover systems where the multicore platform is connected to a time-triggered network. We denote j^{th} instance of task τ_i by τ_{ij} and its absolute deadline by d_{ij}. Task dependencies are given by precedence constraints $\tau_i \prec \tau_j$ meaning that τ_i must finish before τ_j can start execution. This corresponds to implicit communication between tasks, i.e. tasks require input just when they start and provide output when finished. To keep software behavior deterministic, we assume acylic task graphs. Consequently, task dependencies can be described by directed acyclic graphs (DAG). We define the set of source respectively sink nodes as

$$source = \{\tau_i \in \Gamma \mid \nexists \tau_j \in \Gamma : \tau_j \prec \tau_i\}, \tag{2}$$

$$sink = \{\tau_i \in \Gamma \mid \nexists \tau_j \in \Gamma : \tau_i \prec \tau_j\}. \tag{3}$$

While tasks $\tau_i \in source$ make progress as soon as they are scheduled, tasks with predecessors ($\tau \in \Gamma \backslash source$) can only progress when required input has been delivered. Using hypervisor-based virtualization, task set Γ is mapped to a set of virtual machines (VM)

$$\Upsilon = \{\upsilon_k = (\gamma_k, \sigma_k) \mid 1 \leq j \leq m\} \tag{4}$$

where VM υ_k is given by a task set $\gamma_k \subset \Gamma$ and a scheduling σ_k. In general, σ_k can be an online or offline scheduling. In this paper, however, we assume offline singlecore scheduler running within VMs, i.e. σ_k represents a fix order how tasks $\tau_i \in \gamma_k$ are scheduled. We denote worst case start time of task instance τ_{ij} scheduled by σ_k with $\sigma_k^s(\tau_{ij})$ and its worst case finishing time with $\sigma_k^f(\tau_{ij})$.

The hypervisor scheduler is a fix cyclic schedule, i.e. VMs are scheduled by means of time partitions to keep temporal isolation. Each time partition

represents a time interval $I_h = [a_h, a_h + l_h[$ defined by its start time a_h and length (duration) l_h. A VM υ_k mapped to a time partition I_h will be scheduled at time a_h for l_h time units. During this time, VM υ_k can progress according to its schedule σ_k. The hypervisor schedule finally provides for each core a set of time partitions where each partition I_h is associated to a dedicated VM υ_k. We note this association by $I_h^{\upsilon_k}$.

4 Hierarchical Scheduling with Precedence Constraints

Hierarchical scheduling comprises scheduling of schedules and thus introduces different levels of scheduling. We consider hierarchical scheduling for hypervisor-based virtualization that implies two levels: Global scheduling of VMs by hypervisor and local scheduling of tasks within each VM. We restrict local schedulers to offline singlecore schedules, i.e. execution order of tasks is fix within each VM. This restriction simplifies handling a-priori knowledge of local schedules that we require to cover full virtualization.

The main idea of our approach is to combine knowledge of local schedulers' task execution order with a-priori knowledge of tasks' WCETs and dependencies to compute worst case time partitions (WCTP) for VMs. That is, we calculate worst case VM execution time required to guarantee that a dedicated task τ has finished (cf. Sect. 4.2). In Section 4.3, we schedule these time partitions, which represent activation slots of the corresponding VMs, on a multicore system. In case of success, assigning execution time to VMs according to the resulting schedule ensures that task dependencies as well as tasks' deadlines are met.

4.1 Necessary Condition for Schedulability

To our best knowledge, literature provides no schedulability test that is neccessary as well as sufficient for periodic tasks with precedence constraints on multicore systems. For multicore scheduling, there are also no approaches known that convert precedence constraints to real-time constraints – as proposed by Chetto et al. [2] for singlecore scheduling. This makes transferring results of multicore scheduling theory from independent to dependent task sets challenging. However, some results from multicore scheduling theory of independent tasks can be transferred to task sets with precedence contraints at least as necessary conditions. For instance, a trivial fact from scheduling theory is that a task set Γ with computation demand higher than computation supply provided by some hardware with p cores is not schedulable. Consequently, for multicore hardware with p identical cores, utilization of feasible task set Γ cannot be higher than available number of cores, i.e.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq p \tag{5}$$

Although Eq. 5 is just a necessary condition, it allows to exclude at least some non-feasible task sets.

4.2 Decomposition of Local Schedules

Here, we consider local schedules that result from offline singlecore scheduling. Precedence constraints of tasks which are mapped to the same VM are solved by the corresponding local scheduler:

$$\forall\ \tau_i, \tau_j \in \upsilon_k :\ \tau_i \prec \tau_j \quad \Longrightarrow \quad \sigma_k^f(\tau_{il}) \leq \sigma_k^s(\tau_{jl})\ \forall l \in \mathbb{N} \tag{6}$$

Hence, two challenges remain to be solved by hypervisor during VM scheduling: it has to schedule VMs such that (i) deadlines of tasks running within VMs are met and (ii) dependencies between tasks hosted by different VMs are taken into account. For this purpose, we decompose local schedules of VMs based on

1. deadlines of tasks that are sink nodes of dependency graphs ($\tau \in sink$)
2. release times of tasks that are source nodes of dependency graphs ($\tau \in source$)
3. dependencies between tasks that are hosted by different VMs

A first step towards enabling hypervisor to keep deadlines of tasks is done by splitting local schedules at worst case finishing time of sink nodes $\tau \in sink$. This eases handling of different periods within task set Γ. Since execution order of tasks is static within a local schedule σ, fulfilling an absolute deadline d requires to run each local schedule until all task instances with absolute deadline d are finished. Therefore, we split local schedule σ at worst case finishing time of a sink node that is scheduled by σ last amongst all other sink nodes of equal absolute deadline:

$$\max\left\{\sigma^f(\tau_{ij}) \mid \tau_i \in sink,\ d_{ij}\right\} \quad j \in \mathbb{N} \tag{7}$$

Note, that each resulting fragment of a local schedule is associated with the earliest absolute deadline d of all its tasks.

Hypervisor must also consider release time of task instances because VMs with offline schedules cannot progress as long as the currently scheduled task is not ready. To avoid that hypervisor schedules VMs that cannot progress because of unrelased tasks, we apply another decomposition step onto local schedules based on release times. We split local schedule σ at the beginning of a source node that is scheduled first amongst all other source nodes of equal release time by σ:

$$\min\left\{\sigma^s(\tau_{ij}) \mid \tau_i \in source,\ r_{ij}\right\} \quad j \in \mathbb{N} \tag{8}$$

Our last decomposition step is based on precedence constraints of tasks hosted by different VMs. As tasks with precedence constraints are just released when all predecessors have finished execution, we split local schedules based on inter-VM dependencies as follows: if a task τ allocated to VM υ_k has predecessors hosted by another VM $\upsilon_l,\ l \neq k$, we just split schedule σ_k at beginning of τ.

The result of the described decomposition is a totally ordered set Φ_k of scheduling fragments φ_h for each VM υ_k. The order within Φ_k is such that composing all scheduling fragments $\varphi_h \in \Phi_k$ w.r.t. this order results in the original local singlecore schedule σ_k. Finally, we compute worst case time partitions

(WCTP) based on these scheduling fragments and WCETs. For each local schedule fragment φ_h, we sum up WCET of task instances covered by this fragment and define a time partition I_h of this length. As this time partition is associated with the VM that hosts these task instances, we note:

$$l_h^{\upsilon_k} = \sum_{\tau_{ij} \in \varphi_h} C_i \quad \forall \varphi_h \in \Phi_k. \tag{9}$$

4.3 Multicore Scheduling of Time Partitions

Our approach for hierarchical multicore scheduling is based on time partitions I_h that were introduced in Sect. 3. While length of time partitions is set according to the WCTP resulting from decomposition of local schedules (cf. Eq. 9), starting time a_h of time partitions as well as a core must be determined by hypervisor scheduler. So, the challenge addressed by our multicore scheduling approach is to allocate time partitions $I_h^{\upsilon_k}$ to cores $\mathcal{C}_j$ and set their starting time $a_h^{\upsilon_k}$ such that all precedence constraints are met and tasks finish before their deadlines even in worst case.

We have to make scheduling decisions each time that a scheduling fragment is released or finished. As we decomposed local schedules based on precedence constraints, finishing one scheduling fragment usually implies that one or more other scheduling fragments were released during this execution. Therefore, we also make scheduling decisions when worst case finishing of a task $\tau_i \in \gamma_l$ with successor task τ_j hosted by another VM is passed. However, we just need to consider worst case finishing of the task $\tau_i \in \gamma_l$ that is scheduled by υ_l last amongst all other predecessors of τ. Keeping order of local schedules guarantees that all other predecessors hosted by VM υ_l are then finished, too.

As multicore decisions are not only based on deadlines but have to consider dependencies as well, we define two sets of scheduling fragments that are updated at each scheduling decision:

$$\mathcal{R} \subset \bigcup_{k=1}^{m} \Phi_k, \qquad \mathcal{N} \subset \bigcup_{k=1}^{m} \Phi_k \tag{10}$$

$\mathcal{R}$ covers those scheduling fragments $\varphi_h^{VM_k}$ that are ready, i.e. predecessors required to execute $\varphi_h^{VM_k}$ are finished and $\varphi_h^{VM_k}$ is due according to local schedule σ_k. In fact, $\mathcal{R}$ is similar to a ready queue known from common task scheduling. Analogous, $\mathcal{N}$ covers those scheduling fragments $\varphi_h^{VM_k}$ that are next to become ready w.r.t. order of local schedule. Both, $\mathcal{R}$ and $\mathcal{N}$, contain at most one scheduling fragment $\varphi_h^{\upsilon_k}$ of a VM υ_k. Scheduling decisions are based on the following rules with decreasing priority:

1. Schedule the fragment $\varphi_h \in \mathcal{R}$ with earliest deadline (EDF)
   Note: Here, we use deadlines associated to scheduling fragments during first decomposition step (cf. 4.2)
2. Schedule the fragment that has most successor fragments $\varphi \in \mathcal{N}$

While first scheduling rule aims at keeping deadlines, second rule addresses dependencies between different VMs.

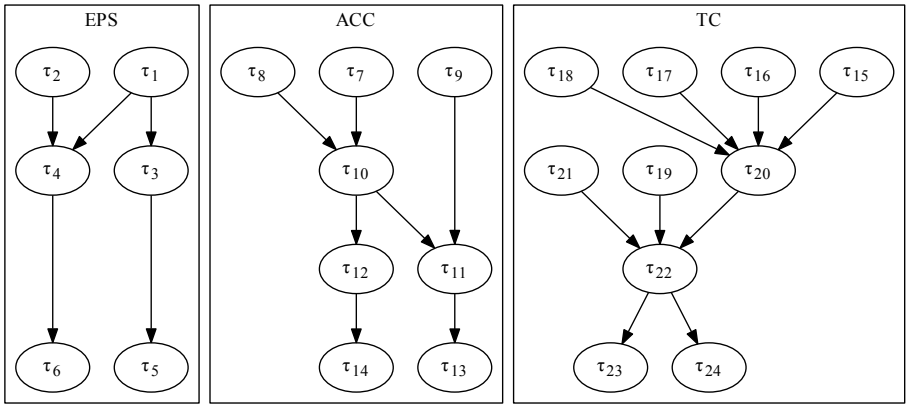

Fig. 1. Three application examples from automotive domain [8]: electric power steering (EPS), adaptive cruise control (ACC), and traction control (TC).

5 Application Example

We will use an application example to demonstrate how our approach presented in Sect. 4 works. Based on the problem definition given in Sect. 1.2, we apply our approach to a minimal system that consists of $p = 2$ cores and $m = 3$ VMs. The task set Γ deployed to VMs is taken from Kandasamy et al. [8]. It covers three applications from automotive domain: Adaptive cruise control (ACC), traction control (TC), and electric power steering (EPS). Figure 1 shows the corresponding direct acyclic task dependency graphs. In Table 1, we provide original task parameters of these applications given in [8]. In addition, we adapted WCETs of tasks by some reduction. This represents a scenario where singlecore applications are consolidated on a multicore hardware with increased computational power related to original singlecore hardware.

Table 1. Task parameters (original WCET C_i^O, adapted WCET C_i, period T_i, and relative deadline D_i) of example applications shown in Fig. 1, cf. [8].

	EPS System						ACC System							
Task τ_i	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	τ_{12}	τ_{13}	τ_{14}
C_i^O	150	175	300	250	150	100	300	150	175	300	250	200	150	200
C_i	75	90	150	125	75	50	150	75	90	150	125	100	75	100
$T_i = D_i$	1500	1500	1500	1500	1500	1500	3000	3000	3000	3000	3000	3000	3000	3000

	TC System									
Task τ_i	τ_{15}	τ_{16}	τ_{17}	τ_{18}	τ_{19}	τ_{20}	τ_{21}	τ_{22}	τ_{23}	τ_{24}
C_i^O	200	200	200	200	150	300	175	400	150	200
C_i	100	100	100	100	75	150	90	200	75	100
$T_i = D_i$	3000	3000	3000	3000	3000	3000	3000	3000	3000	3000

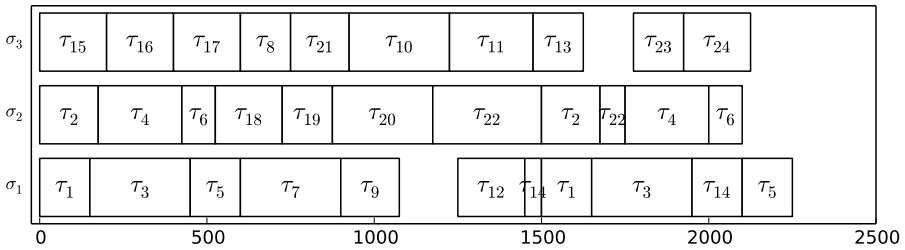

Fig. 2. Local schedules resulting from deployment of example applications to three separate singlecore schedules $\sigma_i,\ 1 \leq i \leq 3$.

Task set Γ is deployed to VMs according to an approach presented by Klobedanz et al.([10], "Algorithm 1: Initial Mapping"). This deployment originally addresses singlecore ECU-networks and thus fits to the indicated scenario of consolidating singlecore systems on a multicore platform. Figure 2 shows the resulting local offline schedules based on original WCETs. These local schedules define execution order of tasks within VMs.

5.1 Decomposition of Local Schedules

Now, we use schedule σ_2 to demonstrate decomposition of local schedules. VM υ_2 hosts tasks of two example applications: EPS and TC. Tasks of EPS have deadline $D = 1500$ while deadline of TC-tasks is $D = 3000$. Our first decomposition step – splitting based on deadlines – therefore splits schedule σ_2 after finishing of $\tau_{6,1}$ and associates first fragment with absolute deadline $d = 1500$ and second fragment with $d = 3000$.

Next decomposition step – splitting based on release times – is driven by second release of EPS system at host time $t = 1500$. According to our description in Sect. 4.2, we split σ_2 before beginning of $\tau_{2,2}$. Thus, second fragment resulting from first step is splitted again. Note, that both fragments resulting from this step keep associated with absolute deadline $d = 3000$.

Last decomposition step – splitting based on precedence constraints – requires to consider dependencies to other VMs. In particular, we split σ_2 at the beginning of tasks that require input from other VMs. In case of σ_2, this results in splits at the beginning of $\tau_{4,1}$, $\tau_{20,1}$, and $\tau_{22,1}$.

Applying these decomposition steps to the other local schedules of our application scenario results in the scheduling fragments depicted in Fig. 3. Rectangles clustering tasks correspond to the results of our decomposition steps: outmost rectangles result from deadline-based decomposition, middle rectangles from splitting based on release times, and innermost rectangles result from splitting based on dependencies.

Next, we compute length of these scheduling fragments using Eq. 9. Due to adaptation of tasks' WCETs, handling of preemptions – e.g., task $\tau_{14,1}$ in schedule $VMschedule_1$ – is challenging. Here, we just split WCET to execution

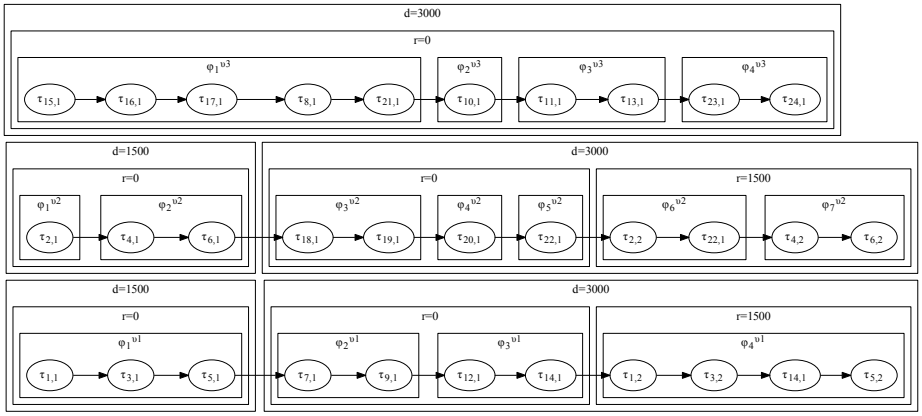

Fig. 3. Scheduling fragments resulting from decomposition of local schedules $\sigma_i,\ 1 \leq i \leq 3$.

Table 2. Worst case time partition length for local scheduling fragments.

	$\varphi_1^{v_1}$	$\varphi_2^{v_1}$	$\varphi_3^{v_1}$	$\varphi_4^{v_1}$	$\varphi_1^{v_2}$	$\varphi_2^{v_2}$	$\varphi_3^{v_2}$	$\varphi_4^{v_2}$	$\varphi_5^{v_2}$	$\varphi_6^{v_2}$	$\varphi_7^{v_2}$	$\varphi_1^{v_3}$	$\varphi_2^{v_3}$	$\varphi_3^{v_3}$	$\varphi_4^{v_3}$
$l_h^{v_k}$	300	240	125	375	90	175	175	150	160	130	175	465	150	200	175

parts of $\tau_{14,1}$ in the same proportion as it was in case of original WCET. Results are summerized in Tab. 2.

5.2 Multicore Scheduling by Time Partitions

Having local schedules decomposed into fragments, we now can allocate time partitions to dedicated cores of a multicore platform. In this example, we consider $m = 3$ VMs given by example applications introduced in this Section and $p = 2$ cores. Table 3 shows for each point in time – when the hypervisor can make scheduling decisions – why scheduling point occurs, what the current host time of hypervisor system is, which scheduling fragments are within sets $\mathcal{R}$ and $\mathcal{N}$, and which scheduling fragments are scheduled next on cores $\mathcal{C}_1$ and $\mathcal{C}_2$. For instance, applying rules defined in Sect. 4.3, hypervisor scheduling makes first decision based on deadlines of scheduling fragments. That is, $\varphi_1^{v_1}$ and $\varphi_1^{v_2}$ get higher priority than $\varphi_1^{v_3}$.

Another interesting circumstance for making scheduling decision is at line 7 where "reason for scheduling" is $\varphi_2^{v_1}$. This is the first time, $\mathcal{R}$ does not contain scheduling fragments of all VMs because $\varphi_3^{v_1} \in \mathcal{N}$ requires input from $\varphi_2^{v_3}$ that in worst case has not finished yet. Therefore, $\varphi_3^{v_1} \in \mathcal{N}$ is not passed to $\mathcal{R}$ and thus is not considered by hypervisor. Finally, mapping of scheduling fragments to cores is used to define time partitions I_h resulting, e.g., for core $\mathcal{C}_1$ in

$$I_1^{v_1} = [0, 540[\tag{11}$$

Table 3. Offline multicore scheduling of time partitions (scheduling fragments).

Reason for scheduling	Host time	$\mathcal{R}$	$\mathcal{N}$	Core $\mathcal{C}_1$	Core $\mathcal{C}_2$
	0	$\varphi_1^{v_1}, \varphi_1^{v_2}, \varphi_1^{v_3}$	$\varphi_2^{v_1}, \varphi_2^{v_2}, \varphi_2^{v_3}$	$\varphi_1^{v_1}$	$\varphi_1^{v_2}$
$\sigma_1^f(\tau_{1,1})$	75	$\varphi_1^{v_1}, \varphi_1^{v_2}, \varphi_1^{v_3}$	$\varphi_2^{v_1}, \varphi_2^{v_2}, \varphi_2^{v_3}$	$\varphi_1^{v_1}$	$\varphi_1^{v_2}$
$\varphi_1^{v_2}$	90	$\varphi_1^{v_1}, \varphi_2^{v_2}, \varphi_1^{v_3}$	$\varphi_2^{v_1}, \varphi_3^{v_2}, \varphi_2^{v_3}$	$\varphi_1^{v_1}$	$\varphi_2^{v_2}$
$\varphi_2^{v_2}$	265	$\varphi_1^{v_1}, \varphi_3^{v_2}, \varphi_1^{v_3}$	$\varphi_2^{v_1}, \varphi_4^{v_2}, \varphi_2^{v_3}$	$\varphi_1^{v_1}$	$\varphi_1^{v_3}$
$\varphi_1^{v_1}$	300	$\varphi_2^{v_1}, \varphi_3^{v_2}, \varphi_1^{v_3}$	$\varphi_3^{v_1}, \varphi_4^{v_2}, \varphi_2^{v_3}$	$\varphi_2^{v_1}$	$\varphi_1^{v_3}$
$\sigma_1^f(\tau_{7,1})$	450	$\varphi_2^{v_1}, \varphi_3^{v_2}, \varphi_1^{v_3}$	$\varphi_3^{v_1}, \varphi_4^{v_2}, \varphi_2^{v_3}$	$\varphi_2^{v_1}$	$\varphi_1^{v_3}$
$\varphi_2^{v_1}$	540	$\varphi_3^{v_2}, \varphi_1^{v_3}$	$\varphi_3^{v_1}, \varphi_4^{v_2}, \varphi_2^{v_3}$	$\varphi_3^{v_2}$	$\varphi_1^{v_3}$
$\sigma_3^f(\tau_{17,1})$	565	$\varphi_3^{v_2}, \varphi_1^{v_3}$	$\varphi_3^{v_1}, \varphi_4^{v_2}, \varphi_2^{v_3}$	$\varphi_3^{v_2}$	$\varphi_1^{v_3}$
$\varphi_3^{v_2}$	715	$\varphi_4^{v_2}, \varphi_1^{v_3}$	$\varphi_3^{v_1}, \varphi_5^{v_2}, \varphi_2^{v_3}$	$\varphi_4^{v_2}$	$\varphi_1^{v_3}$
$\varphi_1^{v_3}$	730	$\varphi_4^{v_2}, \varphi_2^{v_3}$	$\varphi_3^{v_1}, \varphi_5^{v_2}, \varphi_3^{v_3}$	$\varphi_4^{v_2}$	$\varphi_2^{v_3}$
$\varphi_4^{v_2}$	865	$\varphi_5^{v_2}, \varphi_2^{v_3}$	$\varphi_3^{v_1}, \varphi_3^{v_3}$	$\varphi_5^{v_2}$	$\varphi_2^{v_3}$
$\varphi_2^{v_3}$	880	$\varphi_3^{v_1}, \varphi_5^{v_2}, \varphi_3^{v_3}$	$\varphi_4^{v_3}$	$\varphi_5^{v_2}$	$\varphi_3^{v_3}$
$\varphi_5^{v_2}$	1025	$\varphi_3^{v_1}, \varphi_3^{v_3}$	$\varphi_4^{v_3}$	$\varphi_3^{v_1}$	$\varphi_3^{v_3}$
$\varphi_3^{v_3}$	1080	$\varphi_3^{v_1}, \varphi_4^{v_3}$		$\varphi_3^{v_1}$	$\varphi_4^{v_3}$
$\varphi_3^{v_1}$	1150	$\varphi_4^{v_3}$			$\varphi_4^{v_3}$
$\varphi_4^{v_3}$	1255				
$\sigma_1^s(\tau_{1,2})$	1500	$\varphi_4^{v_1}, \varphi_6^{v_2}$	$\varphi_7^{v_2}$	$\varphi_4^{v_1}$	$\varphi_6^{v_2}$
$\sigma_1^f(\tau_{1,2})$	1575	$\varphi_4^{v_1}, \varphi_6^{v_2}$	$\varphi_7^{v_2}$	$\varphi_4^{v_1}$	$\varphi_6^{v_2}$
$\varphi_6^{v_2}$	1630	$\varphi_4^{v_1}, \varphi_7^{v_2}$		$\varphi_4^{v_1}$	$\varphi_7^{v_2}$
$\varphi_7^{v_2}$	1805	$\varphi_4^{v_1}$		$\varphi_4^{v_1}$	
$\varphi_4^{v_1}$	1875				

6 Conclusion

In this paper, we presented an approach for hierarchical scheduling of periodic dependent singelcore real-time systems on a multicore hardware. We introduced a system model that covers tasks with precedence constraints as well as VMs that host subsets of these tasks. To schedule the set of VMs on a multicore hardware with full hypervisor-based virtualization, we first proposed a concept to decompose local singlecore schedules into fragments based on deadlines, release times and inter-VM dependencies. Afterwards, we presented our approach for offline scheduling of these fragments on a multicore platform. Finally, we applied our approach to an automotive use case from literature to demonstrate functionality of the proposed concept. Future work aims at taking overhead induced by virualization as well as communication costs between VMs into account.

Acknowledgment

This work was partly funded by German Ministry of Education and Research (BMBF) through project “it’s OWL Intelligente Technische Systeme OstWestfalenLippe” (02PQ1021) and ITEA2 project AMALTHEA4public (01IS14029J).

References

1. Baruah, S., Bertogna, M., Buttazzo, G.: Multiprocessor Scheduling for Real-Time Systems. Springer (2015)
2. Chetto, H., Silly, M., Bouchentouf, T.: Dynamic scheduling of real-time tasks under precedence constraints. Real-Time Systems 2(3), 181–194 (1990)
3. Cho, H., Ravindran, B., Jensen, E.: An optimal real-time scheduling algorithm for multiprocessors. In: 27th IEEE International Real-Time Systems Symposium. pp. 101–110 (2006)
4. Crespo, A., Ripoll, I., Masmano, M.: Partitioned embedded architecture based on hypervisor: The xtratum approach. In: European Dependable Computing Conference (EDCC). pp. 67–72 (2010)
5. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. 43(4), 35:1–35:44 (Oct 2011)
6. Forget, J., Boniol, F., Grolleau, E., Lesens, D., Pagetti, C.: Scheduling dependent periodic tasks without synchronization mechanisms. In: 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 301–310 (2010)
7. Groesbrink, S., Almeida, L.: A criticality-aware mapping of real-time virtual machines to multi-core processors. In: IEEE Emerging Technology and Factory Automation (ETFA). pp. 1–9 (2014)
8. Kandasamy, N., Hayes, J., Murray, B.: Dependable communication synthesis for distributed embedded systems. In: Anderson, S., Felici, M., Littlewood, B. (eds.) Computer Safety, Reliability, and Security, Lecture Notes in Computer Science, vol. 2788, pp. 275–288. Springer Berlin Heidelberg (2003)
9. Kleidermacher, D.: System virtualization in multicore systems. In: Moyer, B. (ed.) Real World Multicore Embedded Systems – A Practical Approach, pp. 227–267. Elsevier (2013)
10. Klobedanz, K., Jatzkowski, J., Rettberg, A., Mueller, W.: Fault-tolerant deployment of real-time software in autosar ecu networks. In: Schirner, G., Götz, M., Rettberg, A., Zanella, M., Rammig, F. (eds.) Embedded Systems: Design, Analysis and Verification, IFIP Advances in Information and Communication Technology, vol. 403, pp. 238–249. Springer Berlin Heidelberg (2013)
11. Lee, S.K.: On-line multiprocessor scheduling algorithms for real-time tasks. In: Proceedings of TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. pp. 607–611 (1994)
12. Main, C.: Virtualization on multicore for industrial real-time operating systems [from mind to market]. Industrial Electronics Magazine, IEEE 4(3), 4–6 (2010)
13. Masmano, M., Ripoll, I., Crespo, A.: Xtratum: A hypervisor for safety critical embedded systems. In: Proceedings of 11th Real-Time Linux Workshop. pp. 263–272 (2009)
14. Masrur, A., Drossler, S., Pfeuffer, T., Chakraborty, S.: Vm-based real-time services for automotive control applications. In: IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 218–223 (2010)
15. Rönngren, S., Shirazi, B.: Static multiprocessor scheduling of periodic real-time tasks with precedence constraints and communication costs. In: Proceedings of 28th Hawaii International Conference on System Sciences. vol. 2, pp. 143–152 (1995)
16. Xi, S., Xu, M., Lu, C., Phan, L., Gill, C., Sokolsky, O., Lee, I.: Real-time multi-core virtual machine scheduling in xen. In: International Conference on Embedded Software (EMSOFT). pp. 1–10 (2014)